Arrays, vectors and slices in Rust

2020-10-11

Introduction

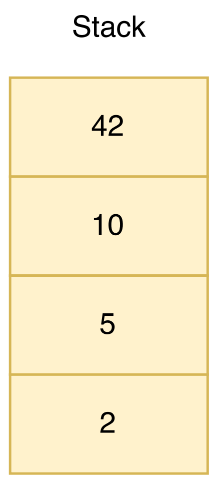
In this post I will introduce you to arrays, vectors and slices in Rust. Programmers coming from C or C++ will already be familiar with arrays and vectors, but because of Rust's focus on safety there are some differences from their unsafe language counterparts. Slices, on the other hand, will entirely be a new, albeit a very useful concept.

Arrays

Arrays are one of the first data types beginner programmers learn. An array is a collection of elements of the same type allocated in a contiguous memory block. For example, if you allocate an array like this:

```rust
let array: [i32; 4] = [42, 10, 5, 2];
```

Then all the `i32` integers are allocated next to each other on the stack:



In Rust, an array's size is part of the type. For example, this code will not compile:

```rust
//error:expected an array with a fixed size of 4 elements,
//found one with 3 elements
let array: [i32; 4] = [0, 1, 2];
```

Rust's strictness also prevents problems like array to pointer decay in C/C++:

```
    cout << "Array size in main function: " << sizeof(arr) << endl;
    print_array_size(&arr);
    return 0;
}
```

The `print_array_size` function prints 8 instead of the expected 20 (5 integers of 4 bytes) because `arr` has decayed from a pointer to an array of 5 integers to just a pointer to an integer. Similar code in Rust does the right thing:

```
use std::mem::size_of_val;

fn print_array_size(arr: [i32; 5]) {
    //prints 20
    println!("Array size in print_array_size function: {}", size_of_va
}

fn main() {
    let arr: [i32; 5] = [1, 2, 3, 4, 5];
    //print 20
    println!("Array size in main function: {}", size_of_val(&arr));
    print_array_size(arr);
}
```

Another difference between an array in C/C++ and Rust is that accessing elements in Rust does bounds checking. For example, in the following C++ code, we try to access 5th element in an array of size 3. This produces undefined behaviour:

```
    const auto index = 5;
    //arr[index] is undefined behaviour
    cout << "Integer at index " << index << ": " << arr[index] << endl
    return 0;
}
```

While similar code in Rust is a panic:

```
fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    let index = 5;
    //arr[index] panics with the following message:
    //index out of bounds: the len is 3 but the index is 5
    println!("Integer at index {}: {}", index, arr[index]);
}
```

You might wonder, how is the Rust version better than the C++ version? Well, because the C++ version exhibits undefined behaviour, it gives a no holds barred license to the compiler to do anything in the name of optimizations. In the worst case, this can leak information to an attacker.

The Rust version, in contrast, will always panic. Moreover, because the process terminates due to the panic, a programmer is more likely to notice and fix this bug. In contrast, C++ sweeps the problem under the rug and the process could carry on as if nothing had happened. I will take a Rust panic any day over a C/C++ undefined behaviour.

Vectors

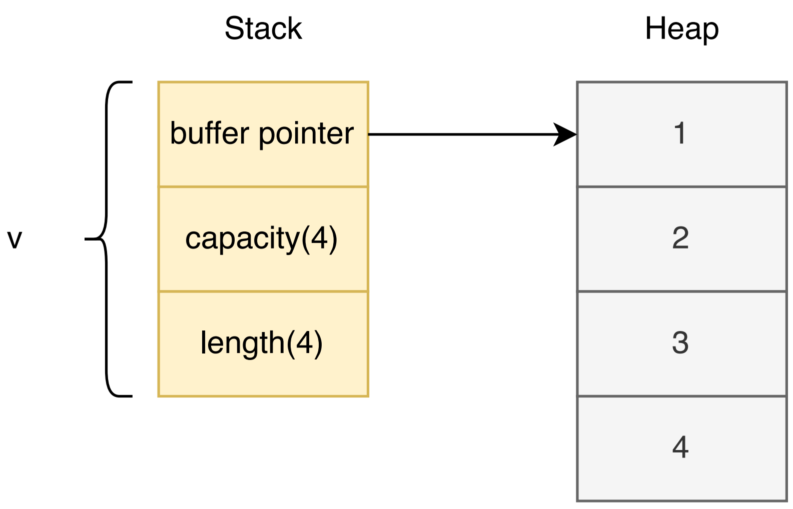
The big limitation of arrays is that they are fixed in size. In contrast, vectors can grow at runtime:

```
fn main() {
    //There are three elements in the vector initially
    let mut v: Vec<i32> = vec![1, 2, 3];
    //prints 3
    println!("v has {} elements", v.len());
    //but you can add more at runtime
    v.push(4);
    v.push(5);
    //prints 5
    println!("v has {} elements", v.len());
}
```

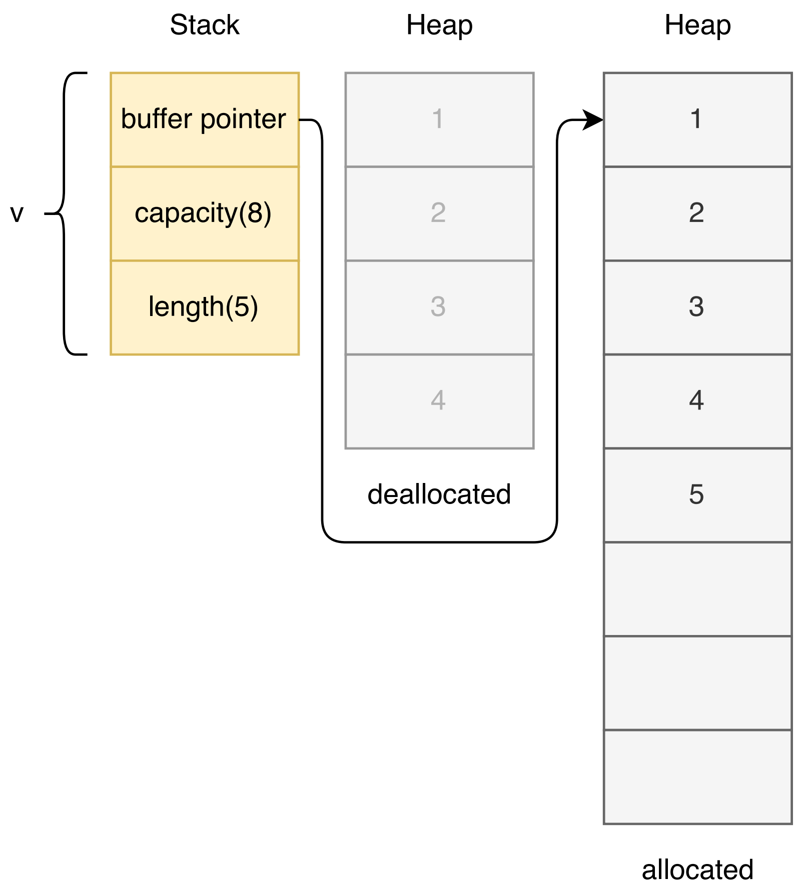
How does a vector allow dynamic growth? Internally, a vector keeps all the elements in an array allocated on the heap. When a new element is pushed, the vector checks if there is still some capacity left in the array. If not, the vector allocates a bigger array, copies all the elements to the new array and deallocates the old array. This can be seen in the following code:

```
fn main() {
    let mut v: Vec<i32> = vec![1, 2, 3, 4];
    //prints 4
    println!("v's capacity is {}", v.capacity());
    println!("Address of v's first element: {:p}", &v[0]);//{:p} print
    v.push(5);
    //prints 8
    println!("v's capacity is {}", v.capacity());
    println!("Address of v's first element: {:p}", &v[0]);
}
```

Initially the capacity of the `v` 's backing array is 4:



A new element is then pushed onto the vector. This makes the vector copy all the elements to a new backing array of capacity 8:



The program also prints the address of the first element of the array before and after pushing a new element onto the vector. Both these printed addresses will be different from each other. This change in addresses is clear evidence of a new array of size 8 being allocated behind the scenes.

NOTE

If you do not see a different address after pushing more elements onto a vector, it might be because the allocator had enough space at the end of the original buffer such that the new and the old buffers have the same starting address. Try pushing more elements and you will see a different address. Read about C library function `realloc` to understand how this might happen.

Slices

Slices act like temporary views into an array or a vector. For example if you have an array:

```
let arr: [i32; 4] = [10, 20, 30, 40];
```

You can create a slice containing second and third elements like this:

```
let s = &arr[1..3];
```

The `[1..3]` syntax creates a range from index 1 (inclusive) to 3 (exclusive). If you omit the first number in the range (`[..3]`) it defaults to zero and if you omit the last number (`[1..]`) it defaults to the length of the array. If you print the elements in the `[1..3]` slice, you get 20 and 30:

```
//prints 20
println!("First element in slice: {:}", s[0]);
//prints 30
println!("Second element in slice: {:}", s[1]);
```

But if you try to access an element outside the range of the slice, it will panic:

```
//panics: index out of bounds
```

```rust
println!("Third element in slice: {:}", s[2]);
```

But how does the slice know that it has only two elements? That's because a slice is not simply a pointer to the array, it also carries around the number of elements of the slice in an additional length field.

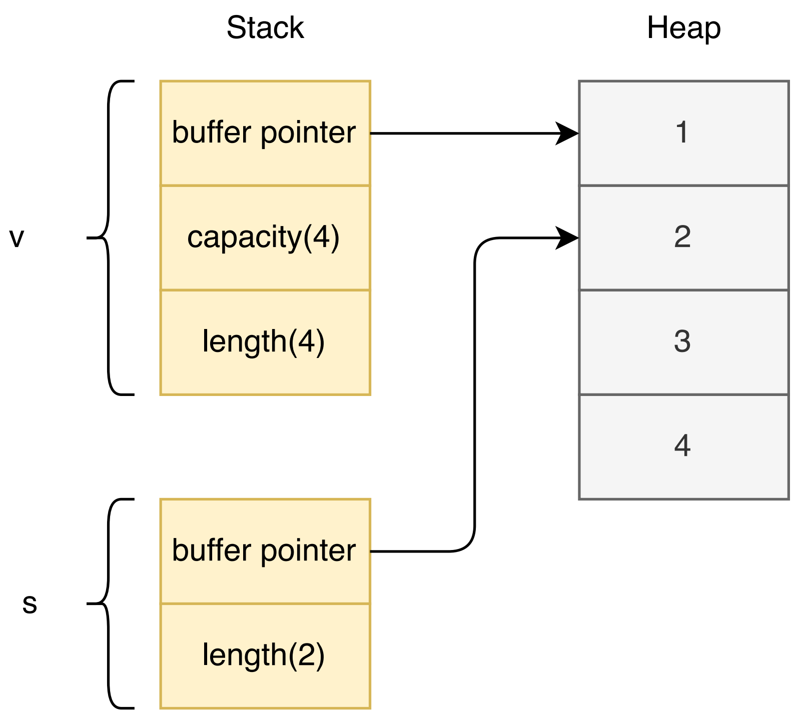
NOTE

A pointer with some additional data besides just the address of the pointed to object is called a fat pointer. Slices are not the only kind of fat pointer in Rust. Trait objects, for example, carry a vtable pointer in addition to the pointer to an object.

For example, if you create a slice to a vector:

```rust
let v: Vec<i32> = vec![1, 2, 3, 4];
let s = &v[1..3];
```

Then in addition to a pointer to the second element in `v`'s buffer, `s` also has an 8 byte length field with value 2:



The presence of the length field can also be seen in the following code in which the size of a slice(`&[i32]`) is 16 bytes (8 for the buffer pointer and 8 for the length field):

```
use std::mem::size_of;

fn main() {
    //prints 8
    println!("Size of a reference to an i32: {:}", size_of::<&i32>());
    //print 16
    println!("Size of a slice: {:}", size_of::<&[i32]>());
}
```

Slices of arrays are similar, but instead of the buffer pointer pointing to a buffer on the heap, it points to the array on the stack.

Since slices borrow from the underlying data structure, all the usual borrowing rules apply. For example, this code is rejected by the compiler:

```
fn main() {
    let mut v: Vec<i32> = vec![1, 2, 3, 4];
    let s = &v[..];
    v.push(5);
    println!("First element in slice: {:}", s[0]);
}
```

Why? Because when the slice is created, it points to the first element of the vector's backing buffer and as a new element is pushed onto the vector, it allocates a new buffer and the old buffer is deallocated. This leaves the slice pointing to an invalid memory address, which if accessed would have lead to undefined behaviour. Rust has saved you from disaster again.

NOTE

Since slices can be created from both arrays and vectors, they are a very powerful abstraction. Hence for arguments in functions, the default choice should be to accept a slice instead of an array or a vector. In fact many functions like `len` , `is_empty` etc. work on slices instead of on vectors or arrays.

Conclusion

Arrays and vectors being one of the first few data structures that new programmers learn, it is no surprise that Rust too has a solid support for them. But as we saw, Rust's safety guarantees do not allow programmers to abuse these fundamental data types. Slices are a novel concept in Rust but since they are such a useful abstraction, you will find them used pervasively in any Rust codebase.