Only the type signature has changed:

```
fn say_hello(out: &mut dyn Write)     // plain function

fn say_hello<W: Write>(out: &mut W)   // generic function
```

The phrase `<W: Write>` is what makes the function generic. This is a *type parameter*. It means that throughout the body of this function, `W` stands for some type that implements the `Write` trait. Type parameters are usually single uppercase letters, by convention.

Which type `W` stands for depends on how the generic function is used:

```
say_hello(&mut local_file)?;  // calls say_hello::<File>
say_hello(&mut bytes)?;       // calls say_hello::<Vec<u8>>
```

When you pass `&mut local_file` to the generic `say_hello()` function, you're calling `say_hello::<File>()`. Rust generates machine code for this function that calls `File::write_all()` and `File::flush()`. When you pass `&mut bytes`, you're calling `say_hello::<Vec<u8>>()`. Rust generates separate machine code for this version of the function, calling the corresponding `Vec<u8>` methods. In both cases, Rust infers the type `W` from the type of the argument. This process is known as *monomorphization*, and the compiler handles it all automatically.

You can always spell out the type parameters:

```
say_hello::<File>(&mut local_file)?;
```

This is seldom necessary, because Rust can usually deduce the type parameters by looking at the arguments. Here, the `say_hello` generic function expects a `&mut W` argument, and we're passing it a `&mut File`, so Rust infers that `W = File`.

If the generic function you're calling doesn't have any arguments that provide useful clues, you may have to spell it out:

```
// calling a generic method collect<C>() that takes no arguments
let v1 = (0 .. 1000).collect();  // error: can't infer type
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

Sometimes we need multiple abilities from a type parameter. For example, if we want to print out the top ten most common values in a vector, we'll need for those values to be printable:

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

But this isn't good enough. How are we planning to determine which values are the most common? The usual way is to use the values as keys in a hash table. That means the values need to support the `Hash` and `Eq` operations. The bounds on `T` must include these as well as `Debug`. The syntax for this uses the `+` sign: