**Summary Of Async Rust:-**

* These are the costs that asynchronous Rust is designed to eliminate
* We can rewrite the function above using Rust's async/.await notation
* Which will allow us to run multiple tasks at once without creating multiple threads
* Overall, asynchronous applications have the potential to be much faster and use fewer resources than a corresponding threaded implementation
* Threads are natively supported by the OS (operating system), and using them doesn't require any special programming model
* Aysnchoronous is running code concurrently or multiple overlapping computations run on a single thread.
* It allow us to runs multiple tasks on same OS thread.
* .await allow us to rum multiple tasks at once without creating multiple threading.
* In Rust, values that are “awaitable” are known as “Futures”.
* Any function can creates thread that uses is usually as easy as calling normal function.
* But, in asynchronous functions requires special support from libraries or languages.
* Async in rust may be different what you are using other languages.
* In rust, “async fn” creates asynchronous function which returns a Future. The returned future must be run to completion.
* Async rust ecosystem has undergone a lot evolution overtime so it is to know what libraries invest, what tools to use and what documentation to read.
* Async/await is still new and its extension like async fn syntax in trait method is still unimplemented.
* Async/.await is a built-in tool for writing asynchronous rust that’s looks like synchronous code.
* “block\_on” blocks the current thread until the provided future has run to completion.
* We can also use .await instead of block\_on inside async fn, .await doesn’t block whole thread but wait for specific future.
* Join! is like .await but can wait for multiple futures concurrently.