Vec and HashMap

Rust basics · Vec and HashMap

In PHP

One array does both jobs. `$numbers[] = 10` grows a list. `$scores["Ada"] = 10` stores a value under a string key. Keys keep insertion order. A missing key is handled with `??`.

counts.php

```
 <?php

$numbers = [];
$numbers[] = 10;
$numbers[] = 30;
$numbers[] = 20;
echo $numbers[0], "\n";

$counts = [];
foreach (preg_split("/\s+/", "the cat and the dog") as $word) {
    $counts[$word] = ($counts[$word] ?? 0) + 1;
}
foreach ($counts as $word => $count) {
    echo "$word $count\n";
}
```

The counts print in insertion order: `the`, `cat`, `and`, `dog`.

`array_map` and `array_filter` become iterator chains. Iterators have their own chapter. A nested array used as a record becomes a struct (chapter 4). JSON arrives through serde, in the API chapter.

In Rust

A list is a `Vec<T>`. `vec![]` starts it. `push`, `pop`, `len`, `is_empty`, and `contains` are the usual operations. Every element has the same type. Mixed types need an enum (chapter 4).

`numbers[0]` reads an element. On a `Vec`, `numbers[9]` compiles and then panics, because the length is not part of the type:

```
 thread 'main' panicked at src/main.rs:3:27:
index out of bounds: the len is 3 but the index is 9
```

`numbers.get(9)` is `None`. `for n in &numbers` reads. `for n in &mut numbers` writes. `sort` orders the values. `sort_by_key` orders by one field.

A string-keyed array is a `HashMap<String, V>`. Build the key with `.to_string()`. `insert` stores it. `get("Ada")` looks it up and returns `Option`. `contains_key` and `remove` do what they say. Walking the map visits every pair.

`*n += 1` and `*counts.entry(...)`: `*` writes through to the value the loop or entry points at. The Rust side of `$counts[$word] = ($counts[$word] ?? 0) + 1` is `*counts.entry(word.to_string()).or_insert(0) += 1`. `entry(...).or_default()` is the same idea when the value type has a useful `Default` (for example `0` for integers). `retain` keeps only matching entries. `extend` appends from another iterator. The `String` is how the map owns the word. `get("Ada")` looks up with the text of the key.

A set of unique values is a `HashSet`. PHP often fakes that with `array_unique` or `array_flip` so the keys act as the set. Rust has a real set type: `use std::collections::HashSet;` then `HashSet::from([1000, 2500, 800])`.

> **Watch for:** A PHP array keeps insertion order. `HashMap` iteration order is unspecified, and it can change between runs. `BTreeMap` walks keys in sorted order. The `indexmap` crate keeps insertion order when that is the behaviour you need.

`contains(&30)`: `&` lends the value for the call. Chapter 3 explains. `|row| row.1` is a closure, a small inline function. Closures get their own lesson.

src/main.rs

```
 use std::collections::{BTreeMap, HashMap};

fn main() {
    let mut numbers: Vec<i32> = vec![];
    let empty = numbers.is_empty();
    println!("{empty}");
    numbers.push(10);
    numbers.push(30);
    numbers.push(20);
```

```
    let len = numbers.len();
    let has_30 = numbers.contains(&30);
    let first = numbers[0];
    println!("{len} {has_30} {first}");
    let missing = numbers.get(9);
    println!("{missing:?}");

    for n in &numbers {
        println!("item {n}");
    }
    for n in &mut numbers {
        *n += 1;
    }
    numbers.sort();
    println!("{numbers:?}");
    let last = numbers.pop();
    println!("{last:?}");

    let mut rows = vec![("pear", 2), ("apple", 1)];
    rows.sort_by_key(|row| row.1);
    println!("{rows:?}");

    let mut scores = HashMap::new();
    scores.insert("Ada".to_string(), 10);
    scores.insert("Grace".to_string(), 12);
    let ada = scores.get("Ada");
    let has_ada = scores.contains_key("Ada");
    println!("{ada:?} {has_ada}");
    for (name, score) in &scores {
        println!("{name} {score}");
    }
    let removed = scores.remove("Grace");
    println!("{removed:?}");

    let sentence = "the cat and the dog";
    let mut counts = BTreeMap::new();
    for word in sentence.split_whitespace() {
        *counts.entry(word.to_string()).or_insert(0) += 1;
    }
    for (word, count) in &counts {
        println!("{word} {count}");
    }
}
```

`empty` is `true`. After the pushes, the line is `3 true 10`, and `missing` is `None`. The loop prints `item 10`, `item 30`, `item 20`. After adding one and sorting, the vec is `[11, 21, 31]` and `pop` returns `Some(31)`. `sort_by_key` prints `[("apple", 1), ("pear", 2)]`. `get("Ada")` is `Some(10)`. The two names can print in either order. `remove` returns `Some(12)`. The `BTreeMap` prints `and`, `cat`, `dog`, then `the`.

Idiomatic Rust: `Vec` for a list, `HashMap<String, V>` for string keys, `BTreeMap` when the printout has to be sorted. `.get` when the index or the key might be absent.

Side by side

	PHP	Rust
List	`$n[] = 10`	`Vec<T>`, `push`
Read	`$n[0]`, missing is `null`	`[0]` panics. `.get` is `Option`
String keys	`$scores["Ada"]`	`HashMap`, `insert` with `.to_string()`, `get("Ada")`
Count	`($counts[$w] ?? 0) + 1`	`*counts.entry(word.to_string()).o += 1`
Order	Insertion order	`HashMap` is unspecified. `BTreeMap` is sorted

Try it

Open `src/main.rs` in the Playground. The last four lines are the word counts in sorted order: `and 1`, `cat 1`, `dog 1`, `the 2`. Change the sentence so a word repeats, and run it again. That word's count goes up, and the lines stay alphabetical.

Key takeaways

- `Vec<T>` is the list: `push`, `pop`, `len`, `is_empty`, `contains`, `sort`, `sort_by_key`. One element type.
- `v[i]` panics when `i` is out of range. `v.get(i)` returns `Option`. `for n in &v` reads, and `for n in &mut v` writes.

- `HashMap<String, V>` is the associative array. `get` returns `Option`. Build keys with `.to_string()`. `entry().or_insert` / `or_default`, `retain`, and `extend` cover common map edits.
- `HashSet` is the unique-value set (`array_unique` / `array_flip` in PHP).
- `HashMap` order can change between runs. `BTreeMap` prints keys in sorted order.

What's next: Strings at a glance