```
    if *item == needle {
      println!("{}", item);
    }
  }
}
```

The syntax *item returns the item's referent.

Each iteration changes the value of `item` to refer to the next item within `haystack`. On the first iteration, `*item` returns `1`, and on the last, it returns `21147`.

2.7 Project: Rendering the Mandelbrot set

So far, we haven't learned much Rust, but we already have the tools to create some interesting pictures of fractals. So let's do that now with listing 2.12. To begin

1. In a terminal window, execute the following commands to create a project that can render the Mandelbrot set:
   a. `cd $TMP` (or `cd %TMP%` on MS Windows) to move to a directory that's not critical.
   b. `cargo new mandelbrot --vcs none` creates a new blank project.
   c. `cd mandelbrot` moves into the new project root.
   d. `cargo add num` to edit Cargo.toml, adding the num crate as a dependency (see the sidebar entitled "2.2" in section 2.3.4 for instructions to enable this cargo feature).
2. Replace `src/main.rs` with the code in listing 2.12, which you'll also find in ch2/ch2-mandelbrot/src/main.rs.
3. Execute `cargo run`. You should see the Mandelbrot set rendered in the terminal:

```
              .................................●●●*●**●............
           ...................................●●●***●●●...............
         ..................................●●●***+%+***●................
       ...............................●●●●●●●●*$%%%%%*●●●●●●..............
      .............................●●**+*●●*******%%*****+●●●●●*●..........
    .............................●●●●●*%%+*%%%%%%%%%%%%%%%x*+*+*●●...........
   ............................●●●●●**++%%%%%%%%%%%%%%%%%%%%%%**●●............
  ................●*●●●●●●●●●●●●●●●*+%%%%%%%%%%%%%%%%%%%%%%%%%%*●●●............
  ...............●●●***●●●**●●●●●●*+%%%%%%%%%%%%%%%%%%%%%%%%%%%%%+●............
 ................●●●●*+%*%#xx%****x%%%%%%%%%%%%%%%%%%%%%%%%%%%%%**●.............
 ..............●●●●*++%%%%%%%%%%+*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*●..............
 .......●●●●●●●●**+**%%%%%%%%%%%%+%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*●●..............
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*●●●●..............
 .......●●●●●●●●**+**%%%%%%%%%%%%+%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*●●..............
 ..............●●●●*++%%%%%%%%%%+*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*●..............
 ................●●●●*+%*%#xx%****x%%%%%%%%%%%%%%%%%%%%%%%%%%%%%**●.............
  ...............●●●***●●●**●●●●●●*+%%%%%%%%%%%%%%%%%%%%%%%%%%%%%+●............
  ................●*●●●●●●●●●●●●●●●*+%%%%%%%%%%%%%%%%%%%%%%%%%%*●●●............
   ............................●●●●●**++%%%%%%%%%%%%%%%%%%%%%%**●●............
    .............................●●●●●*%%+*%%%%%%%%%%%%%%%x*+*+*●●...........
      .............................●●**+*●●*******%%*****+●●●●●*●..........
       ...............................●●●●●●●●*$%%%%%*●●●●●●..............
         ..................................●●●***+%+***●................
           ...................................●●●***●●●...............
```

Listing 2.12 Rendering the Mandelbrot set

```
use num::complex::Complex;

fn calculate_mandelbrot(

  max_iters: usize,
  x_min: f64,
  x_max: f64,
  y_min: f64,
  y_max: f64,
  width: usize,
  height: usize,
  ) -> Vec<Vec<usize>> {

  let mut rows: Vec<_> = Vec::with_capacity(width);
  for img_y in 0..height {

    let mut row: Vec<usize> = Vec::with_capacity(height);
    for img_x in 0..width {

      let x_percent = (img_x as f64 / width as f64);
      let y_percent = (img_y as f64 / height as f64);
      let cx = x_min + (x_max - x_min) * x_percent;
      let cy = y_min + (y_max - y_min) * y_percent;
      let escaped_at = mandelbrot_at_point(cx, cy, max_iters);
      row.push(escaped_at);
    }

    all_rows.push(row);
  }
  rows
}

fn mandelbrot_at_point(
  cx: f64,
  cy: f64,
  max_iters: usize,
  ) -> usize {
  let mut z = Complex { re: 0.0, im: 0.0 };
  let c = Complex::new(cx, cy);

  for i in 0..=max_iters {
    if z.norm() > 2.0 {
      return i;
    }
    z = z * z + c;
  }
  max_iters
}

```

Imports the Complex number type from num crate and its complex submodule

Converts between the output space (a grid of rows and columns) and a range that surrounds the Mandelbrot set (a continuous region near (0,0))

If a value has not escaped before reaching the maximum number of iterations, it's considered to be within the Mandelbrot set.

Parameters that specify the space we're searching for to look for members of the set

Parameters that represent the size of the output in pixels

Creates a container to house the data from each row

Iterates row by row, allowing us to print the output line by line

Calculates the proportion of the space covered in our output and converts that to points within the search space

Called at every pixel (e.g., every row and column that's printed to stdout)

Initializes a complex number at the origin with real (re) and imaginary (im) parts at 0.0

Initializes a complex number from the coordinates provided as function arguments

Checks the escape condition and calculates the distance from the origin (0, 0), an absolute value of a complex number

Repeatedly mutates z to check whether c lies within the Mandelbrot set

As i is no longer in scope, we fall back to max_iters.

```
fn render_mandelbrot(escape_vals: Vec<Vec<usize>>) {
  for row in escape_vals {
    let mut line = String::with_capacity(row.len());
    for column in row {
      let val = match column {
        0..=2 => ' ',
        2..=5 => '.',
        5..=10 => '•',
        11..=30 => '*',
        30..=100 => '+',
        100..=200 => 'x',
        200..=400 => '$',
        400..=700 => '#',
        _ => '%',
      };

      line.push(val);
    }
    println!("{}", line);
  }
}

fn main() {
  let mandelbrot = calculate_mandelbrot(1000, 2.0, 1.0, -1.0,
                                        1.0, 100, 24);

  render_mandelbrot(mandelbrot);
}
```

So far in this section, we've put the basics of Rust into practice. Let's continue our exploration by learning how to define functions and types.

2.8 Advanced function definitions

Rust's functions can get somewhat scarier than the `add(i: i32, j: i32) -> i32` from listing 2.2. To assist those who are reading more Rust source code than writing it, the following sections provide some extra content.

2.8.1 Explicit lifetime annotations

As a bit of forewarning, allow me to introduce some more complicated notation. As you read through Rust code, you might encounter definitions that are hard to decipher because those look like hieroglyphs from an ancient civilizations. Listing 2.13 provides an extract from listing 2.14 that shows one such example.

Listing 2.13 A function signature with explicit lifetime annotations

```
fn add_with_lifetimes<'a, 'b>(i: &'a i32, j: &'b i32) -> i32 {
  *i + *j
}
```

Like all unfamiliar syntax, it can be difficult to know what's happening at first. This improves with time. Let's start by explaining *what* is happening, and then go on to