Rust is a programming language developed by Mozilla Research. Rust combines low-level control over performance with high-level convenience and safety guarantees.

It achieves these goals without requiring a garbage collector or runtime, making it possible to use Rust libraries as a "drop-in replacement" for C.

Rust's first release, 0.1, occurred in January 2012, and for 3 years development moved so quickly that until recently the use of stable releases was discouraged and instead the general advice was to use nightly builds.

On May 15th 2015, Rust 1.0 was released with a complete guarantee of backward compatibility. Improvements to compile times and other aspects of the compiler are currently available in the nightly builds. Rust has adopted a train-based release model with regular releases every six weeks. Rust 1.1 beta was made available at the same time of the release of Rust 1.0.

Although Rust is a relatively low-level language, Rust has some functional concepts that are generally found in higher-level languages. This makes Rust not only fast, but also easy and efficient to code in.

```rust
// This is a comment. Single-line look like this...
/* ...and multi-line comment look like this */

///////////////
// 1. Basics //
///////////////

// Functions
// `i32` is the type for 32-bit signed integers
fn add2(x: i32, y: i32) -> i32 {
    // Implicit return (no semicolon)
    x + y
}

// Main function
fn main() {
    // Numbers //

    // Immutable bindings
    let x: i32 = 1;

    // Integer/float suffixes
    let y: i32 = 13i32;
    let f: f64 = 1.3f64;

    // Type inference
    // Most of the time, the Rust compiler can infer what type a variable is, so
    // you don't have to write an explicit type annotation.
    // Throughout this tutorial, types are explicitly annotated in many places,
    // but only for demonstrative purposes. Type inference can handle this for
    // you most of the time.
    let implicit_x = 1;
    let implicit_f = 1.3;

    // Arithmetic
    let sum = x + y + 13;

    // Mutable variable
    let mut mutable = 1;
```

```
mutable = 4;
mutable += 2;

// Strings //

// String literals
let x: &str = "hello world!";

// Printing
println!("{} {}", f, x); // 1.3 hello world

// A `String` - a heap-allocated string
let s: String = "hello world".to_string();

// A string slice - an immutable view into another string
// This is basically an immutable pointer to a string - it doesn't
// actually contain the contents of a string, just a pointer to
// something that does (in this case, `s`)
let s_slice: &str = &s;

println!("{} {}", s, s_slice); // hello world hello world

// Vectors/arrays //

// A fixed-size array
let four_ints: [i32; 4] = [1, 2, 3, 4];

// A dynamic array (vector)
let mut vector: Vec<i32> = vec![1, 2, 3, 4];
vector.push(5);

// A slice - an immutable view into a vector or array
// This is much like a string slice, but for vectors
let slice: &[i32] = &vector;

// Use `{:?}` to print something debug-style
println!("{:?} {:?}", vector, slice); // [1, 2, 3, 4, 5] [1, 2, 3, 4, 5]

// Tuples //

// A tuple is a fixed-size set of values of possibly different types
let x: (i32, &str, f64) = (1, "hello", 3.4);

// Destructuring `let`
let (a, b, c) = x;
println!("{} {} {}", a, b, c); // 1 hello 3.4

// Indexing
println!("{}", x.1); // hello

//////////////
// 2. Types //
//////////////
```

```rust
// Struct
struct Point {
    x: i32,
    y: i32,
}

let origin: Point = Point { x: 0, y: 0 };

// A struct with unnamed fields, called a 'tuple struct'
struct Point2(i32, i32);

let origin2 = Point2(0, 0);

// Basic C-like enum
enum Direction {
    Left,
    Right,
    Up,
    Down,
}

let up = Direction::Up;

// Enum with fields
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}

let two: OptionalI32 = OptionalI32::AnI32(2);
let nothing = OptionalI32::Nothing;

// Generics //

struct Foo<T> { bar: T }

// This is defined in the standard library as `Option`
enum Optional<T> {
    SomeVal(T),
    NoVal,
}

// Methods //

impl<T> Foo<T> {
    // Methods take an explicit `self` parameter
    fn get_bar(self) -> T {
        self.bar
    }
}

let a_foo = Foo { bar: 1 };
println!("{}", a_foo.get_bar()); // 1
```

```rust
// Traits (known as interfaces or typeclasses in other languages) //

trait Frobnicate<T> {
    fn frobnicate(self) -> Option<T>;
}

impl<T> Frobnicate<T> for Foo<T> {
    fn frobnicate(self) -> Option<T> {
        Some(self.bar)
    }
}

let another_foo = Foo { bar: 1 };
println!("{:?}", another_foo.frobnicate()); // Some(1)

/////////////////////////
// 3. Pattern matching //
/////////////////////////

let foo = OptionalI32::AnI32(1);
match foo {
    OptionalI32::AnI32(n) => println!("it's an i32: {}", n),
    OptionalI32::Nothing  => println!("it's nothing!"),
}

// Advanced pattern matching
struct FooBar { x: i32, y: OptionalI32 }
let bar = FooBar { x: 15, y: OptionalI32::AnI32(32) };

match bar {
    FooBar { x: 0, y: OptionalI32::AnI32(0) } =>
        println!("The numbers are zero!"),
    FooBar { x: n, y: OptionalI32::AnI32(m) } if n == m =>
        println!("The numbers are the same"),
    FooBar { x: n, y: OptionalI32::AnI32(m) } =>
        println!("Different numbers: {} {}", n, m),
    FooBar { x: _, y: OptionalI32::Nothing } =>
        println!("The second number is Nothing!"),
}

///////////////////////
// 4. Control flow //
///////////////////////

// `for` loops/iteration
let array = [1, 2, 3];
for i in array.iter() {
    println!("{}", i);
}

// Ranges
for i in 0u32..10 {
    print!("{} ", i);
}
```

```rust
println!("");
// prints `0 1 2 3 4 5 6 7 8 9 `

// `if`
if 1 == 1 {
    println!("Maths is working!");
} else {
    println!("Oh no...");
}

// `if` as expression
let value = if true {
    "good"
} else {
    "bad"
};

// `while` loop
while 1 == 1 {
    println!("The universe is operating normally.");
}

// Infinite loop
loop {
    println!("Hello!");
}

////////////////////////////////
// 5. Memory safety & pointers //
////////////////////////////////

// Owned pointer - only one thing can 'own' this pointer at a time
// This means that when the `Box` leaves its scope, it can be automatically deallocated safely.
let mut mine: Box<i32> = Box::new(3);
*mine = 5; // dereference
// Here, `now_its_mine` takes ownership of `mine`. In other words, `mine` is moved.
let mut now_its_mine = mine;
*now_its_mine += 2;

println!("{}", now_its_mine); // 7
// println!("{}", mine); // this would not compile because `now_its_mine` now owns the pointer

// Reference - an immutable pointer that refers to other data
// When a reference is taken to a value, we say that the value has been 'borrowed'.
// While a value is borrowed immutably, it cannot be mutated or moved.
// A borrow lasts until the end of the scope it was created in.
let mut var = 4;
var = 3;
let ref_var: &i32 = &var;

println!("{}", var); // Unlike `box`, `var` can still be used
println!("{}", *ref_var);
// var = 5; // this would not compile because `var` is borrowed
// *ref_var = 6; // this would not too, because `ref_var` is an immutable reference
```

```
    // Mutable reference
    // While a value is mutably borrowed, it cannot be accessed at all.
    let mut var2 = 4;
    let ref_var2: &mut i32 = &mut var2;
    *ref_var2 += 2;          // '*' is used to point to the mutably borrowed var2

    println!("{}", *ref_var2); // 6 , //var2 would not compile. //ref_var2 is of type &mut i32, so
    // var2 = 2; // this would not compile because `var2` is borrowed
}
```

Further reading

There's a lot more to Rust—this is just the basics of Rust so you can understand the most important things. To learn more about Rust, read The Rust Programming Language and check out the /r/rust subreddit. The folks on the #rust channel on irc.mozilla.org are also always keen to help newcomers.

You can also try out features of Rust with an online compiler at the official Rust playpen or on the main Rust website.