Functions and expressions

In PHP

A function names its parameters and, when you want it, its return type. `return` sends the value back. A function declared `: void` still produces `null` if you use the call as a value. `return null;` inside that function is a compile error. Leave early with a bare `return;`.

area.php

```
 <?php

function area(int $width, int $height): int
{
    return $width * $height;
}

function greet(string $name): void
{
    echo "Hello, $name\n";
}

echo area(3, 4), "\n";
greet("Ada");

$result = greet("Ada");
var_dump($result);
```

`var_dump` prints `NULL`. The call's value is `null`.

Four PHP habits, and the Rust stand-in for each:

- Default parameter values. Rust uses `Option<T>`, or a second function.
- Named arguments. Rust uses a struct or a builder. Structs are chapter 4.
- Overloading. PHP does not allow two functions with one name either. Rust uses different names, or generics later.
- Variadics (`...$args`). Rust uses a slice or a `Vec`.

A function declared inside another PHP function is registered in the global namespace the first time the outer function runs. Rust's nested `fn` stays inside the outer function, and it cannot see that function's local variables. Closures, which can, come later.

In Rust

`fn` names are `snake_case`. Every parameter has a type. A function that returns a value says so with `->`. The last expression, with no semicolon, is the return value. A semicolon turns that line into a statement, the function's value becomes `()`, and the compiler reports `expected i32, found ()`.

`return` leaves early. A block is an expression too: `let y = { ... };` uses the block's last line as the value of `y`.

A nested `fn` is an item. It does not capture local variables. Pass in what it needs.

src/main.rs

```
 fn area(width: i32, height: i32) -> i32 {
    width * height
}

fn greet(name: &str) {
    println!("Hello, {name}");
}

fn double_or_zero(n: i32) -> i32 {
    if n < 0 {
        return 0;
    }
    n * 2
}

fn main() {
    let value = area(3, 4);
    println!("{value}");

    greet("Ada");

    let y = {
        let part = 2;
        part + 3
    };
    println!("{y}");

    println!("{}", double_or_zero(5));
    println!("{}", double_or_zero(-1));
```

```
    let count = 2;
    fn announce(count: i32) {
        println!("count is {count}");
    }
    announce(count);
}
```

`area` prints `12`. `y` prints `5`. `double_or_zero(-1)` prints `0`. `greet` returns `()`.

Naming the outer local inside the nested function without passing it is `E0434`. This program must fail:

src/main.rs · must fail

```
fn main() {
    let count = 2;
    fn announce() {
        println!("count is {count}");
    }
    announce();
}
```

```
error[E0434]: can't capture dynamic environment in a fn item
 --> src/main.rs:4:28
  |
4 |         println!("count is {count}");
  |                            ^^^^^^^
  |
  = help: use the `|| { ... }` closure form instead
```

Add a semicolon and `area` no longer returns `i32`. This is the compiler's text:

```
error[E0308]: mismatched types
 --> src/main.rs:1:37
  |
1 | fn area(width: i32, height: i32) -> i32 {
  |    ----                             ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
2 |     width * height;
  |                   - help: remove this semicolon to return this value
```

> **Watch for:** The semicolon. `width * height` is an `i32`. `width * height;` is `()`. The error points at the `-> i32` and says `expected i32, found ()`.

> **Idiomatic Rust:** Leave the tail expression without a semicolon. Use `return` only when you leave before the last line. Pass a nested `fn` what it needs. It cannot see the caller's locals.

Side by side

	PHP	Rust
Send a value back	`return`	The last expression. `return` for an early exit
A trailing semicolon	Still needs `return`	The value becomes `()`
No useful value	`: void`, and the call yields `null`	The type is `()`
Default, named, variadic	Built into the call	`Option` or another function, a struct, a slice or `Vec`
Function inside a function	Often becomes global	Nested `fn` cannot capture locals

Try it

Open the first `src/main.rs` in the Playground. Add a semicolon after `width * height` and read the error. It says `expected i32, found ()`. Remove the semicolon. `area(3, 4)` prints `12` again.

Key takeaways

- Parameters are typed, names are `snake_case`, and `->` is the return type.
- The last expression without `;` is the return value. With `;` the type is `()`, and the error is `expected i32, found ()`.

- `return` leaves early. `{ ... }` is an expression. `()` is a value. A PHP `: void` call yields `null`.
- A nested `fn` cannot capture local variables. Closures come later.

What's next: Control flow