10

Os cuidados com arrays usando boas praticas.

Matrizes e memória

Teletransportando fantasmas: cuidados a tomar com a memória

Ainda falta um pouco mais de inteligência em nossos fantasmas. Eles tem que andar para qualquer uma das quatro direções, aleatoriamente, e sempre lembrando que só podem andar em espaços válidos.

Como implementar isso? Teremos que mudar nosso `move_fantasma` para escolher uma nova `posicao` válida. Nosso código atual simplesmente move o fantasma para a direita:

```
def move_fantasma(mapa, linha, coluna)
    posicao = [linha, coluna + 1]
    if posicao_valida? mapa, posicao
        mapa[linha][coluna] = " "
        mapa[posicao[0]][posicao[1]] = "F"
    end
end
```

Primeiro calculamos um lado:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    baixo = mapa[posicao[0] + 1][posicao[1]]
end
```

Agora verificamos se ela é válida, adicionando ao nosso array:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    baixo = mapa[posicao[0] + 1][posicao[1]]
    if posicao_valida? baixo
        posicoes << baixo
    end
end
```

Fazemos a mesma coisa para a direita:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    baixo = mapa[posicao[0] + 1][posicao[1]]
    if posicao_valida? baixo
```

```
            posicoes << baixo
        end
        direita = mapa[posicao[0]][posicao[1] + 1]
        if posicao_valida? direita
            posicoes << direita
        end
    end
```

Para cima e para baixo:

```
    def posicoes_validas_a_partir_de(mapa, posicao)
        posicoes = []
        # ...
        cima = mapa[posicao[0] - 1][posicao[1]]
        if posicao_valida? cima
            posicoes << cima
        end
        esquerda = mapa[posicao[0]][posicao[1] - 1]
        if posicao_valida? esquerda
            posicoes << esquerda
        end
    end
```

Por fim, retornamos nosso array:

```
    def posicoes_validas_a_partir_de(mapa, posicao)
        posicoes = []
        # ...
        posicoes
    end
```

Já temos nossas posições válidas para o fantasma. Agora precisamos utilizá-las:

```
    def move_fantasma(mapa, linha, coluna)
        posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
        posicao = # escolhe uma posicao

        mapa[linha][coluna] = " "
        mapa[posicao[0]][posicao[1]] = "F"
    end
```

Faremos o fantasma seguir a primeira posição possível:

```
    def move_fantasma(mapa, linha, coluna)
        posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
        posicao = posicoes[0]

        mapa[linha][coluna] = " "
```

```
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Mas precisamos verificar que existe alguma posição para mover, claro. Perguntamos ao nosso array se ele está vazio:

```
def move_fantasma(mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Testamos agora e os fantasmas andam, um para baixo, um para a direita:

```
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
Para onde deseja ir?
D
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
XXX    FX
XXX XXX X
XXX XXX X
XXX F   X
```

Corrigindo o teletransporte

Mas aconteceu muito mais do que isso. O fantasma da esquerda se teletransportou? Teletransporte vale? Não vale. O que está acontecendo? Vamos analisar melhor o que acontece com nossa matriz a cada movimento de um fantasma. No `move_fantasma` vamos imprimir nosso mapa temporário:

```
def move_fantasma(mapa, linha, coluna)
    # ...

    puts "Movendo fantasma encontrado em #{linha} #{coluna}"
    desenha mapa
end
```

Rodamos o jogo e acompanhamos o primeiro movimento:

O laço de mover fantasmas percorre nosso array de `String`s até encontrar nosso primeiro fantasma na décima linha, quarta coluna (`9 , 3`). Ele pode andar para baixo, portanto nosso algoritmo de movimento desloca ele para baixo:

```
Movendo fantasma encontrado em 9 3
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
XXX   F X
XXXFXXX X
XXX XXX X
XXX     X
```

Aí o programa continua varrendo a linha, encontra nosso segundo fantasma, na mesma linha, sétima coluna (`6`), que não pode ir para baixo, e acaba por ir para a direita:

```
Movendo fantasma encontrado em 9 6
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
XXX    FX
XXXFXXX X
XXX XXX X
XXX     X
```

Até aqui o algoritmo roda como o esperado. O que acontece a seguir é surpreendente:

```
Movendo fantasma encontrado em 10 3
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
XXX    FX
XXX XXX X
XXXFXXX X
XXX     X
```

O nosso algoritmo procura fantasmas na linha seguinte - e encontra o primeiro novamente, movendo ele mais uma vez para baixo! E o processo continua, descendo, descendo:

```
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
XXX    FX
XXX XXX X
XXX XXX X
XXXF    X
```

Acontece que ao utilizarmos o mesmo pedaço de memória, a mesma matriz, para representar duas coisas, e fazer com que ele possa mudar seu valor, isto é, ele seja mutável (::mutable::), permitimos que nosso algoritmo funcionasse de uma maneira que não desejássemos. Ele altera a matriz de posições atuais, mas a matriz passa a representar tanto o futuro quanto o passado de nossos fantasmas. Valores mutáveis (e matrizes costumam ser tratadas assim) são perigosas por isso: tome cuidado ao alterá-las.

Como resolver nosso problema? Precisamos separar o mapa que representa a situação dos fantasmas no início da rodada, da situação do mapa que representa os fantasmas após seus movimentos. Como fazer isso? Podemos criar uma nova array de `String` s, só com os muros e os heróis, e copiar os fantasmas um a um, a medida que eles se movem.

Se olharmos a matriz antiga para procurar os fantasmas, encontraremos eles somente uma vez, evitando o bug atual.

Mas espera um instante, Guilherme. O fantasma desceu tudo, mas como pode ser que ele não foi tudo para a direita? Repare que no fim do turno ele termina em:

```
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
XXX    FX
XXX XXX X
XXX XXX X
XXX F   X
```

Acontece que ao invocarmos a função `chars` de nossa `String` para iterarmos com o `each_with_index`, o Ruby tira uma cópia de nossa array de caracteres. Portanto qualquer mudança que façamos no array antigo não afeta o novo array. Isto é, de graça já fizemos a cópia que teremos que fazer com as linhas.

Removemos então as duas linhas que imprimiam nossas linhas de informação, voltando a versão antiga:

```
def move_fantasma(mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Copiando nosso mapa

Vamos então copiar nosso mapa para resolver o problema dos fantasmas teletransportadores. Ao iniciar o movimento dos fantasmas copiamos nosso array. Poderíamos fazer dois laços aninhados, dois `for` s passando por cada elemento e copiando-os um a um:

```
def copia_mapa(mapa)
    novo_mapa = []
    mapa.each do |linha|
        nova_linha = ""
        linha.chars.each do |caractere|
            nova_linha << caractere
        end
        novo_mapa << nova_linha
    end
    novo_mapa
end
```

Precisaríamos ainda verificar se o elemento é um fantasma, removendo ele:

```
def copia_mapa(mapa)
    novo_mapa = []
    mapa.each do |linha|
        nova_linha = ""
        linha.chars.each do |caractere|
            if caractere == "F"
                nova_linha << " "
            else
                nova_linha << caractere
            end
        end
        novo_mapa << nova_linha
    end
    novo_mapa
end
```

Que horror. Ao invés disso podemos utilizar aquela função que já conhecíamos que duplica um valor que a suporte, o `dup` , e uma outra função que troca, traduz (::tr::), os `F` s por espaços:

```
def copia_mapa(mapa)
    novo_mapa = []
    mapa.each do |linha|
        nova_linha = linha.dup.tr "F", " "
        novo_mapa << nova_linha
    end
    novo_mapa
end
```

Ainda está bem grande este código. Repare que se podemos trocar `"F"` s por `" "` , podemos juntar todo o mapa em uma única `String` , fazendo um ::join::. O `join` junta diversas `Strings` :

```
nomes = ["guilherme", "de", "azevedo", "silveira"]
puts nomes.join(" ") # guilherme de azevedo silveira
puts nomes.join("\n") # guilherme
# de
# azevedo
# silveira
```

Portanto juntamos nosso mapa em uma única `String` :

```
def copia_mapa(mapa)
  texto = mapa.join("\n")
  # e agora?
end
```

Agora trocamos todos de uma vez, e quebramos novamente:

```
def copia_mapa(mapa)
  novo_mapa = mapa.join("\n").tr("F", " ").split("\n")
end
```

Movendo os fantasmas na matriz copiada

Nossa função que move os fantasmas precisa agora efetuar as mudanças no novo mapa:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    novo_mapa = copia_mapa mapa

    # ...
end
```

E ao mover o fantasma enviaremos tanto o mapa antigo quanto o novo:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    novo_mapa = copia_mapa mapa
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.chars.each_with_index do |caractere_atual, coluna|
            eh_fantasma = caractere_atual == caracter_do_fantasma
            if eh_fantasma
                move_fantasma mapa, novo_mapa, linha, coluna
            end
        end
    end
end
```

Portanto nossa função `move_fantasma` deve receber os dois mapas:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Olhando o código anterior, onde devemos trocar o `mapa` pelo `novo_mapa`? Primeiro avaliamos as posições válidas. Só é válido ir para uma posição se não tem nenhum fantasma lá, e se nenhum se moveu pra lá no novo mapa. Portanto temos que verificar os dois mapas, não só um deles. Passemos então os dois mapas como argumento:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, novo_mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Alteramos também a função `posicoes_validas_a_partir_de` para verificar se os dois mapas tem uma posição válida:

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    baixo = [posicao[0] + 1, posicao[1]]
    if posicao_valida? mapa, baixo && posicao_valida? novo_mapa, baixo
        posicoes << baixo
    end
    direita = [posicao[0], posicao[1] + 1]
    if posicao_valida? mapa, direita && posicao_valida? novo_mapa, direita
        posicoes << direita
    end
    cima = [posicao[0] - 1, posicao[1]]
    if posicao_valida? mapa, cima && posicao_valida? novo_mapa, cima
        posicoes << cima
    end
    esquerda = [posicao[0], posicao[1] - 1]
    if posicao_valida? mapa, esquerda && posicao_valida? novo_mapa, esquerda
        posicoes << esquerda
    end
    posicoes
end
```

Um fantasma só pode andar para posições onde no novo mapa não há nenhum outro fantasma, portanto na primeira invocação a `posicoes_validas_a_partir_de` devemos usar o novo mapa.

Por fim, devemos desenhar o fantasma no novo mapa, portanto:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, novo_mapa, [linha, coluna]
    if posicoes.empty?
        return
    end
```

```
    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    novo_mapa[posicao[0]][posicao[1]] = "F"
end
```

Mas se alteramos um array dentro da função `move_fantasma` será que ele é trocado fora da função também? Pensemos no ::stack trace:::

```
move_fantasma (mapa = ???, novo_mapa = ???)
move_fantasmas (mapa = ???, novo_mapa = ???)
...
```

Qual é o valor de um array? Já vimos que uma variável que referencia um array é somente um valor, e o array em si está em um único canto da memória. Para um mapa mais simples, teríamos:

```
move_fantasma (mapa = 123, novo_mapa = 456)
move_fantasmas (mapa = 123, novo_mapa = 456)
...
Em um canto da memoria:

123: [XXX]
     [ F ]
     [XXX]
456: [XXX]
     [   ]
     [XXX]
```

Se alterarmos uma posição na referência `456` , "ambas" referências são alteradas. Na verdade nenhuma é alterada, ambas continuam referenciando, apontando, para o mesmo valor da memória:

```
move_fantasma (mapa = 123, novo_mapa = 456)
move_fantasmas (mapa = 123, novo_mapa = 456)
...
Em um canto da memoria:

123: [XXX]
     [ F ]
     [XXX]
456: [XXX]
     [  F]
     [XXX]
```

Portanto devemos sempre nos lembrar: se o valor de uma variável é um valor numérico simples, passar ele como parâmetro passa o valor em si. Se ele é um valor mais complexo, um objeto, como `String` , `Array` ou outro, temos somente uma referência para esse valor mais complexo na memória.

Testamos então o jogo:

```
main.rb:1:in `require_relative': fogefoge.rb:56:
    syntax error, unexpected tIDENTIFIER, expecting keyword_do
                        or '{' or '(' (SyntaxError)
    if posicao_valida? mapa, baixo && posicao_valida? novo_mapa, baixo
```

O Ruby se perdeu com nossa invocação a função `posicao_valida` . Note que como não usamos parenteses para a invocação, ele se perdeu com o `&&` e uma nova invocação de função logo após ele. Como foi mencionado anteriormente, em algumas situações somos obrigados a usar o parênteses. Mas isso já é um ótimo indício de que nosso código está horrível. Atacaremos esse problema a seguir. Agora, corrigimos a invocação:

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    baixo = [posicao[0] + 1, posicao[1]]
    if posicao_valida?(mapa, baixo) && posicao_valida?(novo_mapa, baixo)
        posicoes << baixo
    end
    direita = [posicao[0], posicao[1] + 1]
    if posicao_valida?(mapa, direita) && posicao_valida?(novo_mapa, direita)
        posicoes << direita
    end
    cima = [posicao[0] - 1, posicao[1]]
    if posicao_valida?(mapa, cima) && posicao_valida?(novo_mapa, cima)
        posicoes << cima
    end
    esquerda = [posicao[0], posicao[1] - 1]
    if posicao_valida?(mapa, esquerda) && posicao_valida?(novo_mapa, esquerda)
        posicoes << esquerda
    end
    posicoes
end
```

Rodamos o código:

```
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
XXX     X
XXX XXX X
XXX XXX X
XXX     X
```

Agora os fantasmas se foram? Claro! Criamos o nosso novo array, de fim de jogada, mas não trocamos ele pelo antigo! Quando o turno acaba, devemos dizer que o array novo deve ser utilizado para o próximo turno. Fazemos então o

`move_fantasmas` retornar o novo mapa:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    novo_mapa = copia_mapa mapa
    # ...
    novo_mapa
end
```

E ao jogar, trocamos ele:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...

        mapa = move_fantasmas mapa
    end
end
```

Rodamos novamente, agora nossos fantasmas nem desaparecem nem se teletransportam:

```
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
XXX    FX
XXXFXXX X
XXX XXX X
XXX     X
```

Refatorando o movimento do fantasma

O único problema é que nosso `posicoes_validas_a_partir_de` está horrendo.

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    baixo = [posicao[0] + 1, posicao[1]]
    if posicao_valida?(mapa, baixo) && posicao_valida?(novo_mapa, baixo)
        posicoes << baixo
    end
    direita = [posicao[0], posicao[1] + 1]
```

```
    if posicao_valida?(mapa, direita) && posicao_valida?(novo_mapa, direita)
        posicoes << direita
    end
    cima = [posicao[0] - 1, posicao[1]]
    if posicao_valida?(mapa, cima) && posicao_valida?(novo_mapa, cima)
        posicoes << cima
    end
    esquerda = [posicao[0], posicao[1] - 1]
    if posicao_valida?(mapa, esquerda) && posicao_valida?(novo_mapa, esquerda)
        posicoes << esquerda
    end
    posicoes
end
```

Assim como fizemos uma refatoração da movimentação do jogador, esse código parece ter um padrão ao testar cada posição ao redor de nosso fantasma. Podemos então criar um array simples com os valores adequados para cada uma das quatro posições:

```
movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
```

E para cada uma das posições possíveis:

```
movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
movimentos.each do |movimento|

end
```

Calculamos a posição nova:

```
movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
movimentos.each do |movimento|
    nova_posicao = [posicao[0] + movimento[0], posicao[1] + movimento[1]]
    if posicao_valida?(mapa, nova_posicao) && posicao_valida?(novo_mapa, nova_posicao)

    end
end
```

Verificamos se ela é válida, acumulando ela em um array que será retornado no fim de nossa função:

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
    movimentos.each do |movimento|
        nova_posicao = [posicao[0] + movimento[0], posicao[1] + movimento[1]]
        if posicao_valida?(mapa, nova_posicao) && posicao_valida?(novo_mapa, nova_posicao)
            posicoes << nova_posicao
        end
```

```
    end
    posicoes
end
```

Podemos extrair também uma função de soma de vetores, que soma as duas posições:

```
def soma(vetor1, vetor2)
    [vetor1[0] + vetor2[0], vetor1[1] + vetor2[1]]
end

def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
    movimentos.each do |movimento|
        nova_posicao = soma posicao, movimento
        if posicao_valida?(mapa, nova_posicao) && posicao_valida?(novo_mapa, nova_posicao)
            posicoes << nova_posicao
        end
    end
    posicoes
end
```

O fantasma cavaleiro

E se o nosso fantasma se comportar como uma peça do tipo cavalo no jogo de xadrez? Isto é, ao invés de ir para cima, baixo, esquerda e direita, queremos que ele faça movimentos do tipo `L`. O exemplo a seguir mostra a posição do fantasma e marcando com * as oito posições onde ele poderia ir após um único movimento:

```
XXXXXXX
X * * X
X     X
X*   *X
X  F  X
X*   *X
X     X
X * * X
XXXXXXX
```

Como implementar essa lógica? Note que são oito movimentos válidos:

```
    movimentos = [[-2, -1], [-2, +1], [+2, -1], [+2, +1],
                  [-1, -2], [-1, +2], [+1, -2], [+1, +2]]
```

Pronto, os fantasmas agora andam como cavalos do xadrez. Com esse tipo de técnica simples conseguimos inclusive implementar movimentos complexos de peças de tabuleiro em espaços 3d! Bastaria adicionar a terceira dimensão. Claro, no nosso jogo o fantasma não é um cavalo, mas se quiser dificultar o jogo depois, pode alterar os movimentos válidos do fantasma facilmente.

Movimento aleatório dos fantasmas

Precisamos escolher uma das posições, aleatoriamente. Já sabemos fazer isso com o uso do `rand`:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, novo_mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    aleatoria = rand posicoes.size
    posicao = posicoes[aleatoria]

    mapa[linha][coluna] = " "
    novo_mapa[posicao[0]][posicao[1]] = "F"
end
```

Pronto. Podemos testar o jogo duas vezes e ver que cada vez os fantasmas se locomovem de maneira diferente. Note que sua saída pode (e provavelmente será) diferente da minha. Começando o jogo e movendo o herói uma vez:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X  FX X X
XXX  F  X
XXX XXX X
XXX XXX X
XXX     X
```

E começando novamente o jogo, movendo uma primeira vez:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X  FX X X
XXX    FX
XXX XXX X
XXX XXX X
XXX     X
```

Quando o herói perde

Uma outra situação importante que ainda não definimos é quando o nosso herói entra em contato com um fantasma. Isso pode ocorrer quando o turno acaba e tanto o fantasma quanto o herói estão na mesma posição. Mas como descobrir que isso ocorreu?

Como o fantasma é sempre o último a andar, ele vai aparecer "por cima" do herói: o jogador sumirá do mapa. Portanto se não encontramos o jogador no mapa, sabemos que ele perdeu o jogo.

Nossa função que encontra um jogador deve agora retornar que não encontrou, vazio, nulo:

```
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
    nil
end
```

Ao terminar o turno, verificamos se o jogador não está mais no mapa e, se for o caso, mostramos a mensagem de ::Game Over::.

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...
        mapa = move_fantasmas mapa
        if encontra_jogador(mapa) == nil
            game_over
            break
        end
    end
end
```

Em Ruby, verificar se um valor é nulo é a mesma coisa que verificar que ele não existe:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...
        mapa = move_fantasmas mapa
        if !encontra_jogador(mapa)
            game_over
            break
        end
    end
end
```

Mas nossa função está bem feia. Novamente temos uma linha que faz muita coisa: `if` , `!` e invoca uma função. Vamos extrair o código, lembre-se que a variável é desnecessária e só é utilizada para deixar claro o que estamos retornando:

```
def jogador_perdeu?(mapa)
    perdeu = !encontra_jogador(mapa)
end

def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...
        mapa = move_fantasmas mapa
        if jogador_perdeu?(mapa)
            game_over
            break
        end
    end
end
```

Criamos também a função de `game_over` na nossa interface:

```
def game_over
    puts "\n\n\n\n\n\n"
    puts "Game Over"
end
```

Pronto, podemos jogar até o momento que perdemos:

```
XXXXXXXXX
X       X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
 FX F   X
X  HX X X
XXX     X
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
W
```

```
Game Over
```

Como o movimento de dois fantasmas é aleatório, demora para perdemos.

Retorno nulo ou opcional?

Nossa função `encontra_jogador` retorna nada caso ele não encontre um jogador no mapa. O `nil` é muito útil para marcar que uma função que retorna algo pode, sob determinadas circunstâncias, retornar nada. O problema acontece ao usar esse retorno. Como o desenvolvedor pode escrever código que invoca a função e faz qualquer coisa com o retorno, ele não é obrigado a verificar se o valor retornado é vazio. Se não for vazio, a aplicação pode parar com um erro ou ainda se comportar de maneira totalmente inesperada.

Algumas linguagens de programação funcionais como Swift apresentam um conceito de opcional, outras linguagens orientadas a objeto como Java oferecem tal funcionalidade, que já obriga o desenvolvedor a verificar se o retorno é válido antes de utilizá-lo.

Resumindo

Vimos como é importante cuidar direitinho de nossas referências na memória. Um valor como um array pode ser alterado por fora e por dentro de uma função, causando bagunça naquilo que queremos representar. Em jogos baseados em turno (::turn-based::), assim como em processos que devem executar algo baseado em um estado, é comum que o estado seja representado por um valor e a cada novo estado tiremos uma cópia inteira dele. Podemos até voltar ao estado anterior (::undo::) com tal prática - que tem nome, ::Memento::! O cuidado que devemos tomar é com a memória, claro.

Implementamos o movimento aleatório e o fim do jogo quando o herói perde para um fantasma. Em diversos desses instantes utilizamos práticas típicas de um programador de jogos ou de maratona de programação, onde usa-se arrays para trabalhar como atalhos para determinados resultados.

Como veremos adiante, estruturas, classes e objetos podem representar o mesmo tipo de informação, mas com mais valor de significado (um mapa não é um array, é um mapa - um herói não é um array de tamanho 2, é um herói em uma linha e uma coluna), algo que pode ser benéfico para nosso código a longo prazo.