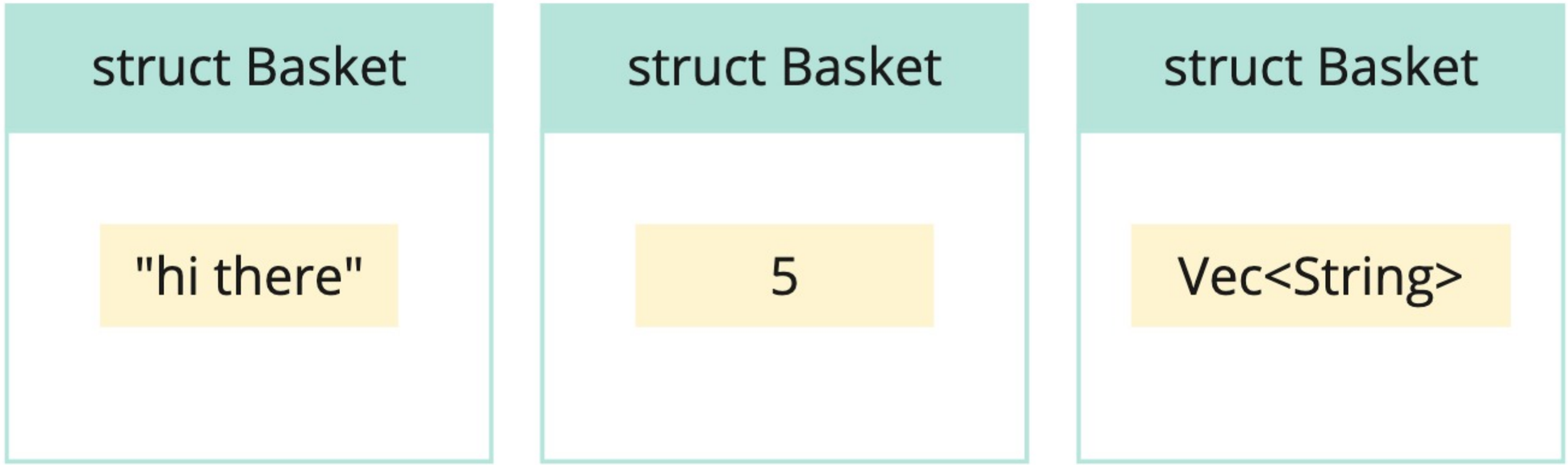
**Goal**

Make a 'Basket' struct that can hold any kind of data

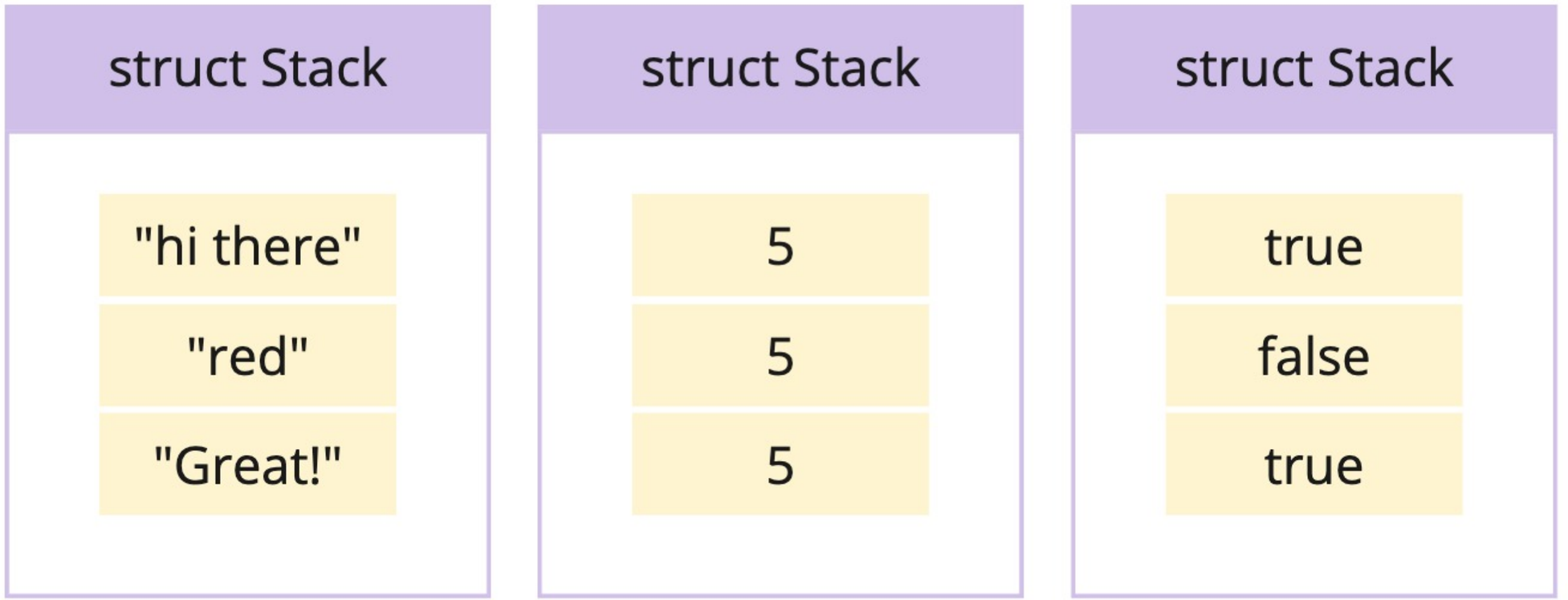


# struct Basket

| Name | Args | Returns | Description |
|---|---|---|---|
| get | - | Option<__> | Returns the value contained by the basket wrapped in an Option. (None if the basket had nothing) |
| put | Value to store | - | Stores a value, replacing whatever the basket stores. **If the basket is storing a number, add the new value to the existing** |
| is_empty | - | bool | True is the basket is empty |

**Secondary Goal**

Make a 'Stack' struct that holds as much data as needed



# struct Stack

| Name | Args | Returns | Description |
| --- | --- | --- | --- |
| get | - | Option<__> | Returns the value most recently added to the Stack, None if the stack is empty |
| put | Value to store | - | Stores a value |
| is_empty | - | bool | True is the stack is empty |

## struct Basket

| Name | Args | Returns | Description |
|---|---|---|---|
| get | - | Option<__> | Returns the value contained by the basket wrapped in an Option. (None if the basket had nothing) |
| put | Value to store | - | Stores a value, replacing whatever the basket stores. **If the basket is storing a number, add the new value to the existing** |
| is_empty | - | bool | True is the basket is empty |

## struct Stack

| Name | Args | Returns | Description |
|---|---|---|---|
| get | - | Option<__> | Returns the value most recently added to the Stack, None if the stack is empty |
| put | Value to store | - | Stores a value |
| is_empty | - | bool | True is the stack is empty |

Methods for Basket and Stack work differently, but have the same signature

We can define these methods in a trait, then have each struct implement that trait

Benefit: throughout our app we can work with a Basket or Stack by using trait bounds

T can be anything that implements the 'Container' trait

```
fn add_string_to_container<T: Container>(container: T, s: String) {
    container.put(s);
}

fn main() {
    let stack = Stack::new();

    add_string_to_container(
        stack,
        String::from("hi")
    );
}
```

This is like a function argument list...

...it is referenced here...

...and here

```
impl<T> Container<T> for Basket<T> {
    fn get(&mut self) -> Option<T> {
        self.item.take()
    }

    fn put(&mut self, value: T) {
        self.item = Some(value);
    }

    fn is_empty(&self) -> bool {
        self.item.is_none()
    }
}
```

Think of this as being like a function argument

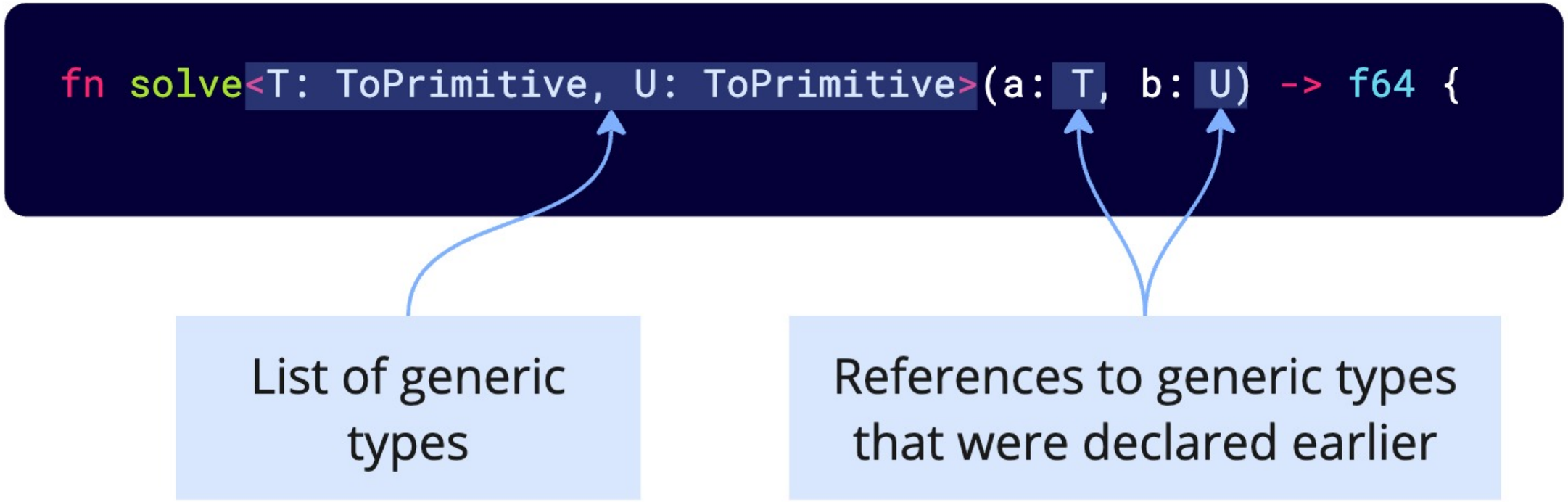
```
pub trait Container<T> {
    fn get(&mut self) -> Option<T>;
    fn put(&mut self, item: T);
    fn is_empty(&self) -> bool;
}
```

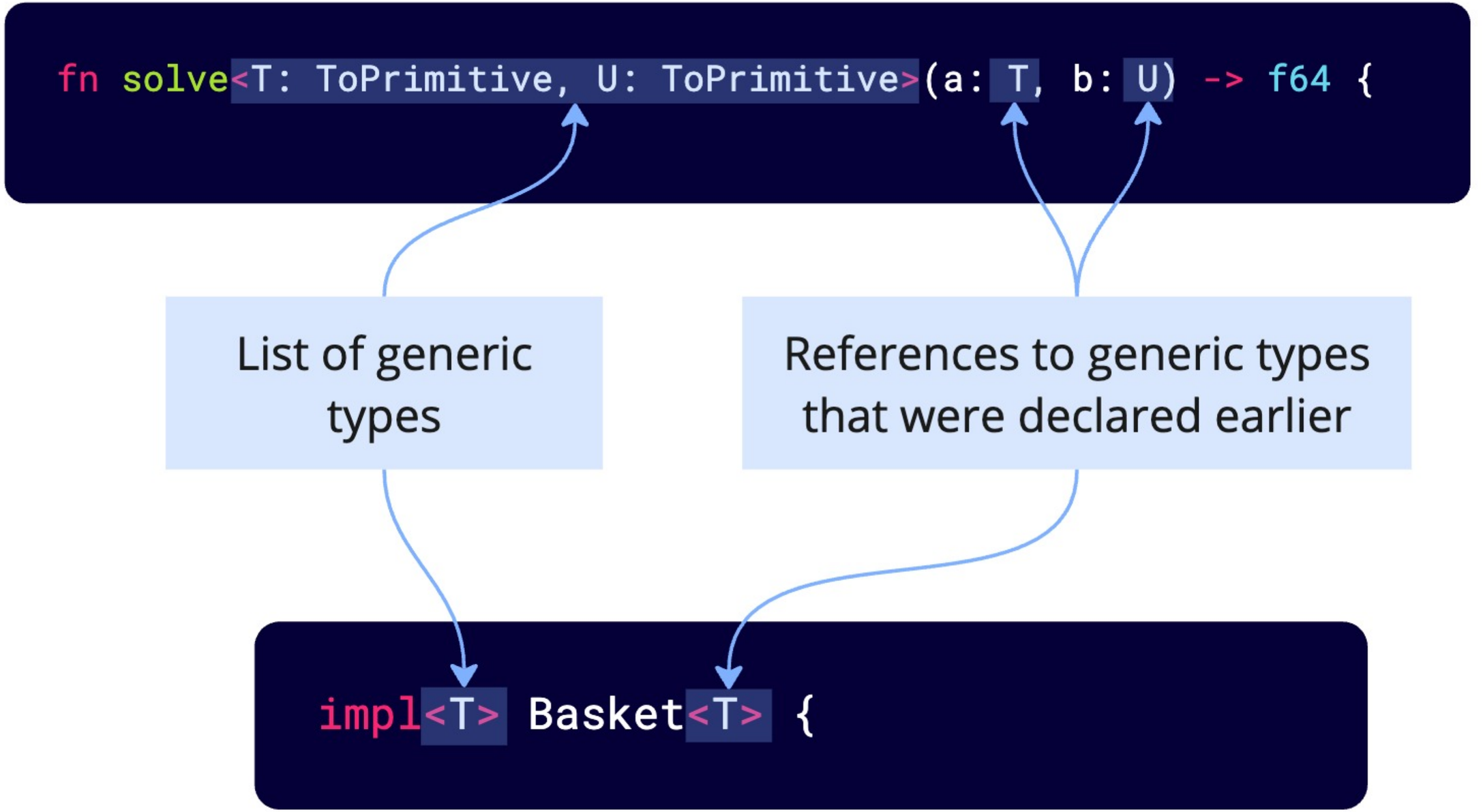
```rust
fn add_string<T: Container<String>>(container: &mut T, item: String) {
    container.put(item);
}
```

```rust
pub struct Basket<T> {
    pub item: Option<T>,
}

impl<T> Basket<T> {
    fn get(&mut self) -> Option<T> {
        self.item.take()
    }

    fn put(&mut self, item: T) {
        self.item = Some(item);
    }
}

fn main() {
    let item = String::from("hi");

    // This version of basket works with strings
    let basket = Basket { item: Some(item) }

    // This version of basket works with i32's
    let basket2 = Basket { item: Some(20) }
}
```

**Basket is a generic struct**

Each instance created can work with one type of data


fn solve<T: ToPrimitive, U: ToPrimitive>(a: T, b: U) -> f64 {
List of generic types
References to generic types that were declared earlier


fn solve<T: ToPrimitive, U: ToPrimitive>(a: T, b: U) -> f64 {
List of generic types
References to generic types that were declared earlier
impl<T> Basket<T> {

# How to Get Help

# How to get the most out of this course

Write code with me

Try the quizzes and exercises

Experiment!

Read error messages