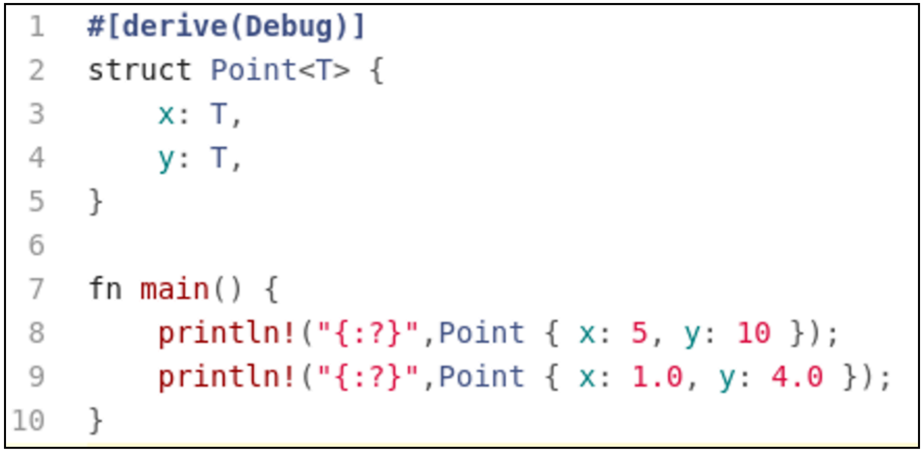
# Generics

- Functions, structs, and enums can be turned into ‘generics’
- By that we mean that generic functions, structs, or enums are *parameterized over some type variables.*
- We often call this **parametric polymorphism** – more on that later.

# Generic Structs

- Generic structs look like ordinary structs with the exception that they introduce a *type variable.*
- In our example the type variable is ‘T’ which allows the type Point to be specified in different coordinate systems.
    - Eg integer coordinates and floating point coordinates.
- Here Rust’s type system insures that both ‘x’ and ‘y’ are instantiated with the same type ‘T’ (whatever that might be)

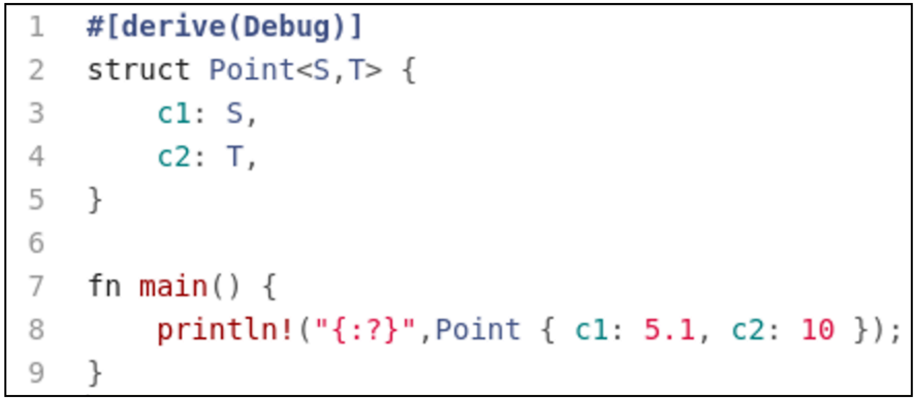


```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    println!("{:?}",Point { x: 5, y: 10 });
    println!("{:?}",Point { x: 1.0, y: 4.0 });
}
```

# Generic Structs

- We can get rid of that constraint by introducing two type variables.

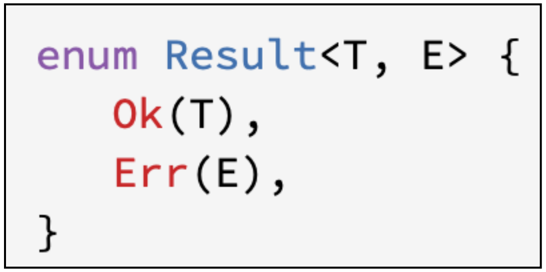


```
#[derive(Debug)]
struct Point<S,T> {
    c1: S,
    c2: T,
}

fn main() {
    println!("{:?}",Point { c1: 5.1, c2: 10 });
}
```

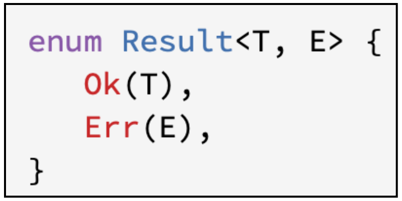
# Generic Enums

- Generic enums look almost identical to generic structs in that the variants of the enum can be parameterized of different data types
- Perhaps the most common generic enum is ‘Result’ which is a type that the standard Rust library uses to return status from a function call

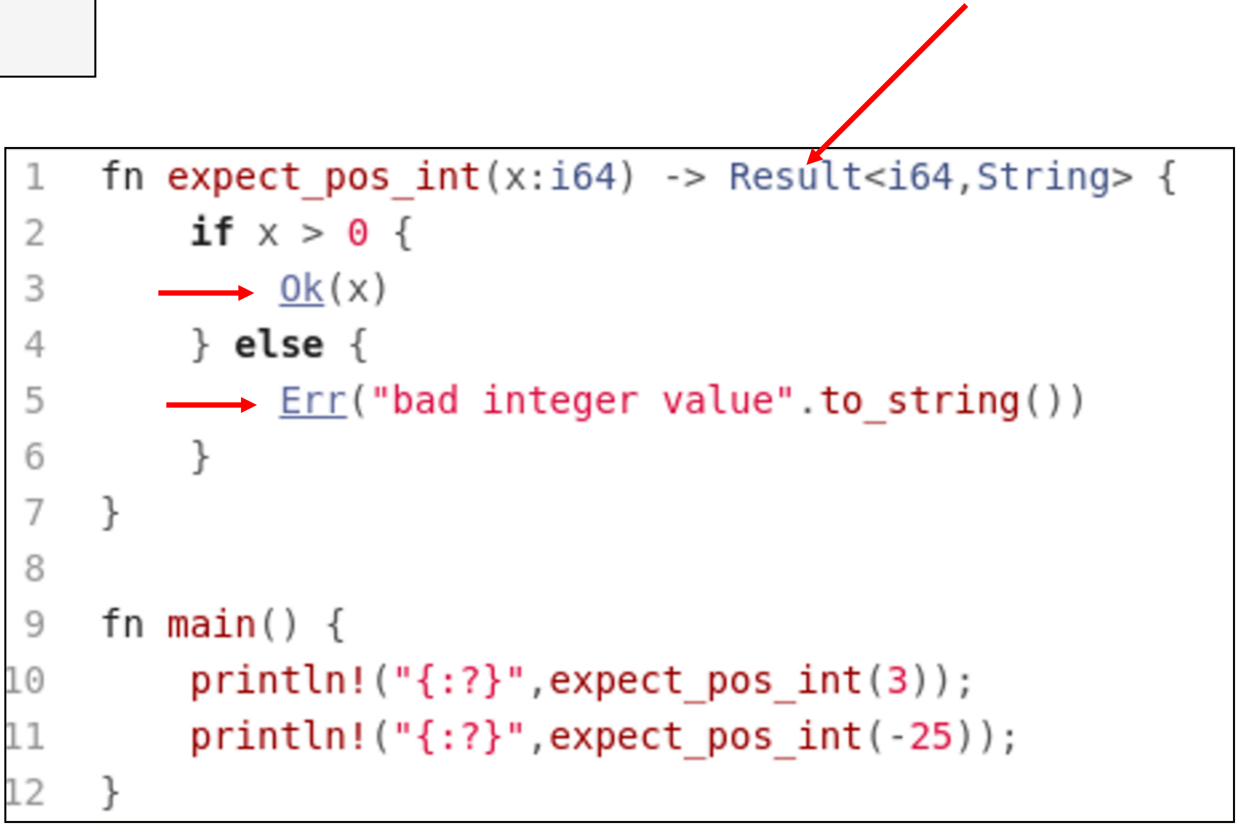


```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

# Generic Enums



```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

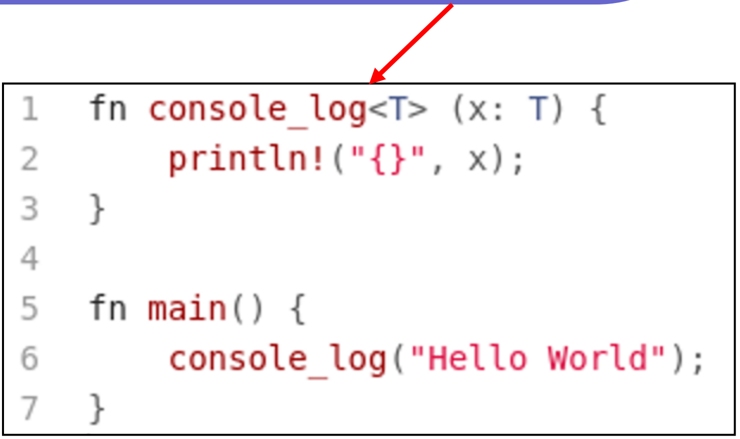


```
fn expect_pos_int(x:i64) -> Result<i64,String> {
    if x > 0 {
        Ok(x)
    } else {
        Err("bad integer value".to_string())
    }
}

fn main() {
    println!("{:?}",expect_pos_int(3));
    println!("{:?}",expect_pos_int(-25));
}
```

# Generic Functions

- As we said at the beginning, functions can also be parameterized over types using type variables.
- However, things are a bit more complicated because given a generic type we want to be able to compute things…but we don't know virtually anything about a generic type.
- Consider the following program, 'console_log' is a generic function parameterized using 'T', yet..

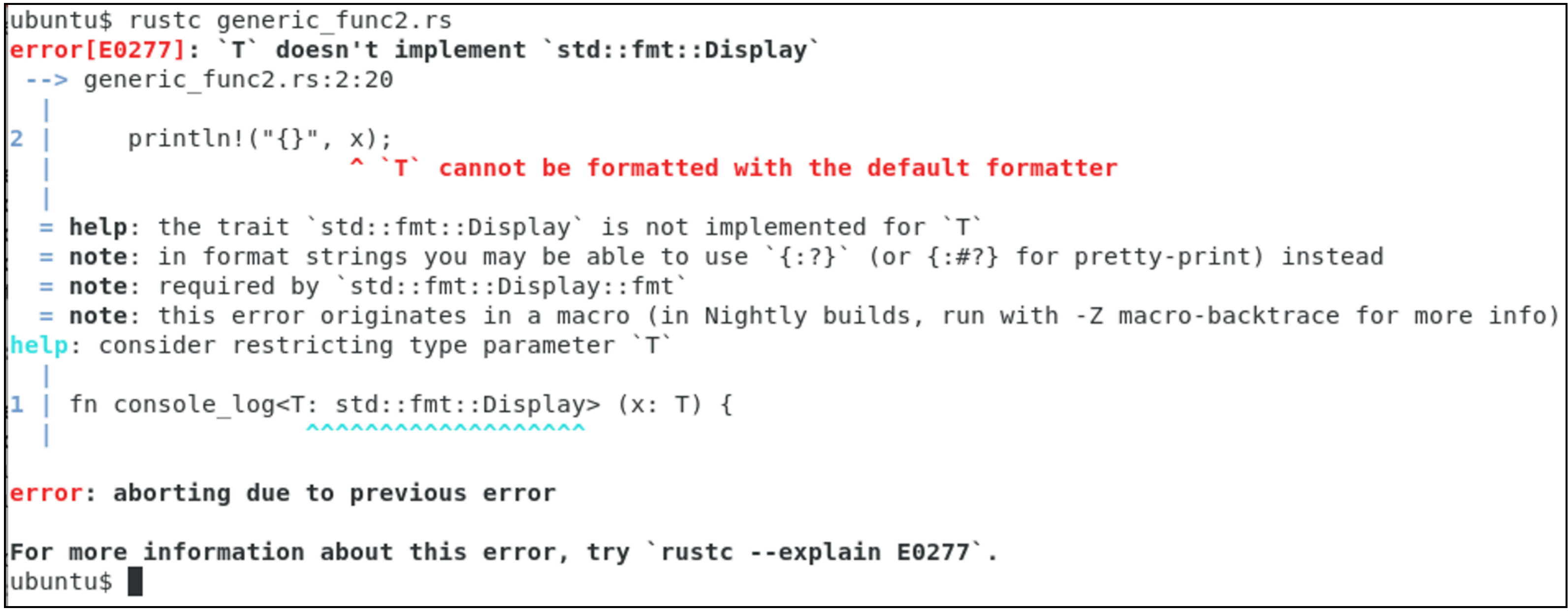


```
fn console_log<T> (x: T) {
    println!("{}", x);
}

fn main() {
    console_log("Hello World");
}
```

# Generic Functions

- Rust does not know how to print generic type ‘T’….



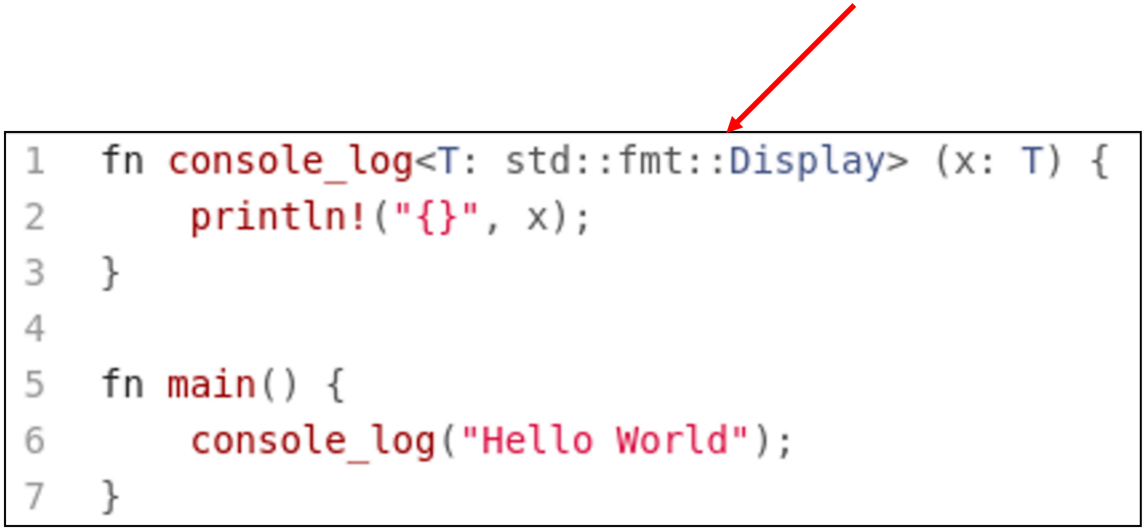
```
ubuntu$ rustc generic_func2.rs
error[E0277]: `T` doesn't implement `std::fmt::Display`
 --> generic_func2.rs:2:20
  |
2 |     println!("{}", x);
  |                    ^ `T` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `T`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
  = note: required by `std::fmt::Display::fmt`
  = note: this error originates in a macro (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider restricting type parameter `T`
  |
1 | fn console_log<T: std::fmt::Display> (x: T) {
  |                ^^^^^^^^^^^^^^^^^^^^
  |

error: aborting due to previous error

For more information about this error, try `rustc --explain E0277`.
ubuntu$ █
```

# Generic Functions

- The solution: tell the compiler that we expect type ‘T’ to have certain **Traits**!
- Introduce **Trait Bounds.**



```
fn console_log<T: std::fmt::Display> (x: T) {
    println!("{}", x);
}

fn main() {
    console_log("Hello World");
}
```

# Generic Functions

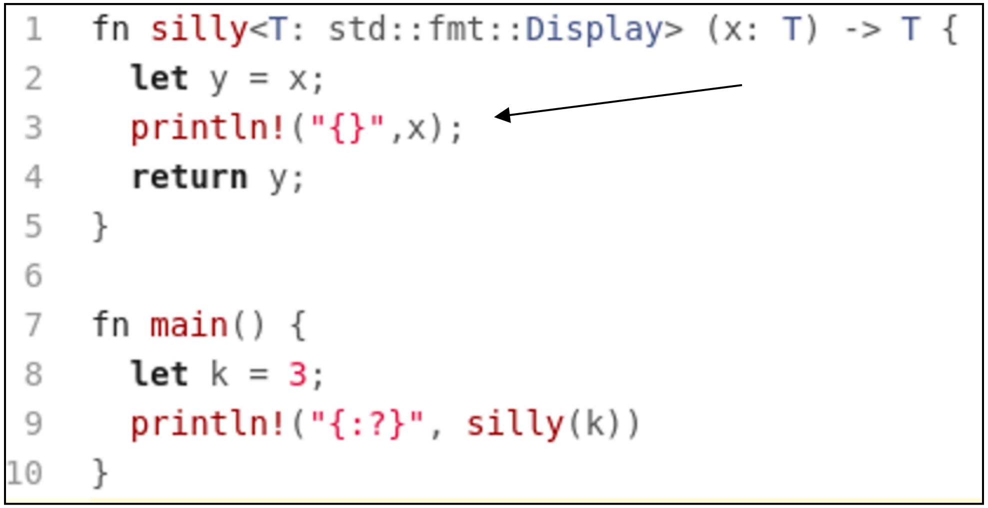
- Being able to bound the types to a generic is sometimes called:
  - *Bounded Parametric Polymorphism*

# Rust Values and Moving

- Rust keeps track how memory is used
- In particular, it keeps track how values move around your program
- It flags situations where you might be using a ‘stale’ value after you copied the value somewhere else.
- Consider…

# Rust Values and Moving

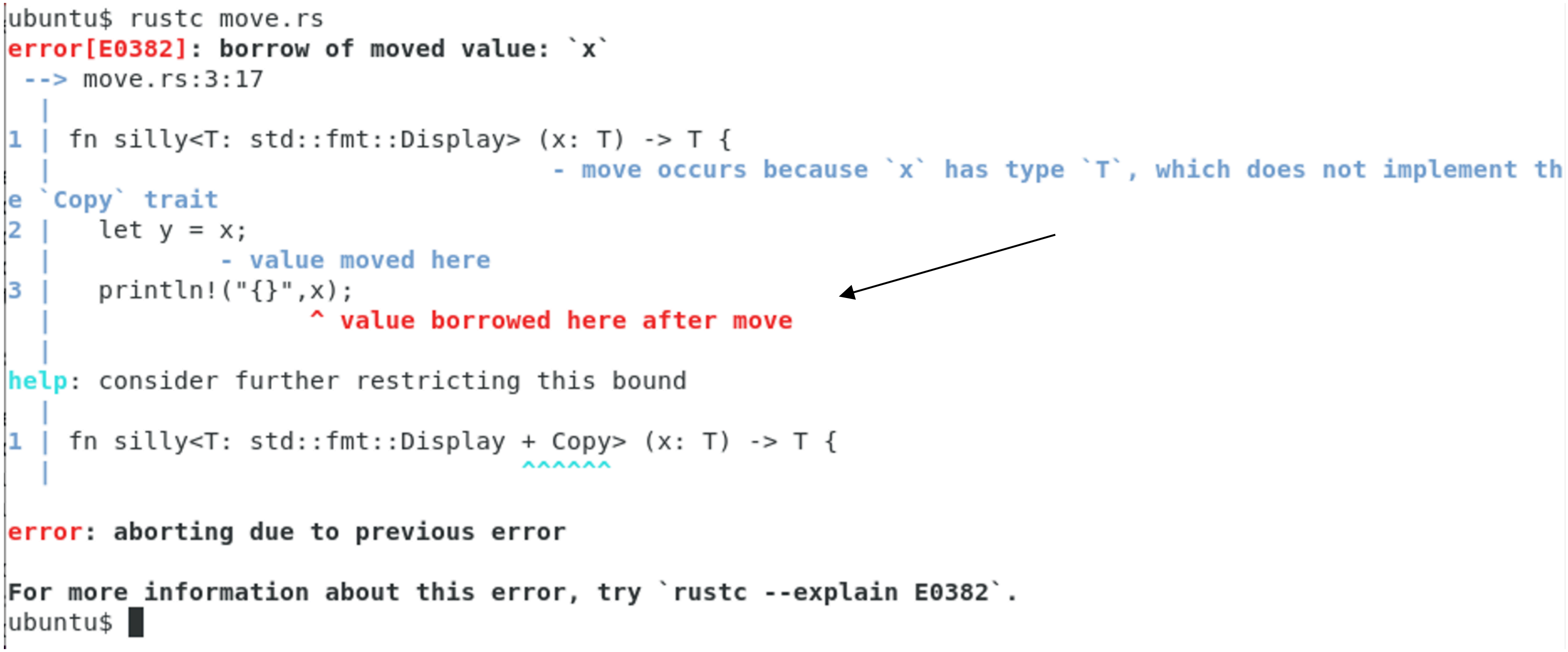
- This program will generate an error because you are trying to use ‘x’ after you copied the value out of ’x’.



```
fn silly<T: std::fmt::Display> (x: T) -> T {
  let y = x;
  println!("{}",x);
  return y;
}

fn main() {
  let k = 3;
  println!("{:?}", silly(k))
}
```

# Rust Values and Moving



```
ubuntu$ rustc move.rs
error[E0382]: borrow of moved value: `x`
 --> move.rs:3:17
  |
1 | fn silly<T: std::fmt::Display> (x: T) -> T {
  |                                 - move occurs because `x` has type `T`, which does not implement th
e `Copy` trait
2 |   let y = x;
  |           - value moved here
3 |   println!("{}",x);
  |                 ^ value borrowed here after move
  |
help: consider further restricting this bound
  |
1 | fn silly<T: std::fmt::Display + Copy> (x: T) -> T {
  |                               ^^^^^^
  |

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
ubuntu$ 
```

# Assignments

- Assignment #4 – See BrightSpace