A CLI calculator

In PHP

`fgets(STDIN)` returns a string, or `false` at end of input. A loop that ignores `false` spins forever when the pipe closes. Integer-looking division is not integer division: `7 / 2` is `3.5`. `intdiv(7, 2)` is `3`.

calc.php

```
 <?php

$line = fgets(STDIN);
if ($line === false) {
    fwrite(STDERR, "end of input\n");
    exit(0);
}

echo '7 / 2 = ', (7 / 2), "\n";
echo 'intdiv = ', intdiv(7, 2), "\n";
```

A prompt only appears before the read if you flush stdout. `echo $prompt;` `fflush(STDOUT);` is the PHP version of what the Rust program does with `print!` and `flush`.

In Rust

The calculator is one binary. `prompt` prints without a newline, flushes, then reads a line. `read_line` returns `Ok(0)` at end of input. That is EOF. Returning `None` from `prompt` makes `main` stop, instead of asking again forever.

`apply` is a normal function so `cargo test` can call it without stdin. It matches on the operator and uses checked arithmetic, so overflow and divide-by-zero both return `None`. PHP 8's `match` expression is the closest equivalent.

Preview, because the Option chapter has not happened yet: `Option<i32>` is either `Some(number)` or `None`. `let Some(x) = ... else { return; }` unwraps the some-case

and leaves the function on none. `parse` returns a `Result`. The same `let Ok(...) else` shape handles a bad number. Both get a full lesson later. `expect` here is only for "stdout and stdin should work." In a library you would return the error instead.

Integer `/` truncates. `7 / 2` is `3`, not `3.5`.

src/main.rs

```
 use std::io::{self, Write};

/// Preview: `Option` is a value that is either `Some(number)` or `None`.
/// The Option chapter covers it properly. Integers divide by truncation: `7 /
2` is `3`.
fn apply(operation: &str, left: i32, right: i32) -> Option<i32> {
    match operation {
        "+" => left.checked_add(right),
        "-" => left.checked_sub(right),
        "*" => left.checked_mul(right),
        "/" => left.checked_div(right),
        _ => None,
    }
}

fn prompt(label: &str) -> Option<String> {
    print!("{label}");
    io::stdout().flush().expect("stdout should flush");
    let mut line = String::new();
    let bytes = io::stdin()
        .read_line(&mut line)
        .expect("stdin should be readable");
    if bytes == 0 {
        return None;
    }
    Some(line.trim().to_string())
}

fn main() {
    println!("Welcome to the Rust CLI calculator.");
    let Some(raw_left) = prompt("First number: ") else {
        return;
    };
    let Some(raw_right) = prompt("Second number: ") else {
        return;
    };
    let Some(operation) = prompt("Operation (+, -, *, /): ") else {
        return;
    };
    let Ok(left) = raw_left.parse::<i32>() else {
        println!("The first value should be a whole number.");
        return;
    };
    let Ok(right) = raw_right.parse::<i32>() else {
```

```
        println!("The second value should be a whole number.");
        return;
    };
    if let Some(result) = apply(&operation, left, right) {
        println!("Result: {result}");
    } else {
        println!("Use + - * or /, and do not overflow or divide by zero.");
    }
}

#[cfg(test)]
mod tests {
    use super::apply;

    #[test]
    fn adds() {
        assert_eq!(apply("+", 10, 4), Some(14));
    }

    #[test]
    fn integer_division_truncates() {
        assert_eq!(apply("/", 7, 2), Some(3));
    }

    #[test]
    fn rejects_divide_by_zero() {
        assert_eq!(apply("/", 7, 0), None);
    }

    #[test]
    fn overflow_is_none() {
        assert_eq!(apply("+", i32::MAX, 1), None);
    }
}
```

crate root

```
 cargo test
cargo run
```

`cargo test` does not need you to type numbers. `cargo run` does. Press Ctrl-D (Ctrl-Z then Enter on Windows) at a prompt and the program should exit, not spin.

> **Watch for:** `println!` for a prompt. It does flush, because stdout is line-buffered, and it puts the input on the next line. `print!` without a newline is what needs the explicit flush.

> **Idiomatic Rust:** put the arithmetic in a function that does not touch stdin, and test that function. `expect` is fine for "this stream should be open" in `main`. It is not how a library reports a bad number.

Side by side

	PHP	Rust
Read a line	`fgets(STDIN)`	`read_line`
End of input	`false`	`Ok(0)`
Integer division	`intdiv(7, 2)`	`7 / 2` is `3`

Try it

The Playground has no stdin, so **Run** only prints the welcome line and exits. Use the Playground's **Test** action (or `cargo test` locally). You should see four passed tests. Locally you can also `cargo run`, enter `7`, `2`, and `/`. The result is `3`.

Key takeaways

- `read_line` returning 0 bytes is end of input. Stop.
- `print!` plus `flush` shows the prompt before the program waits.
- `7 / 2` is `3` for Rust integers, and `3.5` in PHP. `cargo test` checks `apply`, not `main`.

Your turn

Add a `%` operator to `apply` using `checked_rem`. Write the test first, watch it fail, then make it pass. Name the test `remainder`. The assertion is `apply("%", 7, 3) == Some(1)`.

Solution

src/main.rs · the % arm inside apply

```
 "%" => left.checked_rem(right),
```

Open in Rust Playground

Also extend the operation prompt so it shows `%`: `prompt("Operation (+, -, *, /, %): ")`.

src/main.rs · new test

```
 #[test]
fn remainder() {
    assert_eq!(apply("%", 7, 3), Some(1));
}
```

Open in Rust Playground

What's next: What you learned