Write a function to reverse a string in-place.⃕

Breakdown

In general, an in-place⃕

An **in-place** algorithm operates *directly* on its input and *changes* it, instead of creating and returning a *new* object. This is sometimes called **destructive**, since the original input is "destroyed" when it's edited to create the new output.

Careful: "In-place" does *not* mean "without creating any additional variables!" Rather, it means "without creating a new copy of the input." In general, an in-place function will only create additional variables that are $O(1)$ space.

Here are two functions that do the same operation, except one is in-place and the other is out-of-place:

Ruby ▾

```
def square_array_in_place(int_array)

    int_array.each_with_index do |element, index|
        int_array[index] *= element
    end

    # NOTE: we could make this function return nil,
    # since we modify int_array in place.
    return int_array
end

def square_array_out_of_place(int_array)

    # we allocate a new array with the length of the input array
    squared_array = [nil] * int_array.length

    int_array.each_with_index do |element, index|
        squared_array[index] = element ** 2
    end

    return squared_array
end
```

Working in-place is a good way to save space. An in-place algorithm will generally have $O(1)$ space cost.

But be careful: an in-place algorithm can cause side effects. Your input is "destroyed" or "altered," which can affect code *outside* of your function. For example:

Ruby

```ruby
original_array = [2, 3, 4, 5]
squared_array  = square_array_in_place(original_array)

puts "squared: #{squared_array}"
# prints: squared: [4, 9, 16, 25]

puts "original array: #{original_array}"
# prints: original array: [4, 9, 16, 25], confusingly!

# and if square_array_in_place() didn't return anything,
# which it could reasonably do, squared_array would be nil!
```

Generally, out-of-place algorithms are considered safer because they avoid side effects. You should only use an in-place algorithm if you're very space constrained or you're *positive* you don't need the original input anymore, even for debugging.

algorithm will require swapping elements.

Solution

We swap the first and last characters, then the second and second-to-last characters, and so on until we reach the middle.

Ruby

```ruby
def reverse(string)

    left_pointer  = 0
    right_pointer = string.length - 1

    while left_pointer < right_pointer

        # swap characters
        string[left_pointer], string[right_pointer] = \
            string[right_pointer], string[left_pointer]

        # move towards middle
        left_pointer  += 1
        right_pointer -= 1

    end

    return string
end
```

Complexity

$O(n)$ time and $O(1)$ space.