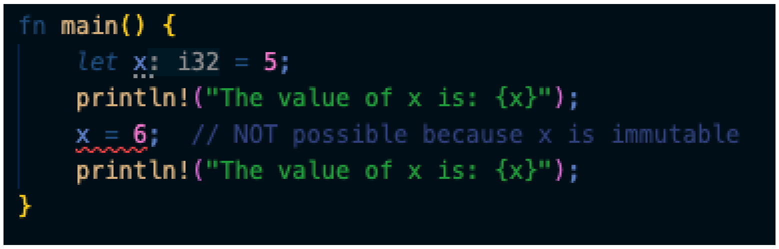
Variables, Mutability and Datatypes

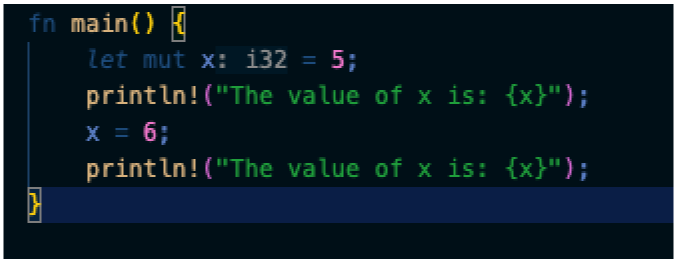
We can define variables in rust using following key word

- const: has ONE memory location and ONE value (cannot be modified on runtime).
- let mut: has ONE memory location and ONE value (can be modified on runtime).
- let: has ONE memory location and ONE value (also cannot be modified on runtime but can be reallocated).

By default, variables are immutable. This is one of many nudges Rust gives you to write your code in a way that takes advantage of the safety and easy concurrency that Rust offers.



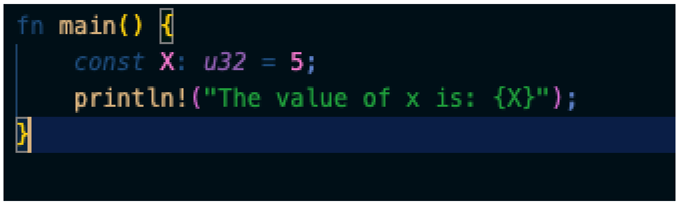
```
fn main() {
    let x: i32 = 5;
    println!("The value of x is: {x}");
    x = 6;  // NOT possible because x is immutable
    println!("The value of x is: {x}");
}
```



```
fn main() {
    let mut x: i32 = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

Constants

- Should always specify the type
- Rust’s naming convention for constants is to use all uppercase with underscores between words

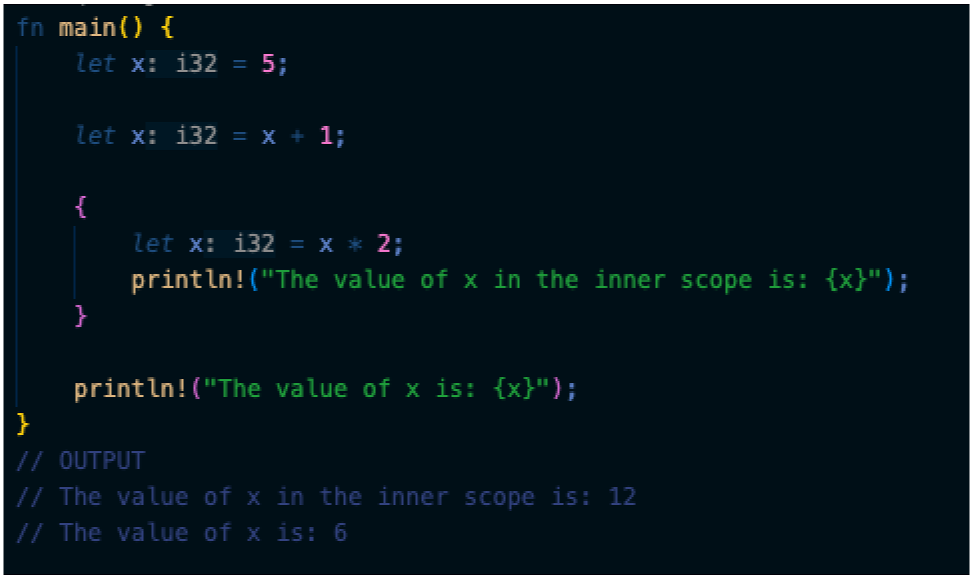


```
fn main() {
    const X: u32 = 5;
    println!("The value of x is: {X}");
}
```

Variables, Mutability and Datatypes

Shadowing

Declare a new variable with the same name as a previous variable (first variable is *shadowed* by the second)



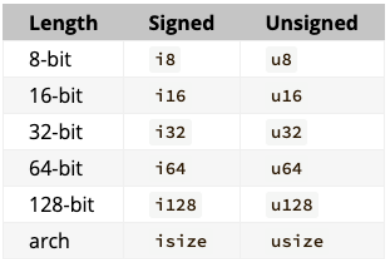
```rust
fn main() {
    let x: i32 = 5;

    let x: i32 = x + 1;

    {
        let x: i32 = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

    println!("The value of x is: {x}");
}
// OUTPUT
// The value of x in the inner scope is: 12
// The value of x is: 6
```

Data Types

Every value in Rust is of a certain data type

- Scalar
- Compound

- **Scalar Types**
    - Integer Types
      a number without a fractional component



Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Variables, Mutability and Datatypes

Signed and unsigned refer to whether it's possible for the number to be negative

i8 can store $-(2^7)\ to\ (2^7 - 1)$ = -128 to 127

U8 can store - $0\ to\ (2^8 - 1)$ = 0 to 255

If you don't know which integer to use , rust default to i32

<u>Integer overflow</u>

Occurs when you try to store the variable to a value outside that range

Can result in one of two behaviors

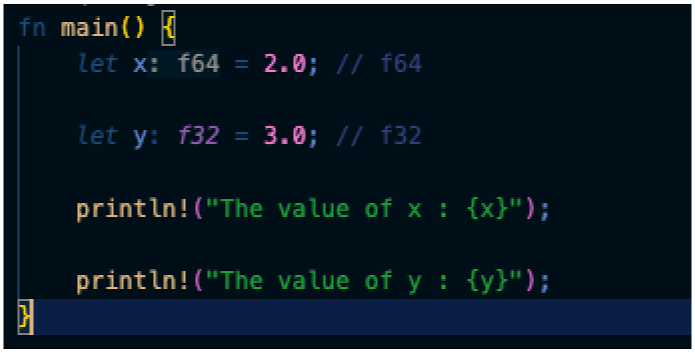
- In debug mode - Rust includes checks for integer overflow that cause your program to panic at runtime if this behavior occurs.
  - *Panicking - is a term in rust when a program exits with an error*
- In release mode - Rust does not include checks for integer overflow that cause panics. Instead, if overflow occurs, Rust performs two's complement wrapping (*values greater than the maximum value the type can hold "wrap around" to the minimum of the values the type can hold*).
  - Eg: In the case of a u8, the value 256 becomes 0, the value 257 becomes 1, and so on

To explicitly handle the possibility of overflow, we can make use of the following methods by standard library.

- Wrap in all modes with the wrapping_* methods, such as *wrapping_add*.
- Return the None value if there is overflow with the checked_* methods.
- Return the value and a boolean indicating whether there was overflow with the overflowing_* methods.
- Saturate at the value's minimum or maximum values with the saturating_* methods.

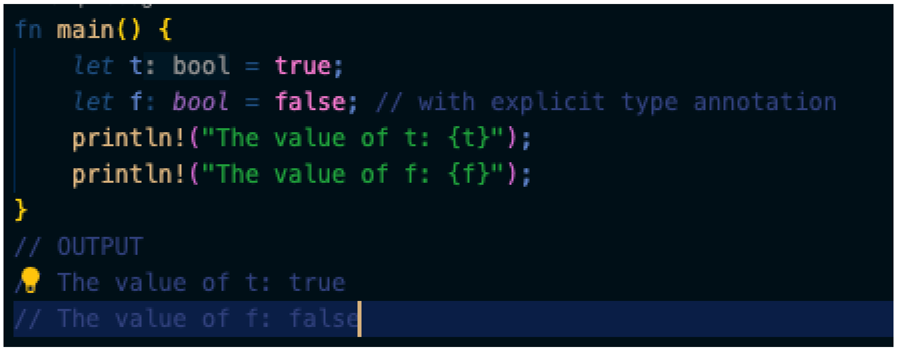
Variables, Mutability and Datatypes

- Floating-point numbers



```rust
fn main() {
    let x: f64 = 2.0; // f64

    let y: f32 = 3.0; // f32

    println!("The value of x : {x}");

    println!("The value of y : {y}");
}
```

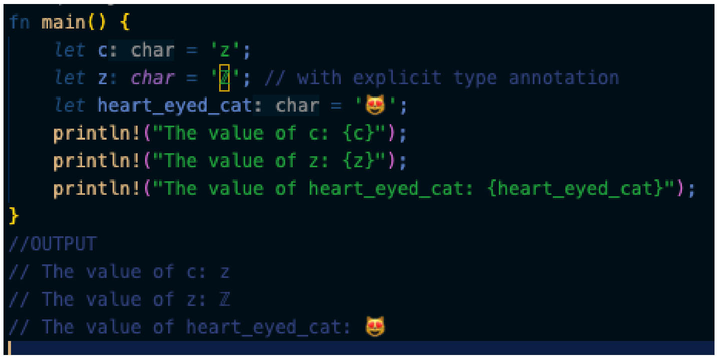
- Booleans



```rust
fn main() {
    let t: bool = true;
    let f: bool = false; // with explicit type annotation
    println!("The value of t: {t}");
    println!("The value of f: {f}");
}
// OUTPUT
// The value of t: true
// The value of f: false
```

- Characters

Char literals in single quotes
String literals in double quotes
Char is 4 bytes in size
Can represent alot more than ASCII (Accented letters; Chinese, Japanese, and Korean characters; emoji; and zero-width spaces)



```rust
fn main() {
    let c: char = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat: char = '😻';
    println!("The value of c: {c}");
    println!("The value of z: {z}");
    println!("The value of heart_eyed_cat: {heart_eyed_cat}");
}
//OUTPUT
// The value of c: z
// The value of z: ℤ
// The value of heart_eyed_cat: 😻
```

Variables, Mutability and Datatypes

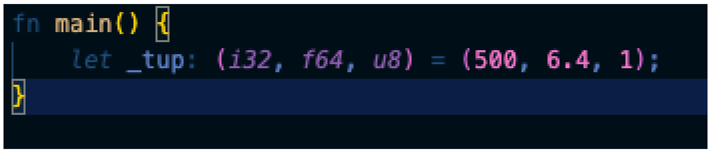
- **Compound Types**
    - Tuples

grouping together a number of values with a variety of types into one compound type.

Tuples have a fixed length: once declared, they cannot grow or shrink in size.

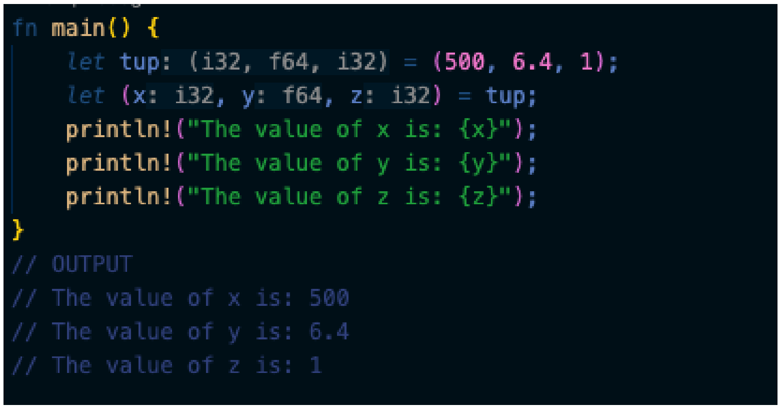
Unit = tuple without any values - empty - ()

declaration



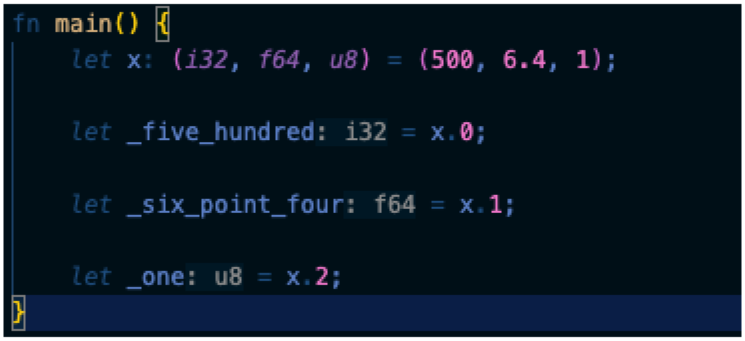
```rust
fn main() {
    let _tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

Destructruing



```rust
fn main() {
    let tup: (i32, f64, i32) = (500, 6.4, 1);
    let (x: i32, y: f64, z: i32) = tup;
    println!("The value of x is: {x}");
    println!("The value of y is: {y}");
    println!("The value of z is: {z}");
}
// OUTPUT
// The value of x is: 500
// The value of y is: 6.4
// The value of z is: 1
```

Accessing element directly

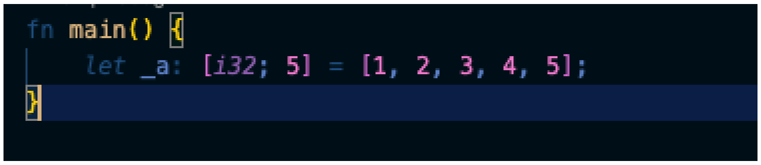


```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let _five_hundred: i32 = x.0;

    let _six_point_four: f64 = x.1;

    let _one: u8 = x.2;
}
```

Variables, Mutability and Datatypes

- <u>Arrays</u>

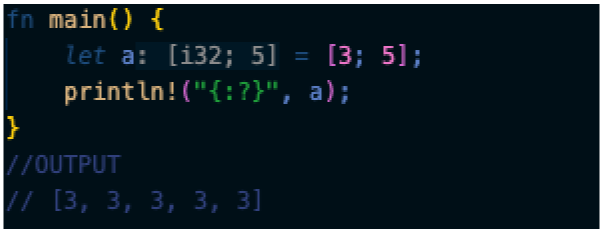
Collection of same type of data
arrays in Rust have a fixed length



```
fn main() {
    let _a: [i32; 5] = [1, 2, 3, 4, 5];
}
```

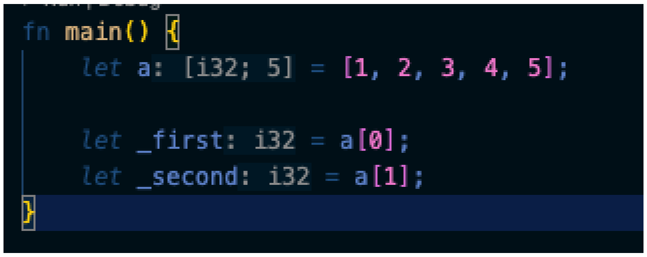
Here, i32 is the type of each element. After the semicolon, the number 5 indicates the array contains five elements.

Create array with



```
fn main() {
    let a: [i32; 5] = [3; 5];
    println!("{:?}", a);
}
//OUTPUT
// [3, 3, 3, 3, 3]
```

Accessing elements in array same value for all elements.



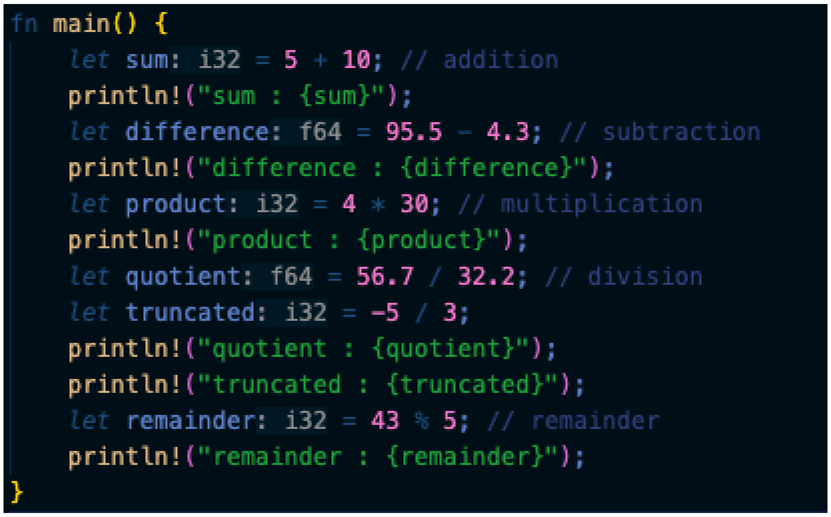
```
fn main() {
    let a: [i32; 5] = [1, 2, 3, 4, 5];

    let _first: i32 = a[0];
    let _second: i32 = a[1];
}
```

When accessing element with invalid index

> the program exited with an error message
> Rust’s memory safety principles
> when you provide an incorrect index, invalid memory can be accessed in many programming languages, but not in Rust.

Variables, Mutability and Datatypes

Numeric Operations



```
fn main() {
    let sum: i32 = 5 + 10; // addition
    println!("sum : {sum}");
    let difference: f64 = 95.5 - 4.3; // subtraction
    println!("difference : {difference}");
    let product: i32 = 4 * 30; // multiplication
    println!("product : {product}");
    let quotient: f64 = 56.7 / 32.2; // division
    let truncated: i32 = -5 / 3;
    println!("quotient : {quotient}");
    println!("truncated : {truncated}");
    let remainder: i32 = 43 % 5; // remainder
    println!("remainder : {remainder}");
}
```