DevSheets

Rust

Core syntax, values, and ownership.

01 Start with Cargo

```
cargo new hello
cd hello
cargo run
cargo check
cargo test
cargo fmt
```

Edit src/main.rs: fn main() { println!("Hi"); }. cargo build --release creates an optimized build. Put later snippets inside main unless a complete main is shown; each stands alone.

02 Variables & types

```
let mut count: i32 = 1;
count += 1;
let cost: f64 = 19.95;
let ready: bool = true;
let letter: char = 'R';
println!("{count} {cost} {ready} {letter}");
```

mut permits changes; types can be inferred. i32/u32 are signed/unsigned; usize indexes collections. char is a Unicode scalar value. Another let shadows a binding.

03 Operators & conditions

```
let n = 7;
let half = n as f64 / 2.0;
let odd = n % 2 != 0;
let kind = if odd { "odd" } else { "even" };
if n > 0 && odd { println!("yes"); }
println!("{half} {kind}");
```

Use + - * / %, == != < <= > >=, and && || !. Conditions must be bool. if is an expression: both result branches need compatible types. Integer division truncates toward zero.

04 Loops & ranges

```
for n in 0..3 { println!("{n}"); }
let mut left = 2;
while left > 0 { left -= 1; }
let answer = loop { break 42; };
println!("{left} {answer}");
```

a..b excludes b; a..=b includes it. loop runs until break and can return a value. continue skips to the next iteration. for works with ranges and iterators.

05 Functions & expressions

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
let total = add(2, 3);
println!("{total}");
```

Parameter and return types are explicit. A final expression without a semicolon becomes the return value. return exits early. Omitting a return type means (), the unit type.

06 Tuples, arrays & slices

```
let point = (3, 4);
let (x, y) = point;
let values = [10, 20, 30];
let part = &values[1..3];
println!("{x} {y} {:?}", part);
println!("{:?}", values.get(8));
```

Tuples group different types; point.0 accesses a field. Arrays have a fixed length. A slice borrows a contiguous range. get returns Option; out-of-bounds indexing or slicing panics.

07 Ownership & moves

```
let first = String::from("Maya");
let second = first; // first is moved
let copy = second.clone();
println!("{second} {copy}");
let n = 7; let m = n; // i32 is Copy
println!("{n} {m}");
```

Moving a String transfers ownership; clone duplicates its data. Copy types such as integers remain usable after assignment. An owner leaving scope drops its value.

08 Borrowing & references

```
let mut name = String::from("Maya");
let view = &name;
println!("{view}");
let edit = &mut name;
edit.push_str("!");
println!("{name}");
```

Borrow shared references or one mutable reference, without overlapping uses. Here view is no longer used before edit starts. References cannot outlive their data.

09 Strings & text

```
let mut name = String::from(" Maya ");
name = name.trim().to_owned();
name.push_str("!");
for ch in name.chars() { print!("{ch}"); }
println!(" {}", name.len());
```

String owns UTF-8 text; &str borrows it. len counts bytes; chars yields Unicode scalar values. Integer indexing is unavailable. Slicing text by byte offsets requires UTF-8 boundaries.

Rust

Collections, types, and fallible work.

10 Vectors

```
let mut nums = vec![10, 20];
nums.push(30);
for n in &mut nums { *n += 1; }
println!("{:?}", nums.get(0));
println!("{:?}", nums.pop());
```

Vec<T> grows dynamically. pop returns Option<T>. Iterating over &nums borrows; over nums consumes it. get checks bounds.

11 Structs & methods

```
struct Counter { value: i32 }
impl Counter {
    fn add(&mut self) { self.value += 1; }
}
let mut c = Counter { value: 0 };
c.add();
println!("{}", c.value);
```

impl defines methods. &self borrows, &mut self mutably borrows, and self takes ownership. Structs name their fields.

12 Enums & match

```
enum State { Idle, Ready(i32) }
let state = State::Ready(42);
let value = match state {
    State::Idle => 0,
    State::Ready(n) => n,
};
println!("{value}");
```

Enum variants can carry data. match covers every case. _ is a catch-all pattern.

13 Option & missing values

```
let names = ["Maya", "Leo"];
let name = names.get(8);
match name {
    Some(value) => println!("{value}"),
    None => println!("No name"),
}
```

Option<T> is Some(T) or None. if let handles one pattern. unwrap_or supplies a fallback. unwrap and expect panic on None.

14 Result & propagation

```
use std::{fs, io};
fn main() -> io::Result<()> {
    let text = fs::read_to_string(
        "notes.txt")?;
    println!("{text}");
    Ok(())
}
```

Result<T, E> is Ok(T) or Err(E). ? propagates an error from a compatible function. This complete main needs notes.txt or reports a read failure.

15 Hash maps

```
use std::collections::HashMap;
let mut counts = HashMap::new();
counts.insert("rust", 1);
*counts.entry("rust").or_insert(0) += 1;
println!("{:?}", counts.get("rust"));
counts.remove("rust");
```

get returns Option<&V>. entry updates or inserts. Keys need Eq and Hash. Order is unspecified.

16 Iterators & closures

```
let nums = vec![1, 2, 3];
let out: Vec<_> = nums.iter()
    .map(|&n| n * 2)
    .filter(|n| *n > 2)
    .collect();
println!("{:?}", out);
```

Closures use |arguments| expression and can capture values. Adapters are lazy; collect consumes results. into_iter consumes the collection.

17 Traits & shared behavior

```
trait Named { fn name(&self) -> &str; }
struct User;
impl Named for User {
    fn name(&self) -> &str { "Maya" }
}
println!("{}", User.name());
```

Traits declare behavior. A bound like T: Named requires it. #[derive(Debug, Clone)] generates standard trait implementations.

18 Write files

```
use std::{fs, io};
fn main() -> io::Result<()> {
    fs::write("output.txt", "Hello\n")?;
    println!("Saved output.txt");
    Ok(())
}
```

Complete program: write replaces or creates a file; ? propagates errors. read_to_string needs UTF-8; fs::read reads bytes. File handles close when dropped.