ARITHMETIC IN RUST

- This video covers arithmetic operations in Rust, data types, type conversions, and casting.
- Rust is a statically typed language, requiring careful attention to data types and potential overflows.

BASIC VARIABLES AND DATA TYPES

- Example variables: `let x = 9u8;` and `let y = 10i8;`.
- Comments in Rust: `// This is a comment`.
- `u8` range: 0 to 255.
- `i8` range: -128 to 127.
- Assigning a value outside the range of a type results in a compile-time error.
- A literal is a primitive value like `256`, `1`, or `1.23`.

ADDING VALUES

- Adding variables of different types (e.g., `u8` and `i8`) results in a compile-time error.
- The types must be the same to perform arithmetic operations.
- Example: `i64 + i8` will produce an error.
- Floating-point types also have the same restriction (`f32` and `f64` cannot be directly added).
- Integer values must be explicitly defined as floats (e.g.,

- Integer values must be explicitly defined as floats (e.g., `22.0f32`).

DIVISION, SUBTRACTION, AND OVERFLOWS

- Adding 255u8 + 1u8 results in an overflow error.
- Overflows occur when the result of an operation exceeds the maximum value representable by the data type.
- Rust prevents compilation when overflows are detected.
- Subtraction can also cause overflows if the result is a negative number and the type is unsigned (e.g. `u8`).
- Division between same types will work.

DIVISION DETAILS

- Result of any arithmetic is the same data type as the operands.
- Integer division truncates the decimal portion.
- To obtain a floating-point result, use floating-point types for the operands.

MULTIPLICATION AND MODULUS OPERATOR

- Multiplication uses the asterisk `*`.
- The mod operator `%` returns the remainder after division.

TYPE CONVERSIONS AND CASTING

- Literals can be specified with a specific type suffix, e.g., `255.0f32`.
- Underscores can be used to improve readability: `127_000i64` is equivalent to 127000.
- The `as` keyword performs explicit type conversion (casting).
- Example: `x as i64`.
- It's generally safer to cast a smaller type to a larger type to avoid potential overflows.

STRING CONVERSION FROM USER INPUT

- Reading user input using `std::io`.
- Trimming the input string: removes the newline character using `.trim()`.
- Parsing the string to an integer: `input.trim().parse::<i64>().unwrap()`.
- `.parse()` returns a `Result` type.
- `.unwrap()` extracts the parsed value or causes a panic if parsing fails.
- Errors occur if the input cannot be parsed to the specified integer type.
- Demonstrates adding the converted input value.

INTEGER OVERFLOWS

- Converting a larger type to a smaller type can cause overflows without compiler errors.
- Example: converting `i32::MAX + 1` (as `i64`) to `i32` results in a negative number due to two's complement wrapping.
- Always convert the smaller value to the larger to avoid overflows.

CONVERTING TO FLOAT

- Showing how to convert to float by using `as` keyword
- Example: `10 as f32`