

RUST

Bu dökuman çeşitli Udemy kurslarından, çeşitli Rust kitaplarından, çeşitli açık kaynaklı paylaşımlardan kodlar derlenerek hazırlanmıştır. Yorumlar ve ve içerik tamamen bana aittir. Herhangi bir hata veya belirsiz bir durumla karşılaşılırsa bana ulaşabililirsiniz, birlikte düzeltebiliriz.

Muhammet Buğrahan Kara

<https://www.linkedin.com/in/bugrahan-kara-eee/>

Rust Giriş – C++ Benzerlik ve Farklılıklar

- ❖ Rust, Mozilla'nın başlattığı fakat daha sonra başka bir kuruluşun sahiplendiği bir dildir.
- ❖ StackOverflow anket araştırmalarında en sevilen diller arasında olup (2020 anketinde birinci olmuştur), popüler olacak diller arasında olarak gösterilmektedir. Belli bir süre önce Google, Microsoft, Apple gibi şirketler Rust Foundation'ına katıldıklarını da duyurdular.
- ❖ Rust C++'a benzer bir syntax'a sahip olup, C++ gibi garbage collector kullanmadan memory'de daha yüksek güvenlik sunmayı hedefleyen ve bu amaç için ortaya çıkmış bir dildir.
- ❖ Rust ile yazılmış birçok işletim sistemi vardır, Redox, Rux, intermezzOS, Tock gibi.
- ❖ Ayrıca backend işlerinde Go'nun baskınlığını kırmak için adımlar atılmış, örneğin Dropbox'ın sync engine kodları Rust ile yazılmış.
- ❖ Rust'ın 2 modu vardır: "unsafe" ve "safe". Safe modunda "object ownership management" (concurrency, data race'leri düzenler.) gibi birtakım kısıtlamalar bulunur fakat kodun daha güvenli çalışması sağlanır. Unsafe modunda ise programcıya herhangi bir kısıtlama olmaz fakat kodların C++ gibi çakma şansı vardır. Unsafe modunu C++ gibi düşünebiliriz. 2 modunu birlikte kullanabilmek memory crash'lerinin önüne geçmeye yardımcı olur.
- ❖ Rust, data race'leri (thread mutex çakışmaları) compile time'da tespit edebilir, ve atomic objeler (başka operasyonlar tarafından bölünmemesi gereken operasyonlar) için kontrolleri compile time'da sağlar. C'de memory violationlar ve data race'ler kontrol edilmediği için runtime'da problemler ortaya çıkabiliyordu bildiğiniz gibi.
- ❖ C++ ve Rust'ın benzerlikleri için şunları söyleyebiliriz:
 - İkisi de native code'u compile eder.
 - İkisi de "runtime coding" yapmaz.
 - İkisinde de garbage collector yok.
 - İkisinde de belleğe direk erişim mevcut.
 - İkisi de low-level programlama dili.
 - Genel hız performansı açısından da yakın performans sonuçları elde edilmiştir.
 - C++ Zero-Overhead Principle = Rust Zero Cost Abstraction: "Kullanmadığın şey için ödeme yapmazsın.", compiler kodunu büyütecek ve yavaşlatacak bir davranış sergilemez.
- ❖ C++ ve Rust'ın farklılıkları için ise şunları söyleyebiliriz:
 - Hem C++ hem Rust kernel driver geliştirmede kullanılabilse de Rust kernel driver development konusunda yavaşça öne çıktığı söylenebilir.
 - Microservice security platformlarında ise C++'ın Rust'a göre daha çok avantajları olduğu belirtiliyor. Rust'ın ise network monitoring için web uygulamalarında daha kullanışlı olduğu belirtilmiş.
 - C++ ile Rust arasındaki en büyük farkın bellek güvenliği (memory security) açısından geldiği belirtilmektedir. Rust safe-by-default mantığını kullanarak uygulamanın patlamasına neden olan sorunların engellendiği belirtilmektedir.
 - C++ dilin uzun zamanlı olmasından ve olgunluğundan da kaynaklı çeşitli tool'ları bulunmaktadır, örnek olarak farklı debugger çeşitleri ve ticari static analyzer'ları verebiliriz. Bu çeşitliliğin bir güzelliği yanında bir de standartlamamış olması durumu bazen iki farklı derleyici de farklı sonuçlar vermesine sebep olabiliyor. Rust'da ise şu anda sadece "Cargo" isminde bir packet manager'i ve "crates.io" isimli bir package library'si bulunmaktadır. Tek

olduğu için bir standart halini almıştır. Çeşitliliğin ve standartlaşmanın hem avantaj hem dezavantajları bulunmaktadır, fakat derleyiciler arasında farklı sonuçların olması bir güvenilirlik durumu yaratacaktır.

- ❖ “CXX” library’si sayesinde C++ içerisinde Rust kodları çağrılabilirken, Rust içerisinde C++ kodları çağrılabilir. Böylece iki tip kodlar bir arada çalışabilir. CXX aslında kodların type dönüşümünü gerçekleştiriyor. Örnek olarak:
- ❖ ""The FFI signatures are able to use native types from whichever side they please, such as Rust's String or C++'s std::string, Rust's Box or C++'s std::unique_ptr, Rust's Vec or C++'s std::vector, etc in any combination. CXX guarantees an ABI-compatible signature that both sides understand, based on builtin bindings for key standard library types to expose an idiomatic API on those types to the other language. For example when manipulating a C++ string from Rust, its len() method becomes a call of the size() member function defined by C++; when manipulating a Rust string from C++, its size() member function calls Rust's len().""
- ❖ FFI (Foreign Function Interface) sayesinde Rust ile C/C++ arasında bir interface sağlanır. Ayrıca iki dil arasında bir interop (birlikte çalışma) platformu oluşturmak için, Google Chrome’un geliştirdiği bir “autocxx” librarysi de mevcuttur.
- ❖ Temel olarak şu söylenebilir: "Rust’da amaçlanan C++’da runtime’da çıkabilecek error’leri compile time’da göstermektir."
- ❖ Özetle Rust’ın C++’dan en büyük avantajı bellek hatalarında (memory errors) ve eş zamanlı (concurrent) programlamada daha güvenli bir kontrol sunmasıdır. Bellek güvenliğinin çok önemli olduğu ve performansın gerekli olduğu kod kısımlarında kullanılabilmesidir. Örneğin eğer bellekte “reference”i olmayan/kaybolan bir dataya ‘refer’ edecek olan bir kodda, derleyici runtime memory hataları olmadan compile time’da bunu tespit edip engelleyecektir (Borrow Checker), bu nedenle de garbage collector’a ihtiyaç kalmamaktadır. Ayrıca Rust’da reference’lerin bir “lifetime”ı vardır. Böylece uzun süre kullanılmayan referencelar için ayrı bir bellek yönetimi yapılmaktadır.
- ❖ Eğer topluluğu bol, iyi desteklenmiş ve zengin frameworklerine ihtiyacımız varsa C++ tercih edilmesi gerekirken, bellek açısından güvenilir olması gerektiği durumlarda Rust kullanımı tercih edilmelidir.
- ❖ Rust’ın önceden C++ ile yazılmış kodlarının yerini alacağı ve aslında C++’ın deryadeniz bilgi birikiminin yerine geçeceği söylenemez fakat şirketlerin yeni implementasyonlarını C++ yerine daha güvenilir bellek(memory) yönetimi sunan Rust ile yapmaya daha sıcak baktıkları ve buna yöneldikleri söylenebilir.
- ❖ Güncelleme: Android’de ve Linux kernel’inde Rust kullanılmaya başlanmıştır. Ayrıca savunma sanayinde sık sık kullanılan gerçek zamanlı işletim sistemi olan VxWorks’ün yeni versiyonlarında Rust’ı destekleyeceğini belirtmiştir.

Rust Language Compilation Genel Bilgiler

- ❖ “rustc” rust compiler’ıdır. “g++” gibi düşünülebilir. “rustc projectFile.rs” şeklinde terminalden çağrılabilir eğer terminalden compile edeceksek. Bunun sonucunda exe file vs oluşturulur.

- ❖ Convention olarak main file “main.rs” olarak yazılır.
- ❖ “toml” yapılandırma/konfigurasyon dosyaları için kullanılan bir dosya formatıdır. “Tom’s Obvious, Minimal Language”tir açılımı. Açık kaynaklı, adı gibi minimum bilgi ile okunabilir olmayı hedeflemiştir.
- ❖ “rustc” komutu kullanılmadan terminalden “cargo” komutu kullanılarak proje çalıştırılabilir. “cargo build” diyerek projemizi compile edebiliriz. “cargo run” diyerek projeyi run edebiliriz. Bunun için sırayla şu işlemleri yapmalıyız:

- Bu yöntemde önceden bir “.rs” dosyamız bulunmalıdır. Önce bir proje folder’ımız bulunmalıdır, onun içerisinde “src” isimli bir folder bulunmalı ve “.rs” dosyamız bunun içerisinde bulunmalıdır.

- “notepad Cargo.toml” diyerek bir notepad açar, configuration dosyamızın içeriğini şu şekilde yazarız:

```
1 [package]
2 name= "hello_world"
3 version= "0.0.1"
4 authors=["Bugrahan Kara<bugrakara@blabla>"]
```

- Ana proje klasörü içinde “cargo build” diyerek projemizi derleriz aşağıdaki gibi. Bu derleme sonucu “target” isimli bir klasör oluşur, içerisinde “debug” klasörü vardır, bunun içerisinde proje exe’si bulunur.

```
PS C:\Users\bugrakara\Desktop\Rust_Denemeler\HelloWorld> move main.rs .\src\main.rs
PS C:\Users\bugrakara\Desktop\Rust_Denemeler\HelloWorld> notepad Cargo.toml
PS C:\Users\bugrakara\Desktop\Rust_Denemeler\HelloWorld> cargo build
Compiling hello_world v0.0.1 (C:\Users\bugrakara\Desktop\Rust_Denemeler\HelloWorld)
Finished dev [unoptimized + debuginfo] target(s) in 0.61s
```

- “cargo run” diyerek projeyi çalıştırırız, projenin target dosyası silinse bile cargo run diyerek projeyi çalıştırabiliriz; Cargo, proje aşamalarını kendi cachesinde tuttuğu için.

```
PS C:\Users\bugrakara\Desktop\Rust_Denemeler\HelloWorld> cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
Running `target\debug\hello_world.exe`
Hello, world!
PS C:\Users\bugrakara\Desktop\Rust_Denemeler\HelloWorld> █
```

- ❖ “cargo”yu kullanarak sıfırdan(daha önceden bir “.rs” dosyamızın bulunmasına gerek yoktur.) bir proje de yaratılabilir. Bunun için “cargo new proje_ismi proje_flag’ı/tipi” deriz. Direk bize bir yeni bir klasör içerisinde proje oluşturur. Proje Flag’ı dediğimiz şey projenin library mi yoksa executable olacağını gösterir. Bu yöntemde “Cargo.toml” otomatik olarak oluşur. “cargo build” ve “cargo run” diyerek bu projeyi de çalıştırabiliriz.

```
PS C:\Users\bugrakara\Desktop\Rust_Denemeler\helloworld> cargo new helloworld --bin █
```

- ❖ “crate” Rust’ın derleme birimidir. “rustc file.rs” dediğimiz zaman aslında file crate olarak davranır. “rustc” ile cratelerden compile edilerek binary veya library oluşturulur. “package” ise bir veya birden fazla crate içeren şeylere denir.
- ❖ “Cargo.toml” ise cratelerin nasıl build edileceği bilgisini içerir.

Tutorial

Entrypoint

- ❖ C++’daki gibi macro’lara benzer bir kullanımla Rust “attribute”ları sayesinde compiler’ın bazı vereceği warning’ler kapatılabilir ve
- ❖ ya özellikle açılabilir. Warningleri kapatmak için “#![allow(...)]” veya “[allow(...)]” kullanılır. “!” işareti konulunca o attribute bütün crate’e, yani bütün koda uygulanır. “!” konulmazsa kendinden önce yazıldığı modül veya item’a sadece uygulanır. Özellikle bir warningi açmak içinse “[warn(...)]” kullanılır. Örneğin kullanılmayan function’lar için warning almamak için dead_code’a allow verebiliriz.

```
src > main.rs
1  #![allow(dead_code)]
2  #![allow(unused_imports)]
3  #![allow(unused_variables)]
4  #[allow(unused_variables)]
5  //#[warn(unused_variables)]
6
7  fn main() {
8      println!("Hello, world!");
9      let x=0;
10 }
```

Core Data Types

- ❖ “!” işaretini içeren keywordler macro sayılır Rust içinde. “println” de bir macro’dur Rust’ta, çünkü “!” işaretini içerir.
- ❖ Bir variable değerini print içerisinde yazdıracağımız zaman “{ }” işaretini kullanırız, %d yazmak yerine.
- ❖ Rust’ta declare edilen variable’lar default olarak immutable’dır. Yani sabittir. Bu nedenle bir variable’ın değiştirilebilir olmasını istiyorsak açık bir şekilde bunu belirtmemiz gerekiyor. Bunu “mut” keywordü ile belirtiriz.
- ❖ Unsigned için “u”, signed için “i” harfi kullanılır.

```
main.rs
1  #[allow(dead_code)]
2  #[allow(unused_variables)]
3
4  fn main() {
5
6      let a: u8 = 123; //u8=uint8 -> unsigned
7      println!("a= {}",a); //immutable
8
9      let mut b: i8 =0; //i8=int8 -> signed
10     println!("b= {} before",b); //mutable
11     b=42;
12     println!("b= {} after",b);
```

- ❖ “Type inference” özelliği bulunur, bunun için “let” keywordü kullanılır. Modern C++’daki “auto” keywordü gibi çalışır, value’dan variable tipini tahmin eder, belirtmeye ihtiyaç kalmaz.
- ❖ “use std::mem” ile “memory” ile ilgili bir modülü eklemiş olduk. “use” keywordü std’den import etmek ve çağrılan kütüphane fonksiyon isimlerini direk çağırarak isimsel kısaltma için kullanılır.

```
4  use std::mem;
```

```
16 let c = 1234556789; //i32 =32 bits = 4 bytes
17 println!("c= {}, takes up {} bytes", c,mem::size_of_val(&c));
```

c= 1234556789, takes up 4 bytes

- ❖ Bazı variable size'larını işletim sistemimize göre belirlemek isteyebiliriz. Bunun için hazırlanmış 2 keyword vardır. "usize" ve "isize" keywordleri bizim işletim sistemimizin bit uzunluğuna eşittir ve gene unsigned, signed olarak ayrılır.

```
19 let z: isize = 123;
20 let size_of_z = mem::size_of_val(&z);
21 println!("z = {}, takes up {} bytes, {}-bit OS", z,size_of_z,size_of_z*8); //1byte=8bit
```

z = 123, takes up 8 bytes, 64-bit OS

- ❖ Boolean, char, float kullanımları şöyledir:

```
25 let d:char = 'x';
26 println!("{}", is a char, size ={} bytes",d,mem::size_of_val(&d));
27
28 let e: f32 = 2.5; //The default is f64 for floating points
29 println!("{}", size={} bytes",e,mem::size_of_val(&e));
30
31 let g: bool = false;
32 println!("{}", size={} bytes",g,mem::size_of_val(&g));
```

x is a char, size =4 bytes
2.5, size=4 bytes
false, size=1 bytes

Operators

- ❖ Arithmetik işlemlerde kullandığımız 1 arttırıp azaltmaya yarayan "++,"--" kullanımı maalesef Rust'ta bulunmamaktadır. Fakat "+=1, -=1" gibi kullanımlar vardır.
- ❖ Bir sayının karesini küpünü alırken kullandığımız "pow" "i32::pow(a,3)" şeklinde kullanılır. "pow" sanki i32'nin bir methodu gibi kullanılır. Sonuç i32 cinsinden olacağı için tipini böyle veririz.

```
14 let a_cubed = i32::pow(a,3);
15 println!("{}", cubed is {}",a,a_cubed);
```

- ❖ Eğer float için power alıyorsak üstlü sayımızın üs'sü integer ise "powi", float'sa "powf" dememiz gerekiyor.
- ❖ Pi sayısı için "std::f64::consts::PI" şeklinde bir sabit vardır.

```
16 let b=2.5;
17 let b_cubed = f64::powi(b,3);
18 let b_to_pi = f64::powf(b,std::f64::consts::PI);
19 println!("{}", cubed={}, {}^pi={},b,b_cubed,b,b_to_pi);
```

- ❖ Diğer operatörler C++ ile aynıdır.

```
let mut a= 2+3*4;
println!("{}",a);
a=a+1;
a-=2; // a=a-2
```

```
let pi_less_4 = std::f64::consts::PI <4.0;
let x=5;
let x_is_5 = x==5;
```

Scope and Shadowing

- ❖ C++ ile scope kuralları ve durumları aynı şekilde Rust'da da geçerlidir.

```
main.rs
1  use std::mem;
2
3  fn scope_and_shadowing(){
4
5      let a=123;
6      {
7          let b=456;
8          let a=789;
9          println!("inside, b={}",b);
10         println!("inside, a={}",a);
11     }
12     println!("a={}",a);
13     //println!("outside, b ={}",b); //gives an error.
14 }
15
16 fn main(){
17
18     scope_and_shadowing();
19 }
```

Declaring and Using Constants

- ❖ Rust'daki constant için kullanılan "const"lar, C++'daki macro'lar gibi kullanılırlar. Bunların sabit bir adresleri yoktur. Sadece eşitlediğimiz şeyi temsil ederler.
- ❖ "const"lar için type'ını kesinlikle vermemiz gerekiyor, u8 gibi.
- ❖ Eğer memory adresinin olmasını istediğimiz ve tüm program boyunca var olmasını beklediğimiz bir sabit varsa "static" keywordünü kullanıyoruz. Global value yaratır.
- ❖ Eğer biz static variable'ımızı "mut" ile değiştirilebilir yapmak istersek bu aslında biraz unsafe bir durum oluşturacaktır. Farklı threadler aynı anda okuma ve yazmaya çalışabilirler. Bu nedenle Rust bunu safe bulmadığı için kısıtlamış ve "unsafe" Rust modunda çalışmasına müsaade etmiştir. Yani "mut" ile değiştirilebilir bir static yaparsak o variable ile ilgili işlemlerimizi "unsafe" block içinde yapmamız gerekecek.

```

main.rs
1  use std::mem;
2
3  const MEANING_OF_LIFE: u8 = 42; //const: No fixed address. Type of const must be given.
4  static z:i32 = 123; //have fixed address
5  static mut y:i32 = 123; // should be used with unsafe block if static is mutable
6
7
8
9  fn main(){
10
11      println!("z={}",z);
12
13      unsafe{
14          println!("y={}",y);
15      }
16  }

```

Stack and Heap

- ❖ Stack hızlıdır fakat kapasitesi limitlidir. Short-term memory
- ❖ Heap yavaştır. Long-term memory. Bir variable heap'te tutulacağı zaman onun pointer'ı yani onu heap'te gösteren adres, stack içerisinde tutulur.
- ❖ Bir source file'ı diğer source file içine header'ını ekler gibi dahil etmek için, "mod" keywordü kullanılır. "mod sh" diyerek #include "sh.rs" demiş gibi olduk.
- ❖ "mem::size_of_val(&variable_name)" o variable'ın stack'te tutulan memory bytes miktarı verir.
- ❖ Functionlarda return type'ını göstermek için "fn func_name(...) -> return_type" syntaxı kullanılır. Function içinde eğer son satırda return etmesi gereken şey yazıldıysa "return" keywordü kullanılmasına gerek yoktur. Ama if-else gibi bir şey kullanıp function ortalarında return etmemiz gerekirse o zaman "return" keywordü C++'daki gibi kullanılır.
- ❖ Heap'te tutulacak bir variable oluşturmak istersek "Box::new(variable_type)"ı kullanırız. Bu o variable'ın pointer'ını dönecektir.
- ❖ Dışarıdan erişmek istediğimiz function'lara "pub" keywordünü koyarak public yapar, erişime izin veririz.
- ❖ Dereference(*) operatörü kullanarak C++'daki gibi variable'ı "unbox" yapabiliriz, stack'e taşıyabiliriz.

```

main.rs  X  sh.rs
main.rs
1  #![allow(unused_variable)]
2  #![allow(dead_code)]
3
4  mod sh;
5  use std::mem;
6
7  fn main(){
8
9      sh::stack_and_heap();
10 }

```



```

main.rs  sh.rs  x
sh.rs
1  #![allow(dead_code)]
2  use std::mem;
3
4  struct Point
5  {
6      x:f64,
7      y:f64
8  }
9
10 fn origin() -> Point // "->" shows the return type.
11 {
12     Point{x: 0.0, y: 0.0}
13 }
14
15 pub fn stack_and_heap(){
16
17     let p1 = origin();
18     let p2 = Box::new(origin()); //To create dynamic variable, use Box::new(...)
19
20     //mem::size_of_val shows the size in stack
21     println!("p1 takes up {} bytes on the stack",mem::size_of_val(&p1)); //16 bytes due to 2 * f64(64bit=8 bytes)=16 bytes
22     println!("p2 takes up {} bytes on the stack",mem::size_of_val(&p2)); //It stores memory adress of variable,
23                                     // so the result is the same with the size of any address(Our computer is 64 bit=8byte).
24                                     //result=8bytes(for stack). For heap, result will become 16 bytes.
25     let p3= *p2; //To unbox it
26     println!("{}",p3.x);
27 }

```

Debugging Rust Applications

- ❖ Debugging, VSCode üzerinden “CodeLLDB” extension’ı kullanarak yapılabilir.
- ❖ CLion IDE’si de ayrıca kullanışlı bir debugging arayüzü sunar.
- ❖ Normal debugging kullanımının dışında bir fark görülmemiştir.

Control Flows

If-Else-Else If

- ❖ C++’dan farklı olarak if-else if’lerde “()” kullanılmadan yazılıyor. Yani “if temp<15 { ... }” şeklinde kullanılır.
- ❖ “{ }” işaretini kullanmak zorundayız if scope’unu yazarken.
- ❖ C++’daki if-else için “ condition ? result_if_true : result_if_false ” kullanımına benzer bir şekilde Rust içerisindeki kullanım şöyledir:

```

17     let day = if temp>20 {"sunny"} else {"cloudy"};
18     println!("today is {}",day);
19
20     println!("it is {}", if temp>20 {"high"} else if temp<10 {"low"} else {"OK"});

```

```

1  fn if_statement(){
2
3      let temp=15;
4      if temp>30
5      {
6          println!("high");
7      }
8      else if temp<10
9      {
10         println!("low");
11     }
12     else
13     {
14         println!("OK");
15     }
16
17     let day = if temp>20 {"sunny"} else {"cloudy"};
18     println!("today is {}",day);
19
20     println!("it is {}", if temp>20 {"high"} else if temp<10 {"low"} else {"OK"});
21
22     println!("it is {}",
23         if temp>20{ if temp>30 {"very high"} else {"high"} }
24         else if temp<10 {"cold"}
25         else {"OK"});
26 }
27
28 fn main() {
29     if_statement();
30 }

```

While and Loop

- ❖ “while” da da “()” işaretleri kullanılmadan yazılmıştır.
- ❖ “while true” için “loop” keywordü kullanılmış.
- ❖ C++ ile benzer şekilde “break”, “continue” keywordleri kullanılmış.

```

28  fn while_and_loop(){
29
30      let mut x=1;
31      let mut y=1;
32
33      while x<1000
34      {
35          x*=2;
36          if x==64 {continue;}
37          println!("x= {}",x);
38      }
39
40      loop{ // while true
41          y*=2;
42          println!("y= {}",y);
43          if y== 1<<10 {break;}
44      }
45 }

```

For Loops

- ❖ For loop için Python'dakine benzer şekilde "in" keywordü kullanılmış, ve "a..b" şeklinde bir range verilmiş. Örnek olarak "for x in 30..41" şeklinde for loop kurulmuş. Burada x 30 ve 41 arasındaki elemanlara eşittir, modern C++'daki for(auto x:list) olarak düşünülebilir.
- ❖ Eğer for loop'undaki index sayısını ve range'deki objeye aynı anda erişmek istersek bunun için de "for (index,y) in (range).enumerate()" şeklinde bir yapı geliştirilmiştir.

```
47 fn for_loop(){
48
49     for x in 1..11 // from 1 to 10
50     {
51         println!("x= {}",x);
52     }
53     for (pos,y) in (30..41).enumerate()
54     {
55         println!("{}",pos,y);
56     }
57 }
```

x= 1	0 : 30
x= 2	1 : 31
x= 3	2 : 32
x= 4	3 : 33
x= 5	4 : 34
x= 6	5 : 35
x= 7	6 : 36
x= 8	7 : 37
x= 9	8 : 38
x= 10	9 : 39
	10 : 40

Match Statement

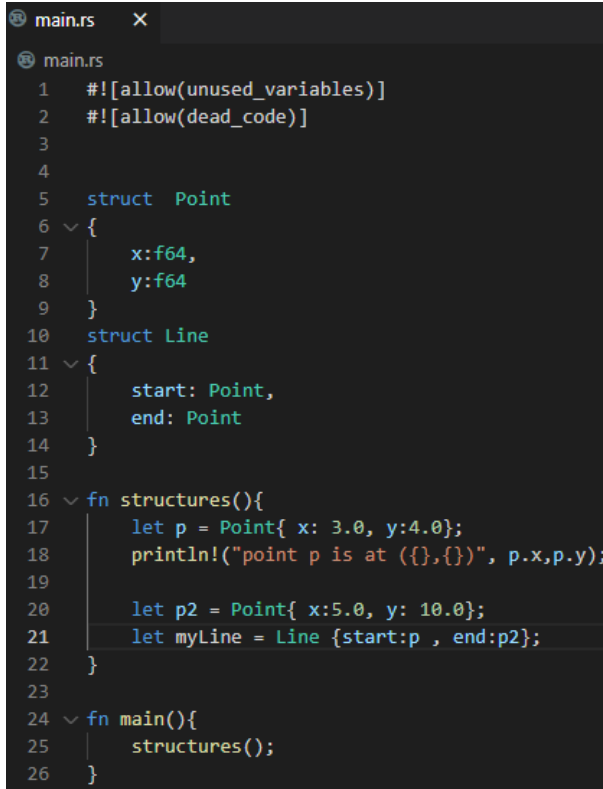
- ❖ "match" C++'daki switch-case yapısına benzer şekilde bir yapıdır. "_" önceki case'lerin olmadığı durumları gösterir, switch-case'deki "default" keywordü gibi.
- ❖ Range verirken "1..=1000" gibi bir örnekteki "=" işareti "dahil" anlamına gelmektedir. "1000"de range'e dahildir.

```
59 fn match_statement(){
60     let country_code = 90;
61
62     let country = match country_code{
63         44 => "UK",
64         90 => "TURKIYE",
65         7 => "Russia",
66         1..=1000 => "unknown", // =inclusive range, 1 to 1000 included
67         _ => "invalid"
68     };
69
70     println!("the country with code {} is {}",country_code,country);
71 }
```

Data Structures

Structs

- ❖ C++'daki struct'lar gibi tanımlanır. Sadece struct objesi initialize edilirken variable'ları için "=" yerine ":" kullanılır.



```
main.rs X
main.rs
1  #![allow(unused_variables)]
2  #![allow(dead_code)]
3
4
5  struct Point
6  {
7      x:f64,
8      y:f64
9  }
10 struct Line
11 {
12     start: Point,
13     end: Point
14 }
15
16 fn structures(){
17     let p = Point{ x: 3.0, y:4.0};
18     println!("point p is at ({},{})", p.x,p.y);
19
20     let p2 = Point{ x:5.0, y: 10.0};
21     let myLine = Line {start:p , end:p2};
22 }
23
24 fn main(){
25     structures();
26 }
```

Enumerations

- ❖ Enum kullanımı da C++'a benzer olmakla birlikte birkaç farklılık vardır.
- ❖ Parametrelili enum oluşturabiliyoruz, tuple olarak. Arguman typelerini "(" içerisinde gösterip o şekilde bir enum objesi oluşturabiliyoruz.
- ❖ Enum içerisinde struct tanımlayabiliyoruz. Ona argümanlarını verirken "{" kullanarak, struct içerisindeki variable isimleri ile ":" kullanıp eşleyerek tanımlıyoruz (Örnekten daha net anlaşılacak). Bu şekilde kullanılması gerekiyor, aksi halde compile hatası veriyor.

```

28 enum Color{
29     Red,
30     Green,
31     Blue,
32     RgbColor(u8,u8,u8), //tuple
33     CmykColor{cyan:u8, magenta:u8, yellow:u8, black:u8}, //struct
34 }
35
36 fn enums(){
37     //let c:Color = Color::Red;
38     //let c:Color = Color::RgbColor(10,0,0);
39     let c:Color = Color::CmykColor{cyan:0, magenta:128, yellow:0, black:255};
40
41     match c
42     {
43         Color::Red => println!("RED"),
44         Color::Blue => println!("BLUE"),
45         Color::Green => println!("GREEN"),
46         Color::RgbColor(r,g,b) | Color::CmykColor{cyan:_,magenta:_,yellow:_,black:255} => println!("Black"),
47         Color::RgbColor(r,g,b) => println!("rgb({},{},{})",r,g,b),
48         Color::CmykColor{cyan:cy,magenta:ma,yellow:ye,black:bl} => println!("cmyk {}, {}, {}, {}",cy,ma,ye,bl),
49     }
50 }

```

Unions

- ❖ Union'ların C/C++'daki gibi kullanımları Rust içinde de geçerlidir.
- ❖ Variable'ları 32 bit olarak tanımlanırlar.
- ❖ Özel bir data tipidir, farklı data tiplerini (int, float gibi) aynı yerde memoryde tutmaya yarar. Aynı memory lokasyonunu birden fazla amaç için kullanmaya yaradığından verimlidir. İçerisindeki en büyük data type'ın büyüklüğünde bir memory'de yer ayırılır. Aslında içindeki sadece bir data type'ını tutar fakat isteğe bağlı bunun tipini ayarlayabilmiş oluyoruz, "auto" keywordün işlevine benziyor gibi diyebiliriz.
- ❖ Aşağıdaki örnekte biz mesela union içindeki int'ı sadece tanımladık, fakat float değerini okumaya çalışıyoruz. Böyle bir durumun olabilesinden dolayı, Rust'da union içindeki variable'ı okumak için "unsafe" block içinde bu okuma işlemini yapmamız gerekir. Çünkü C++'daki union örneklerinde, son atanmamış değeri okumaya çalıştığımızda dummy/random değerler dönüyor, bu da runtime'da tehlike arz ediyor. Union variable'ına yazma işlemi için herhangi bir kısıtlama yoktur.

```

53 //32 bit
54 union IntOrFloat
55 {
56     i:i32,
57     f:f32
58 }
59
60 fn unions(){
61     let mut iof= IntOrFloat{ i:123 };
62     iof.i=234;
63
64     let value = unsafe{ iof.i };
65     println!("iof.i ={}",value);
66
67 }

```

- ❖ “match” ile union’ları okuyacak bir yapı kurduğumuzda bunun okuma işlemi yapmasından dolayı unsafe block içerisinde olması gerekir. Ayrıca match caselerinde uç noktaları da düşünmemiz gerekir. Örneğin burada int=5 durumunu float case’i ele almaya çalışır, type casting yapar ve sonuç yanlış gelir.

```
69 fn unions_process_value(iof: IntOrFloat){
70     unsafe{//should be in unsafe block, because we read variables in union
71         match iof{
72             IntOrFloat{i:42} => {println!("meaning of life value");} //it only takes when int=42
73             IntOrFloat{f} => {println!("value = {}",f);} //it takes all float variables
74         }
75     }
76 }
77
78 fn main(){
79     //structures();
80     //enums();
81     //unions();
82     unions_process_value(IntOrFloat{i:42});
83     unions_process_value(IntOrFloat{f:40.5});
84     unions_process_value(IntOrFloat{i:5}); //It is treated as float number but the result is not seen correct.
85 }
```

Option<T> Some/None and If Let/While Let

- ❖ Rust içerisinde “Option<T> -> Some(v) | None” olarak geçen bir kalıp vardır. Sıfıra bölme durumlarında vs bu seçeneği kullanabiliriz. Keywordlerin “Some” ve “None” olması gerekiyor.
- ❖ Ayrıca “if let” ve “while let” şeklinde “Option” ile kullanabileceğimiz bir kullanım da vardır. “result”ın Some(v)’ye eşit olduğu durumlar için gerçekleştirilecek, yapılacak kodların bütünü içerirler.
- ❖ API’lar için işleri kolaylaştırmak amacıyla oluşturulmuş keywordlerdir.
- ❖ Bir fonksiyondan dönecek sonucun gerçekten anlamlı bir şey olup olmadığını yani gerçek bir değere mi eşit yoksa tam olarak bir değer döndürmemekte mi; şeklinde sorulara cevabı bu yapıyı kullanarak veririz. Some(v) sonucun bir değer döndürmüş olduğunu belirtirken, None sonucun bir değer döndürmediğini belirtir.

```

1  fn main(){
2      let x=3.0;
3      let y=2.0;
4
5      //Option<float> -> Some(v) | None
6      let result= if y!= 0.0 {Some(x/y)} else {None};
7
8
9      match result{
10         Some(z) => println!("{}",x,y,z),
11         None => println!("cannot divide by zero")
12     }
13
14     if let Some(z) = result {
15         println!("result={}",z);
16     }
17     while let Some(z) =result{
18         println!("resultWithWhile={}",z);
19         break;
20     }
21 }

```

```

3/2=1.5
result=1.5
resultWithWhile=1.5

```

Arrays

- ❖ Array tipi belirtilirken “[type_element; number_elements]” şeklinde belirtilmesi gerekiyor.
- ❖ Array’ın bütün elemanlarını aynı değere initialize etmek istersek de üstteki yapıyı type_element yerine değer yazarak eşitleriz, “[value; number_elements]” gibi. “value”nun tip ve bit sayısını belirterek de yazabiliriz ([1u16;10]’da 1 yazmak yerine 1u16 gibi, 1 yazarsak 32 bit olacaktı.).
- ❖ “{:?}” (debug) işaretini kullanarak bir array’ın bütün elemanlarını yazdırabiliyoruz.
- ❖ If örneğinde görüldüğü şekilde array karşılaştırması yapabiliriz.

```

1  use std::mem;
2
3  fn arrays(){
4      let mut a:[i32;5]=[1,2,3,4,5]; //Type of elements are i32, Number of elements=5
5
6      println!("a has {} elements, first is {}", a.len(), a[0]);
7      a[0]=321;
8      println!("a[0]={}",a[0]);
9      println!("{:?}",a); //Print all elements of a
10
11      if a == [321,2,3,4,5] ←
12      {
13          println!("ok");
14      }
15      let b=[1u16;10]; ← //1 which is 16 bit //b=[1;10]>1 which is 32 bit //Initialize 10 elements with 1. //b.len()=10
16      for i in 0..b.len()
17      {
18          println!("{}",b[i]);
19      }
20      println!("b took up {} bytes", mem::size_of_val(&b)); //20 bytes if 16 bit, 40 bytes if 32 bit
21 }

```

- ❖ 2D array oluşturmak için de iç içe [[]] oluşturulacak şekilde C++ benzeri bir yapı kurulur. İçteki parantez içerisinde element_type ve column_number verilirken, dıştaki parantez içinde

row_number verilir. Hem iç içe kurulmuş for loop ile hem de {:?}(debug) ile 2D arrayin elemanlarına bakabiliriz.

```
22 let matrix:[f32;3];2] =
23 [
24     [1.0, 0.0, 0.0],
25     [0.0, 2.0, 3.0]
26 ];
27
28 println!("{:?}",matrix);
29
30 for i in 0..matrix.len() {
31     for j in 0..matrix[i].len() {
32         if i==j {
33             println!("matrix[{}][{}]={}",i,j,matrix[i][j]);
34         }
35     }
36 }
```

- ❖
- ❖ Array'lerin büyüklükleri sabittir. Farklı boyutta array yapılarını fonksiyon argümanlarında vs kullanabilmek için "slice" data yapıları kullanılır. Var olan bir array'in belirli boyutlarına referans olan bir yardımcı veri tipidir diyebiliriz.

Slices

- ❖ Array'in bir çeşididir fakat büyüklükleri(size) compile-time'da compiler tarafından bilinmez. Array'in değişken bir parçasını veya anlık oluşturulmuş değişken boyuttaki bir arrayi tutmaya yarar.
- ❖ "&mut" olarak vermemiz gerekti, mutable olduğunu bildirmek ve değişimini sağlamak için. Slice'ları reference olarak vermeliyiz, bu şekilde vermeyince slice'lar için compile error veriyor (slice olduğunu ayırt edemiyor eğer & olarak vermezsek, çünkü slice'lar var olan array'e refer ediyor, bu nedenle onun adresini tutmalı.)

```
40 fn use_slice(slice: &mut [i32]) //to make slice mutable, add mut keyword
41 {
42     println!("first element of slice={},len={}",slice[0],slice.len());
43     slice[0]=1453;
44 }
45
46 fn slices()
47 {
48     let mut data =[1,2,3,4,5];
49
50     use_slice(&mut data[1..4]); ←
51     //use_slice(&mut data);
52
53     println!("{:?}",data);
54 }
```

- ❖
- ❖ first element of slice=2,len=3
[1, 1453, 3, 4, 5]

Tuples

- ❖ Tuple “(a,b,c,...)” şeklinde kullanılan kümelerdir. Diğer dillerdeki gibi kullanımı Rust’da mevcuttur.
- ❖ “st=(a,b)” şeklindeki s tuple’ını parçalarına ayırabiliyoruz. Buna “destructuring” deniyor. “let (a,b)=st” şeklinde yapabiliriz.
- ❖ Tuple’ın elemanlarına “st.0” ve “st.1” diyerek ulaşırız.
- ❖ Birden fazla tuple da bir tuple oluşturabilir. Bunu “tp=((c,d),(e,f))” olarak oluşturabilir hem destructuring yapabiliriz.
- ❖ Tuple elemanlarının aynı tipte olmasına gerek yoktur. (int, float, bool) olabilir, “foo” örneği.
- ❖ “(x,)” yazılarak tek elementi olan bir tuple de yapılabilir, “meaning” örneğindeki gibi.
- ❖ “println!(“{0},{1},{0},{2}”,x,y,z)” için farklı bir kullanım mevcut. “{ }” içlerini 0’dan başlayarak numaralandırınca, o bölgelere numaralara göre bizim variable’larımızı yerleştirir. Örneğin 0 yazan yerlere ‘x’, 1 yazan yerlere ‘y’, 2 yazan yerlere ‘z’ gelecektir.

```
56 fn tuples_sum_and_product(x:i32, y:i32)->(i32,i32)
57 {
58     (x+y,x*y)
59 }
60
61 fn tuples(){
62     let x=3;
63     let y=4;
64     let sp=tuples_sum_and_product(x,y);
65
66     println!("sp= {:?}",sp);
67     println!("{0} + {1} = {2}, {0} * {1} = {3}",x,y,sp.0,sp.1);
68
69     //destructuring
70     let (a,b) =sp; ←
71     println!("a ={}, b={}",a,b);
72
73     let sp2=tuples_sum_and_product(4,7);
74     let combined=(sp,sp2); ←
75     println!("{:?}",combined);
76     println!("last elem={}",(combined.1).1);
77
78     let ((c,d),(e,f))=combined; ←
79     println!("c={}, f={}",c,f);
80     let combined2=((c,d),(e,f));
81     println!("last elem={}",(combined2.1).1);
82
83     let foo=(true,42.0,-1i8); ←
84     println!("{:?}",foo);
85
86     let meaning=(42,); ←
87     println!("{:?}",meaning);
88 }
```

Pattern Matching

- ❖ “match” caselerinde “9...11” diye veya “9..=11” yazarsak 11’de aralığa dahil edecek. Eğer 3 nokta değil 2 nokta yazsaydık dahil etmeyecekti. Ayrıca range’lere isim verebiliriz. Burada “z @ range” diyerek o range için z ismini vermiş olduk, z’yi range’i belirtmek için kullanabiliriz.
- ❖ “_ if (x%2==0) => “some” dediğimizde, “_” yazarak x’in ne olduğunu önemsemiyoruz, bu şartın sağlanmasını önemsiyoruz diye belirttik.

```

1 fn pm_how_many(x:i32) -> &'static str //we should define return as &'static str
2 {
3
4     match x
5     {
6         0 => "no",
7         1|2 => "one or two",
8         12 => "dozen",
9         z @ 9...11 => "lots of", //inclusive range //give a name to this range, call this range with this name
10        _ if (x%2 == 0) => "some",
11        _ => "a few",
12    }
13 }
14
15 fn pattern_matching(){
16
17     for x in 0..13{
18         println!("{}", I have {} oranges",x,pm_how_many(x));
19     }
20
21     let point=(3,4);
22     match point
23     {
24         (0,0)=>println!("{}",origin");
25         (0,y)=>println!("{}",y axis, {},y);
26         (x,0)=>println!("{}",x axis, {},x); //If we want we can declare x as reference and mutable with "(ref mut x,0)=>..."
27         (_,y)=>println!("{}",?,{})"y); //don't care x
28     }

```

- ❖ “_” yerine yeni bir kullanım yöntemi olarak, enum örneğinde kullandığımız “_(don’t care)”ler için, her bir enum parametresini “_” ile belirtmek yerine “..” kullanarak kalan parametrelerin önemsiz olduklarını belirtebiliriz.

```

30 //We can use it ".." instead of "_"(dont care) in enum cases.
31 let c:Color = Color::CmykColor{cyan:0, magenta:128, yellow:0, black:255};
32 match c
33 {
34     //Color::RgbColor(0,0,0) | Color::CmykColor{cyan:_,magenta:_,yellow:_,black:255} => println!("Black"),
35     Color::RgbColor(0,0,0) | Color::CmykColor{black:255,..} => println!("Black"), //Same before!
36     _ => ()
37 }

```

Generics

- ❖ C’nin template’leridir.
- ❖ Eğer type’ını belirtmek istersek aşağıdaki örnek gibi kullanır, T ve V’yi belirtiriz (belirtmesek de olurdu, sadece let a=Point{...} yeterli.).

```

39 // Option<T,V>
40 struct Point<T,V>
41 {
42     x:T,
43     y:V
44 }
45
46 fn generics(){
47     let a:Point<u16,i32> = Point{ x:0,y:4 };
48     let b:Point<f64,f64> = Point{ x:1.2,y:3.4};
49 }

```

Standard Collections

Collection	What is it?	C++	C#	Java	Python
Vec<T>	Dynamic (growable) array	vector	List	ArrayList	list
VecDeque<T>	Double-ended queue	deque	—	ArrayDeque	collections.deque
LinkedList<T>	Doubly linked list	list	LinkedList	LinkedList	—
BinaryHeap<T> where T : Ord	Max heap	priority_queue	—	PriorityQueue	heapq
HashMap<K,V> Where K : Eq+Hash	Dictionary (key-value table)	unordered_map	Dictionary	HashMap	dict
BTreeMap<K,V> Where K : Ord	Sorted dictionary (key-value table)	map	SortedDictionary	TreeMap	—
HashSet<T> where T : Eq + Hash	Hashtable	unordered_set	HashSet	HashSet	set
BTreeSet<T> where T : Ord	Sorted set	set	SortedSet	TreeSet	—

Taken from Dmitri Nesteruk

Vectors

- ❖ Fixed size olmayan dynamic-sized bir array oluşturmak için kullanılır.
- ❖ Vector oluştururken “Vec::new()” keyword’ü kullanılır. “mut” keywordü vector doğası gereği zorunludur.
- ❖ Bir vectorün indexi için bilgisayarın memory size’ından(32bit-64bit gibi) büyük olamayacağı ve negatif olamayacağı kesindir. Bu nedenle index için variable tanımlarken, type’ı için “usize” mantıklı olacaktır (hem unsigned hem bilgisayar memory size’ı ile aynı). “signed” olursa veya bilgisayar size’ından büyük olursa hata verecektir.
- ❖ Baktığımız index’te bir element olmazsa program “panic” verir, yani hata verir, bu nedenle güvenli bir şekilde bunu ele almak için “get” function’ından yararlanabiliriz.

“vector_name.get(index)” ile o indexteki elemente ulaşabiliriz ve “get”, bize “Option” type’ı döner böylece “Some,None”ları kullanarak programın çakması engellenir.

```
1 fn vectors(){
2     let mut a = Vec::new();
3     a.push(1);
4     a.push(2);
5     a.push(3);
6     println!("a={:?}",a);
7     a.push(4);
8     println!("a={:?}",a);
9
10    let idx:usize = 0;
11    a[idx]=312;
12    println!("a[0]={}",a[idx]);
13
14    //Option
15    match a.get(6) //Returns a "Option" //There is no element at 6 index
16    {
17        Some(x) => println!("a[3]={}",x),
18        None => println!("error, no such element")
19    }
```

```
a=[1, 2, 3]
a=[1, 2, 3, 4]
a[0]=312
error, no such element
```

- ❖ For loop kullanarak vector içerisinde dönebiliriz. Burada dikkat etmemiz gereken şey vectorün reference olarak for loop’a vermemiz gerektiğidir. Aksi halde, for loop bittikten sonra o vector’ü tekrar kullanamayacağız, çünkü for loop’taki iterator o vectorü taşıyor ve ownership’liği ona geçiyor. Bunun engellenmesi için vector reference olarak verilir. Aslında for loop içindeki “println!("{}",x)”deki x’i dereference yaparak “*x” vermek daha doğru olacak. Fakat Rust’da bunu yapmaya gerek yok, kendisi içerikten bunu direk çözüp ona göre print’te bastırıyor.
- ❖ Vector’e “push” diyerek sonuna eleman eklerken “pop” diyerek sondaki elemanı vectordən çıkarırız. “pop”, “Option” dönecektir, yani “Some,None” ikilisini döndürecektir, pop edilen elementi alabilmek için if-let veya while-let kullanılabilir.

```
20
21 for x in &a { //Iterating in vector //Give it as reference, because iterating with loop on vector, moves it and not to lose it use reference.
22     println!("{}",x);
23 }
24
25 a.push(77);
26 a.push(88);
27 a.push(99);
28 println!("a={:?}",a);
29
30 let last_element_option = a.pop(); //Pop returns a type of "Option".
31 println!("Last element is {:?}, a={:?}",last_element_option,a);
32
33 if let Some(x) = a.pop()
34 {
35     println!("Last element with if-let: {}",x);
36 }
37 while let Some(x) = a.pop()//Iterate all elements by popping
38 {
39     println!("Elements with while-let: {}",x);
40 }
41 }
```

```

312
2
3
4
a=[312, 2, 3, 4, 77, 88, 99]
last element is Some(99), a=[312, 2, 3, 4, 77, 88]
Last element with if-let: 88
Elements with while-let: 77
Elements with while-let: 4
Elements with while-let: 3
Elements with while-let: 2
Elements with while-let: 312

```

HashMap

- ❖ Kullanmak için “use std::collections::HashMap;” diyerek modülünü dahil ederiz öncelikle.
- ❖ Map yaparak (key,value) şeklinde depolama yapar.
- ❖ “HashMap::new()” keywordü ile oluşturulur. “mut” olması gerekir doğası gereği bu da.
- ❖ “insert” methodu ile map’e element eklenir. Eğer o element önceden map’te var ise üzerine yazılır. Bu durum tehlike yaratabileceğinden ötürü, “entry(key).or_insert(value)” methodu vardır. Bu method ekleme yapmadan önce map’te eklenecek key’i kontrol eder, yoksa eğer ekler, varsa bir değişiklik yapmaz.
- ❖ For loop ile içerisinde gezinilebilir ve (key,value) şeklinde elementleri ayrı ayrı gözlenebilir. Ayrıca debug printi ile de gözlenebilir.
- ❖ “entry(key).or_insert(value)” methodu reference olarak elementi dönmektedir, buna atama yaparak elementin “value”sunu değiştirebiliriz.

```

47 fn hashMaps(){
48
49     let mut shapes = HashMap::new();
50
51     shapes.insert(String::from("triangle"),3);
52     shapes.insert(String::from("square"),4);
53
54     for (key,value) in &shapes{
55         println!("{}",key,value);
56     }
57
58     shapes.insert("square".into(),5); //it overwrites the previous
59     println!("{}",shapes);
60
61     //".entry" checks whether "circle" exists before.
62     //If not, add it. If yes, don't change anything.
63     shapes.entry("circle".into()).or_insert(1);
64
65     {
66         let actual = shapes.entry("circle".into()).or_insert(2); //this returns a reference to element
67         *actual=0; //make it 0, inside or outside scope, it becomes 0 due to referenceness.
68     }
69
70     println!("{}",shapes);
71     println!("{}",shapes["square"]);
72 }

```

HashSet

- ❖ Kullanmak için "use std::collections::HashSet;" diyerek modülünü dahil ederiz öncelikle.
- ❖ İsmi gibi set'leri kümeleri temsil eder. İçerisindeki her elemandan tek bir tane olduğunun garantisi vardır. Aynı elemandan eklemeye çalışırsak eklemeyecektir, bir şey değiştirmeyecektir. Ayrıca küme içerisinde eleman sırası yoktur.
- ❖ "insert" methodu ile kümeye eleman eklenir, "remove" methodu ile kümeden eleman çıkarılır. "contains" methodu ile kümede elemanın olup olmadığı sorgulanır.

```
84 //HASHSETS
85 fn hashSets(){
86
87     let mut greeks = HashSet::new();
88     greeks.insert("gamma");
89     greeks.insert("delta");
90     println!("{:?}",greeks);
91
92     let added_vega = greeks.insert("vega"); //return boolean
93     if added_vega{//if "vega" already exist , "else" is working.
94         println!("we added vega!");
95     }
96
97     if !greeks.contains("kappa"){ //return boolean
98         println!("we dont have kappa");
99     }
100
101     let removed=greeks.remove("delta");
102     if removed{
103         println!("we removed delta");
104     }
105     println!("{:?}",greeks);
}
```

- ❖ "given_range.collect()" methodu ile bir hash set oluşturulabilir.
- ❖ Set'ler üzerinde kümeler için normalde kullandığımız; birleşim,keşişim,farkı,alt küme vs gibi işlemler uygulanabilir:"is_subset","is_disjoint","union","intersection","difference"...

```
106
107     let _1_5: HashSet<_> = (1..5).collect();
108     let _6_10: HashSet<_> = (6..10).collect();
109     let _1_10: HashSet<_> = (1..10).collect();
110     let _2_8: HashSet<_> = (2..8).collect();
111
112     //subset
113     println!("is {:?} a subset of {:?}? {:?}",_2_8,_1_10,_2_8.is_subset(&_1_10));
114     //disjoint=no common elements
115     println!("{:?} and {:?} disjoint? {:?}",_1_5,_6_10,_1_5.is_disjoint(&_6_10));
116     //union
117     println!("items in either {:?} and {:?} are {:?}",_2_8,_6_10,_2_8.union(&_6_10));
118     //intersection =>> _1_10.intersection(&_2_8)
119     //difference =>> _1_10.difference(&_2_8)
120     //symmetric difference=union-intersection =>> _1_10.symmetric_difference(&_2_8)
121 }
```

```

{"gamma", "delta"}
we added vega!
we dont have kappa
we removed delta
{"gamma", "vega"}
is {4, 3, 7, 2, 5, 6} a subset of {5, 3, 9, 7, 4, 8, 1, 2, 6}? true
{2, 3, 4, 1} and {7, 9, 8, 6} disjoint? true
items in either {4, 3, 7, 2, 5, 6} and {7, 9, 8, 6} are [4, 3, 7, 2, 5, 6, 9, 8]

```

Iterators

- ❖ For loop ile vector içerisinde iterate ettiğimizde, vectorün iteration'da taşınmaması için reference olarak vectorü verip dereference ile onun elemanlarına erişiyoruz. Rust'ta print kullanırken dereference operatörünü kullanmasak da oluyor, içerikten kendisi asıl yapmaya çalıştığımız şeyi anlıyor. Eğer elemanları sadece print ile kullanmayacaksak, her zaman dereference kullanarak o elemanlara erişmemiz gerekiyor.
- ❖ "vector_name.iter()" ile bu elemanlara iterator yardımı ile direk referenceli olarak da erişebilme seçeneğimiz var. Bunun içinde print yaparken dereference veya dereference'sız elemanlara erişebiliriz. Diğer durumlarda dereference ile o elemanlara erişmemiz gerekiyor.

```

124 //ITERATORS
125 fn iterators(){
126 |
127 |     let vec=vec![3,2,1]; //immutable
128 |
129 |     for x in &vec
130 |     { //For the println, we can also write only "x", because Rust can understand whether it should be dereferenced or not.
131 |         println!("{}",*x); //It adjusts it itself.
132 |     }
133 |     for x in vec.iter() //immutable
134 |     {
135 |         println!("we got {}",x); //Same rules are valid in also this case. x or *x
136 |     }

```

- ❖ Eğer vector elemanlarını değiştireceksek hem vectorü mutable yapmamız gerekiyor, hem de "vector_name.iter_mut()" iterator'ı mutable olarak kullanmamız gerekiyor.
- ❖ Ters sırayla (reverse order) iterate yapmak istersek de "vector_name.iter().rev()" ile iterator yapısı kurarız.

```

138 let mut vec=vec![3,2,1]; //mutable
139 for x in vec.iter_mut() //mutable iterator
140 {
141 |     *x += 2;
142 | }
143 println!("{:?}",vec);
144
145 for x in vec.iter().rev(){
146 |     println!("in reverse: {}",x);
147 | }

```

- ❖ "move" operasyonu bir collection'ının elemanlarını iterator ile collection dışına çıkarılıp taşınmasıdır.

- ❖ “vector_name.into_iter()” methodu kullanılırken vectore “move” operasyonu uygulanıyor, o vectorün tüm elemanlarını alıp (“borrow”) iterator içine yerleştiriyor, bir dönüşüme sokuyor ve o move operasyonu uygulanan ilk vector loop sonrası kullanılamaz hale geliyor.
- ❖ “vector_name2.extend(vector_name1)” methodu kullanılırken vec2’nin sonuna vec1 eklenir. Bu method aslında arka planda “into_iter()” methodunu kullanıyor. Bu nedenle vec1, bu işlemten sonra kullanılamıyor.

```
149 let mut vec1= vec![3,2,1];
150 let mut vec2= vec![1,2,3];
151 vec2.extend(vec1);
152 println!("{:?}",vec2);
153 //println!("{:?}",vec1); //gives error, because vec1 was moved due to into_iter() which is in extend().
```

Strings

- ❖ String, aslında karakter’lerden oluşan bir vector’dur.
- ❖ String type’ı aslında direk “str”den oluşmuyor. String’in iki farklı kullanım tipi vardır. Birisi “&static str” type’ından oluşuyor. “&str”, “string slice” anlamına geliyor. Buradaki “static” keywordü ile ise programdaki belirli bir lokasyona reference olup, compile edilirken oraya sabitlenir ve program sonuna kadar yani ‘static lifetime’ı(ileride bahsedilecek) kadar geçerli olur. Bu string tipindeki string değişkenleri initalize ettikten sonra değiştiremeyiz, karakter bazlı olarak string’in bir karakterine(s[0] gibi) erişemeyiz.
- ❖ Fakat bazı methodlar ile bu karakterlere erişme şansını sunar Rust. “string_name.chars()” ile string karakterlerine for loop içinde erişebiliyoruz.”rev()” methodu ile reverse order ile de erişebiliriz. “string_name.chars().nth(index)” ise Option tipinde karakterleri döndürür, if-let ile kullanabiliriz.

```
1 fn strings(){
2     //utf-8
3     let s : &'static str = "hello there"; //&str=string slice
4     //s="abc"; //gives error, Changing static string is not allowed.
5     //let h=s[0]; //gives error, Reading char of static string is not allowed.
6
7     for c in s.chars().rev()
8     {
9         println!("{}",c);
10        //c='s'; //gives error
11    }
12    if let Some(first_char) = s.chars().nth(0)
13    {
14        println!("first letter is {}",first_char);
15    }
16 }
```

- ❖ Eğer stringi runtime’da (compile-time’da da oluşturulabilir) oluşturursaydık, durumlar static string gibi olmayacaktı, string memorydeki bir alana sahip olup, o memory alanı ile işlem yapacaktık. Bu tarz string’ler için “heap”te yer ayrılacaktır.
- ❖ Bu string kullanım tipini vector gibi büyütebilecek, değişebilecek string’lerde kullanırız. Bunu “String” olarak (ilk harfi büyük) tanımlıyoruz.
- ❖ “String::new()”ile oluştururuz. “push” ile karakter(char) ekleyebiliriz. Karakterleri “as u8” veya “as char” olarak ne hali ile kullanacağımızı belirtiriz. Eğer “,” gibi karakterler ekleyeceksek “push_str(“,”)” gibi özel bir fonksiyonla ekleriz.


```

17 //heap
18 //String
19 let mut letters = String::new();
20 let mut a='a' as u8; //define ascii int value of 'a'
21 while a <= ['z' as u8]
22 {
23     letters.push(a as char); //define char of 'a'
24     letters.push_str(",");
25     a+=1;
26 }
27 println!("{:?}",letters);

```

- ❖ “String” tip objeden “str” objesine dönüşüm için ‘&String_name’ şeklinde reference vererek eşitleme yaparız.
- ❖ “Concatenation” için “String” ve “str”leri direk toplayabiliriz fakat “z1” örneğinde birleştirme sonrası “letters”a erişimi kaybederiz. Eğer iki “String” tipi ile toplamak istersek 2. elemanını ‘&String_name’ şeklinde vermemiz gerekir. Aynı String’i art arda toplamak ise mümkün olmadığı için birleştirme esnasında aynı elemana tekrar erişmeye çalışıyor, ama erişim öncesinde kayboluyor.
- ❖ “str”dan “String”e bir dönüşüm gerçekleştirmek istersek de “String::from(“...”)” yapısı ve “.to_string()” kalıplarını kullanırız.
- ❖ “remove(index)”, “push_str(“...””, “replace(existed,will_place)” methodları String için de aşağıdaki gibi kullanılabilir.

```

29 //&str <= String
30 let u: &str = &letters;
31
32 //concatenation
33 //String+str
34 //let z1= letters + "abc";
35 let letters2="ggzytm".to_string(); //String
36 let z = letters + &letters2; //String+String //think &letters2(&String) as string which is &str type
37 //let z = letters + &letters; //NOT VALID |
38
39 println!("{:?}",z);
40
41 //String <= str
42 let mut abc1= String::from("Hello");
43 let mut abc = "hello world".to_string();
44 abc.remove(0);
45 abc.push_str("!!!");
46 println!("{}",abc.replace("ello","goodbye"));

```

String Formatting (format!)

- Bir Rust macro’sudur (sonunda ünlem işareti var oradan anlaşılabilir.). Print macrosu gibidir fakat String döndürür. C++’da “sscanf” görevini görür.
- “format!” macrosuna argüman verirken 3 farklı yol vardır. Birinci yol klasik olarak “{” yazarak sırasıyla argümanları vermektir. İkinci yol ise argümanların yazılış sırasına göre numaralanmasını sağlayıp “{1}” şeklinde o argümanın oraya yerleşmesini sağlamaktır. Üçüncü yol ise

“{argument_name}” yazıp “argument_name=argument” şeklinde argument_name yazan yerlerin yerine argument’lerin gelmesini sağlamaktır.

```
55 fn stringFormatting() {
56
57     let name="Dmitri";
58     let greeting = format!("hi,I'm {}, nice to meet you",name);
59     println!("{}",greeting);
60
61     let hello ="hello";
62     let rust ="rust";
63     let hello_rust = format!("{}", {}!",hello,rust);
64     println!("{}",hello_rust);
65
66     let run="run";
67     let forest="forest";
68     let rfr= format!("{}",{1},{0}!",run,forest);
69     println!("{}",rfr);
70
71     let info= format!("the name's {last}, {first} {last}",first="James",last="Bond");
72     println!("{}",info);
}
```

- Bu 3 farklı format!’a argüman verme yol ortak olarak kullanılabilir. Burada numaralandırmalar argümanların verilme sırasına göre seçilir. Boş {} parantezlerde argümanlar tarafından sırayla doldurulur.
- Eğer kullanılmayan fazladan bir arguman olursa compiler hata verecektir. Bu nedenle bütün argümanların kullanılması gerekiyor.

```
74 let mixed = format!("{}",{1} {} {0} {} {data}", "alpha", "beta", data="delta");
75 println!("{}",mixed);
76
77 //Gives Error! "gamma" is not used.
78 //let mixed = format!("{}",{1} {} {0} {} {data}", "alpha", "beta", "gamma", data="delta");
```

```
hi,I'm Dmitri, nice to meet you
hello, rust!
run,forest,run!
the name's Bond, James Bond
beta alpha alpha beta delta
```

Functions

Functions and Function Arguments

- Functionları daha önceki örneklerde birçok kez tanımladık, burada tekrar üstünden geçeceğiz.
- Function declarationında arguman verirken type’ını belirterek tanımlarız, “x:i32” gibi.
- “&”a “shared reference” deniyor, shared” çünkü aynı anda farklı pointer’lar o lokasyonu göstermesine izin veriliyor. “&mut” da ise shared olmuyor, tek bir pointerın o lokasyonu göstermesine sadece izin veriliyor ve o pointer artık o lokasyonu ödünç almış gibi sayılıyor, onunla işlem yapılabilir. Read aynı anda yapılabilirken write tek bir şekilde yapılabilir gibi düşünebiliriz. Yani şu işlemin yapılmasına müsaade yoktur. Ownership, Borrowing konularında da detaylı inceleyeceğiz bu durumu.

```

28     let mut x=0;
29     let y=&mut x;
30     //let z=&mut x; //ERROR, x is already borrowed mutably
31     *y=1;
32     println!("{}",x); //x=1

```

- C++’daki pointer/reference ile Rust’un reference’ının bir karşılaştırılmasının yapıldığı bir tablo var. Rust ile C++ arasında ufak bir fark var. C’de pointer ve reference benzer şeyler olmasına rağmen bir ayrımı varken, Rust’da bu durum pek yok gibi (bunu reference edilen değeri okurken anlayabiliyoruz. Hep variable pointer’ını dereference operatörü ile okuyor.).

1.5.0.1 Reading The Value Without Mutation with A Pointer or Reference

step	C++ (pointer)	C++ (reference)	Rust	Rust (raw pointer)
init referent	<code>const int a = 5</code>	<code>const int& a = 5</code>	<code>let a = 5</code>	<code>let a = 5</code>
make ptr/ref	<code>const int* p = &a</code>	<code>const int& r = a</code>	<code>let r: &i32 = &a</code>	<code>let p: *const i32 = &a;</code>
read referent value	<code>*p</code>	<code>r</code>	<code>*r</code>	<code>*p</code>

1.5.0.2 Mutating the Value of the Referent with A Pointer or Reference

step	C++ (pointer)	C++ (reference)	Rust	Rust (raw pointer)
init referent	<code>int a = 5</code>	<code>int a = 5</code>	<code>let mut a = 5</code>	<code>let mut a = 5</code>
make ptr/ref	<code>int* p = &a</code>	<code>int& r = a</code>	<code>let r: &mut i32 = &mut a</code>	<code>let p: *mut i32 = &mut a;</code>
mutate	<code>*p = 10</code>	<code>r = 10</code>	<code>*r = 10</code>	<code>*p = 10</code>

-
- Rust’da reference ve değiştirebilir bir elemanı argüman olarak vereceksek hem function declaration’ında, hem functiona argüman verilirken o variable “&mut” olarak işaretlenmelidir.
- Fonksiyon return typelerini “->” ile gösteririz.

```

1  fn func_print_value(x:i32){
2      println!("value= {}",x);
3  }
4  fn func_increase(x: &mut i32){ //reference
5      *x +=1; //dereference
6  }
7  fn func_product(x:i32,y:i32)->i32
8  {
9      x*y
10 }
11
12 fn functions(){
13     func_print_value(33);
14
15     let mut z=1;
16     func_increase(&mut z); //change original variable
17     println!("z={}",z);
18
19     let a=3;
20     let b=5;
21     let p=func_product(a,b);
22     println!("{}",*{}={}",a,b,p);

```

Methods

- Struct için class methodları gibi fonksiyonlar yazabiliriz. Bu fonksiyonlara da “method” diyoruz. Bu tanımlanacak methodları “impl struct_name” keywordü ile tanımlanan alana yazıyoruz.
- Bu methodlarda argüman olarak “&self” veririz ki, böylece struct’ın içindeki variable’lara ulaşımı mümkün kılarız. Variable’lara “self.variable” şeklinde erişim sağlayabiliriz.

```

38 struct Point
39 {
40     x:f64,
41     y:f64
42 }
43 struct Line
44 {
45     start: Point,
46     end: Point
47 }
48 impl Line //methods for struct Line
49 {
50     fn len(&self)->f64{
51         let dx=self.start.x-self.end.x;
52         let dy=self.start.y-self.end.y;
53         (dx*dx+dy*dy).sqrt()
54     }
55 }
56
57 fn methods(){
58     let p = Point{ x: 3.0, y:4.0};
59     let p2 = Point{ x:5.0, y: 10.0};
60     let myLine = Line {start:p , end:p2};
61     println!("length={}",myLine.len());
62 }

```

Closures

- Function'ları variable'lar içerisinde tutabiliriz.
- Closure'da buna benzer bir işlev görür. C++'daki "lambda function"a benziyor gibi. Bir variable'a karşılık argüman ve return tipleri işlev function içeriği eşitlenir. Daha sonra o variable çağrılır function çağrılır gibi. İstersek argument ve return'leri auto olarak closure'ın tiplerini kendisinin tahmin etmesini bekleriz, istersek de tiplerini açık olarak verebiliriz.
- Closure argumentlerini "[...]" içerisinde verirken, return type'ını "->" ile veririz. Argument type'ını ve return type'ını otomatik tahmin etmesini beklersek bunları girmeyiz. Argument yoksa "||" şeklinde boş olarak belirtilir.

```
66 fn clo_say_hello(){ println!("Hello");}
67
68 fn closures(){
69     let sh = clo_say_hello; //Variable can store function
70     sh();
71
72     let plus_one = |x:i32| -> i32 {x+1};
73     let a=6;
74     println!("{}",a,plus_one(a));
75
76     let two=2; ←
77     let plus_two = |x|
78     {
79         let mut z=x;
80         z+=two;
81         z
82     };
83     println!("{}",3,plus_two(3));
84 }
```

- Yukarıdaki örnekte olduğu gibi, closure'lar scope'u dışındaki variable'ları(two) kendi scope'u içerisinde kullanabilir. Burada dikkat edilmesi gereken closure'ın 'two' variable'ını kullanırken onu borrow etmesidir. Bu nedenle bu functiondan sonra aşağıdaki gibi "let borrow_two = &mut two" dediğimizde compile hatası alacağız, daha önce borrow edildiğine dair. Bunu engellemek için bu closure'ı bir scope içine alacağız ki scope bitince sanki closure ölmüş gibi olacak, borrow ettiği variable'ları serbest bırakacak. DÜZENLEME: Denendiğinde, bu durumun dinamik değişkenler için geçerli olduğu görülmüştür. DÜZENLEME2: Tekrar denendiğinde dinamikler için de görülmemiştir. Closure'a argüman olarak verilirse move edilmiş olduğu görüldü. Kontrol edilecek bir daha. UNUTMA.

```
76 let mut two=2;
77 {
78     let plus_two = |x|
79     {
80         let mut z=x;
81         z+=two;
82         z
83     };
84     println!("{}",3,plus_two(3));
85 }
86 let borrow_two=&mut two; ←
```

-

- Argumanı reference olarak verirsek, closure içerisinde argümanı değiştirebiliriz. “&mut” olarak argüman type’ı belirlenir.

```

93 let plus_three= |x: &mut i32| *x+=3; //give it as referenced
94
95 let mut f=12;
96 plus_three(&mut f);
97 println!("f={}",f);

```

High-Order Functions

- Argument olarak function alan fonksiyonlara ve return type olarak function döndüren(generator) functionlara denir.
- Function döndüren fonksiyonlar “-> impl Fn(type) -> return type” şeklinde signature alırlar. Buradaki Fn daha sonraki konularda işlenecek.

```

110 fn HO_is_even(x:u32)->bool{
111     x%2==0
112 }
113
114 fn HO_greater_than(limit:u32) -> impl Fn(u32) -> bool //function which return function, must take these signatures
115 {
116     move |y| y>limit // put a move keyword to extend lifetime of y
117 }
118
119 fn highOrderFunctions(){
120
121     let limit =500;
122     let mut sum=0;
123     let above_limit = |y:u32| y>limit; //Closure
124     let above_limit2 = HO_greater_than(limit); //For the case which "function returns function"
125
126     for i in 0..{
127         let isq = i*i;
128
129         //if isq>limit {break;} //Normal
130         //if above_limit(isq) {break;} //Closure
131         if above_limit2(isq) {break;} //For the case which "function returns function"
132
133         else if HO_is_even(isq){ sum += isq; }
134     }
135 }
136

```

- Functionları arguman olarak da aşağıdaki şekilde verebiliriz. Aşağıdaki fonksiyonlar özel fonksiyonlar olarak kullanılıyor Rust içinde.

```

137 //map:takes a value and transforms to other value
138 //take_while: controls the loop condition
139 //filter: to filter,check condition
140 //fold: pairwise operation like accumulator
141 let sum2=(0..)
142     .map(|x| x*x)
143     .take_while(|&x| x<limit)
144     .filter(|x:&u32| HO_is_even(*x))
145     .fold(0,|sum,x| sum+x);
146
147 println!("highOrderFunc sum={}",sum2);

```

Traits

Trait

- Object Oriented Programming için gerekli olan, Rust'ın temel keywordüdür.
- Aslında C++'daki class'lara ve interface özelliklerine denktir. Bunun içerisinde tanımlanmış fonksiyonlarla inheritance, interface gibi OOP özellikleri kullanılır.
- Traitler struct'lar için kullanılır. Onlar için parent class fonksiyonlarını tutan class gibi davranabilir.
- Oluşturduğumuz struct üzerine trait kullanarak inheritance uygulayabiliriz. Bunun için "impl trait_name for struct_name" (implement) şeklinde bir keyword kullanırız.
- "&self" argumanı alan fonksiyonlar genelde "instance function" olarak geçer. Eğer static function oluşturmak istersek buna "&self" argumanı verilmez. Buradaki "create" function static'tir ve instance döner. Instance dönmek için factory trait'inde "Self"i (ilk harfi büyük) return type olarak kullandık. Factoryden inherit eden structları ise struct isimleri ile döndük.

```
1  #![allow(dead_code)]
2
3  trait Animal
4  {
5      fn create(name: &'static str) -> Self; //Static function
6      fn name(&self) -> &'static str; //Instance function
7      fn talk(&self) //Instance function
8      {
9          println!("{}", cannot talk", self.name());
10     }
11 }
12 struct Human
13 {
14     name: &'static str
15 }
16 impl Animal for Human
17 {
18     fn create(name:&'static str)->Human
19     {
20         Human{name:name}
21     }
22     fn name(&self) -> &'static str
23     {
24         self.name
25     }
26     fn talk(&self)
27     {
28         println!("{}", says hello", self.name());
29     }
30 }
```

```
32 struct Cat
33 {
34     name: &'static str
35 }
36 impl Animal for Cat
37 {
38     fn create(name:&'static str)->Cat
39     {
40         Cat{name:name}
41     }
42     fn name(&self) -> &'static str
43     {
44         self.name
45     }
46     fn talk(&self)
47     {
48         println!("{}", says miyav", self.name());
49     }
50 }
```

- Factory trait üzerinden instance oluşturabiliriz, C++'daki polymorphism gibi düşünebiliriz. Hem de struct'lar üzerinden static functionlarla instance oluşturabiliriz.

```
51
52 fn traits(){
53
54     //let h= Human{name:"John"}; //Normal instance
55     //let h=Human::create("John");
56     let h:Human= Animal::create("John"); //Factory instance
57     h.talk();
58     let k=Cat{name:"Misha"};
59     k.talk();
60 }
```

- Bilindik data type'ları için de trait'leri kullanabiliyoruz. Onlara özgü fonksiyon eklemek için "impl trait_name for datatype" ile o datatype için oluşturulmuş yeni fonksiyon içeriklerini tanımlayabiliyoruz.

```

76     let a=vec![3,2,1];
77     println!("sum={}",a.sum());

52     trait Summable<T>
53     {
54         fn sum(&self)->T;
55     }
56
57     impl Summable<i32> for Vec<i32> //Use it for generic Vec<i32> type
58     {
59         fn sum(&self)->i32
60         {
61             let mut result:i32 =0;
62             for x in self { result += *x; } //self=vector
63             return result;
64         }
65     }

```

-

Trait Parameters

- Fonksiyon'a verilecek objelerin tiplerinin hangi trait'lerden türemek zorunda olduğunu göstermek için kullanılır. Sanki bir class objesini fonksiyon parametresi olarak alıyoruz gibi düşünebiliriz. 3 farklı yolu vardır.
- İlk yolu parametreyi "variable_name: impl trait_name + another_trait_name..." şeklinde vermektir. "+" ile diğer traitleri de verebiliyoruz bu yöntemde. Örnek olarak "debug" printini kullanmak üzere use ile eklediğimiz Rust'ın sunduğu "std::fmt::Debug" trait'ini aşağıdaki örnekte {:?}'yi kullanmak için veriyoruz. Bu arada struct'lardan önce "#[derive(Debug)]" yaparak bu struct'ları hazır modül "Debug"tan türetiyoruz.
- Diğer bir yöntem template'i kullanmak: "<T:trait_name + another_trait_name>(variable_name: T,another_variable_name:T)". Birden fazla trait parametresi vereceğimiz zaman bunu kullanmak üsttekine göre avantajlı olacak.
- Diğer yöntemimiz ise ikinci yönteme benzer bir biçimde: "<T>(variable_name: T) where T:trait_name + another_trait_name" şeklindedir.


```

85  #[derive(Debug)]
86  struct Circle{
87      radius:f64,
88  }
89  #[derive(Debug)]
90  struct Square{
91      side:f64,
92  }
93
94  trait Shape{
95      fn area(&self) -> f64;
96  }
97  impl Shape for Square{
98      fn area(&self)->f64{
99          self.side * self.side
100      }
101  }
102  impl Shape for Circle{
103      fn area(&self)->f64{
104          self.radius*self.radius * std::f64::consts::PI
105      }
106  }

```

```

108  //fn TP_print_info(shape: impl Shape + Debug) 1
109  //fn TP_print_info<T>(shape:T) where T:Shape+Debug }
110  fn TP_print_info<T:Shape+Debug>(shape:T) 2
111  {
112      println!("{:?}",shape);
113      println!("The area is {}",shape.area());
114  }
115  fn traitParameters(){
116      let c=Circle{radius: 2.0};
117      TP_print_info(c);
118  }

```

Into

- “str” ve “String” gibi arasındaki dönüşümlere izin veren Rust tarafından hazır sunulan trait’e denir. “From ve Into” olarak bizim belirlediğimiz ‘impl’ ile implementasyonunu yaptığımız dönüşümleri yapmayı sağlar.
- “str” verirse onu “String”e dönüşümünü sağlayacak “into()” fonksiyonunu kullanabilmemizi sağlar. “String”i “String”e dönüştürmeye çalışsa da problem olmayacaktır. Önceki konu başlığındaki gibi “Into<String>” traitimizi parametre olarak fonksiyona vererek kullanırız.
- Biz “&str” istediğimiz yere Stringi vermek için onun reference’ını vermemiz gerekiyor. “string_name.as_ref()” diyerek verebiliriz.

```

124 struct Person{ name: String}
125 impl Person{
126
127     //Previous/Normal Version
128     /*
129     fn new(name: &str)->Person{ //take argument as str
130         Person{name:name.to_string()} //convert str to String
131     }
132     */
133
134     //Version with Into Trait
135     //fn new<S>(name:S)->Person where S:Into<String> //Third way ←
136     fn new<S: Into<String>>(name:S) -> Person //Second way ←
137     {
138         Person{ name: name.into()}
139     }
140 }
141
142 fn IntoTraitFunc(){
143     let John = Person::new("John");
144     let name:String = "Jane".to_string();
145     //let jane=Person::new(name.as_ref()); //For previous/Normal version
146     let jane=Person::new(name); //For version with Into Trait
147     //println!("{:?}",jane); //Can not be used without Debug trait for case with Into Trait
148 }

```

Drop

- “Destructor” görevi gören Rust’un sunduğu diğer bir hazır trait’dir.
- Variable’ın ne zaman ve nerede yok edilmesi gerektiğine dair işlemleri sağlar. Aslında Rust’da memory’deki variableları silmek, memory işlemleri Rust’da unsafe olduğu için biz direk bir destructor kullanma işlemini yapmıyoruz, ama onun nerede kullanıldığını vs bu yolla gözlemleyebiliyoruz. Yani biz “variable_name.drop()” diye çağırmamıza compiler müsaade etmiyor. Fakat illa o variable’ı drop etmek istersek “deterministic finalization” ile global function kullanarak “drop(variable_name)” diyerek bunu yapabiliriz. Zaten scope bitince kendisi bunu(“variable_name.drop()”) otomatik çağırılmaktadır.

```

151 //DROP
152 struct Creature{
153     name:String
154 }
155 impl Creature{
156     fn new(name:&str)->Creature{
157         println!("{}", name);
158         Creature{name:name.into()}
159     }
160 }
161 impl Drop for Creature{
162     fn drop(&mut self){
163         println!("{}", self.name);
164     }
165 }
166
167 fn DropTraitFunc(){
168     let goblin=Creature::new("Jeff");
169     println!("game proceeds");
170     //goblin.drop(); //Error
171     //drop(goblin); //OK

```

```

Jeff enters the game
game proceeds
Jeff is dead

```

- Bu arada compiler variable'ları destruct ederken aşağıdaki şekilde kontrollerini yapıyor ve daha sonra kullanacağı variable'ı scope biterken silmiyor:

```

fn DropTraitFunc(){
    let mut clever:Creature;
    {
        let goblin: Creature = Creature::new(name: "Jeff");
        println!("game proceeds");
        clever=goblin;
        println!("end of scope");
    }
    println!("after ending scope");
}

```

```

Jeff Enters the game
game proceeds
end of scope
after ending scope
Jeff is dead

```

Operator Overloading

- Operator'leri tekrar spesifik olarak tanımlamak için yapılması gerekenler anlatılacaktır.
- Rust'da operator overloading trait'ler kullanılarak yapılır.
- Template bir struct'ı implement ederken "impl<T> Struct_name<T>{...}" şeklinde ederiz.
- "type" keywordü ile oluşturulmuş type'a "associated type" denir. Associated type tanımlamamız gereken typedır. Yani anlaşılır bir isimlendirme ile eşitlendiği bir type'ın yerine, o isimlendirme kullanılabilecektir ve overload edilmiş fonksiyon kalıbında kullanıldığı noktalar var ise associated type'ı kesin tanımlamamız gerekmektedir.
- "self"(ilk harfi küçük) o objenin reference'i olurken, "Self"(ilk harfi büyük) o template'de kullanılan type'ı belirtir, yani burada "Complex<i32>" oluyor.
- Add operatorünü overload edeceğimiz için "use std::ops::{Add}" modülünü include ederiz.
- Complex struct'ı için add operator overloading'i sadece Complex<i32> için değil, "Complex<T>" için generic yapmak istersek, o zaman ayrıca "+" operatörünün sonucunun T+T durumlarında T'ye eşit olduğu bilgisini vermemiz gerekiyor. Bunun için Add'i implement ederken "where T: Add<Output=T>" keywordü ile bu durumu belirtmemiz gerekiyor.

```

1  #![allow(unused_mut)]
2  #![allow(unused_variables)]
3  #![allow(dead_code)]
4  #![allow(unused_imports)]
5
6  use std::ops::{Add};
7
8  #[derive(Debug)]
9  struct Complex<T>
10 {
11     re:T,
12     im:T
13 }
14 impl<T> Complex<T>{
15     fn new(re:T,im:T)->Complex<T>{
16         Complex::<T> {re,im}
17     }
18 }
19
20 //To make generic, instead of i32, write T,
21 //and also specify T supports "addition(+)" with "where" keyword
22 //to say output is also T for "+" like "self.re + rhs.re"
23
24 //impl Add for Complex<i32> {
25 // type Output = Complex<i32>;
26 impl<T> Add for Complex<T>
27     where T:Add<Output = T>
28 {
29     type Output = Complex<T>;
30     fn add(self, rhs:Self) -> Self::Output{
31         Complex{
32             re: self.re+rhs.re,
33             im: self.im+rhs.im
34         }
35     }
36 }
37
38 fn main(){
39
40     let mut a=Complex::new(1.0,2.0);
41     let mut b=Complex::new(3.0,4.0);
42
43     println!("{:?}",a+b);

```

-
- “+” operatörünü de Complex<T>’lerde kullanabilmek için ayrıca bunu da overload etmemiz gerekiyor. Operatörün adı “AddAssign” diye geçiyor. Bu isimle modülü include etmemiz gerekiyor.
- “-” operatorü içinde “Neg”’i tekrardan “AddAssign” gibi implement etmemiz gerekiyor.

```

38 impl<T> AddAssign for Complex<T>
39     where T:AddAssign<T>
40 {
41     fn add_assign(&mut self,rhs:Self){
42         self.re += rhs.re; ←
43         self.im += rhs.im; ←
44     }
45 }
46
47 impl<T> Neg for Complex<T>
48     where T:Neg<Output = T>
49 {
50     type Output=Complex<T>;
51     fn neg(self) -> Self::Output{
52         Complex{
53             re:-self.re,
54             im:-self.im,
55         }
56     }
57 }

```

-
- Float sayılar için eşitlik'e tam olarak bakılamıyor, virgülden sonraki devamından dolayı. Bu nedenle "partial equality" diye bir kavram var. Float gibi eşitliklere bakabilmek için bunu da overload ile implement etmemiz gerekiyor. Operatorü implement ederken "where T:PartialEq" diye belirtmemizin sebebi, T'nin "=" operatörünü desteklemesi gerektiğini belirtmemizdir.
- Bu arada bu operatörleri implement ederken kullanılan fonksiyonlara modüllerinden ulaşabiliyoruz.
- Partial Equality traitini implement ettiysek, Full Equality trait'ini implement etmeye gerek yok çünkü PartialEquality trait'ini kullanıyor.

```

59 //partial eq
60 //full eq: x=x
61 //NAN=not a number 0/0 inf/inf
62 //NAN==NAN => false
63
64 impl<T> PartialEq for Complex<T>
65     where T:PartialEq
66 {
67     fn eq(&self,rhs:&Self)->bool{//Equal Operator Overloading
68         self.re == rhs.re && self.im == rhs.im
69     }
70     fn ne(&self, rhs:&Self)->bool{//Not Equal Operator Overloading
71         self.re != rhs.re || self.im != rhs.im
72     }
73 }

```

-
- Kendimiz operatörleri implement edebildiğimiz gibi bunları "derive" da edebiliriz. Kimin için kullanacaksak onun üzerinde "Debug"ı nasıl derive ediyorsak o şekilde diğer operatörleri de ekleyebiliriz.

```

8  #[derive(Debug,PartialEq,Eq,Ord,PartialOrd)] //We can use operator by deriving
9  #[derive(Debug)]
10 struct Complex<T>
11 {
12     re:T,
13     im:T
14 }

```

-

Static Dispatch

- Static dispatch, run-time'dan daha önceden compile time'da fonksiyon template'indeki type'ın tanımlanmış belirlenmiş olmasıdır.
- Fonksiyonu template tipinde yazıyoruz fakat compile time'da bu fonksiyon argümanının tipi belirlenmiş haline dönüşür biçimde derleniyor. Bu duruma "monomorphisation" da diyoruz. Polymorphic koddan monomorphic koda dönüşümü ifade eder.

```
8  trait Printable
9  {
10     fn format(&self)->String;
11 }
12 impl Printable for i32{
13     fn format(&self)->String{
14         format!("i32: {}",*self)
15     }
16 }
17 impl Printable for String{
18     fn format(&self)->String{
19         format!("String: {}",*self)
20     }
21 }
22
23 fn SD_print_it<T:Printable>(z:T){
24     println!("{}",z.format());
25 } //monomorphisation
26 //Previous function returns to the below function at compile time when we use with string argument
27 //fn SD_print_it(z:String){...}
28
29 fn staticDispatch(){
30     let a=123;
31     let b="hello".to_string();
32
33     println!("{}",a.format());
34     println!("{}",b.format());
35
36     SD_print_it(a);
37     SD_print_it(b);
38 }
```

Dynamic Dispatch

- Fonksiyon argümanındaki ufak bir değişimle (argüman olarak "&Printable" seçildi) birlikte artık fonksiyon dynamic dispatch olarak görülüp run-time'da fonksiyon argümanları belirleniyor ve ona göre gerekli function tiplerini çağırıyor. Çünkü biz burada a veya b'nin adresini verirken compiler artık pointer a ve b'yi bir trait object olarak yorumluyor.
- Biz "print_it(&a)" dediğimiz zaman compiler'ı a'nın type'ına göre Printable'daki format function'ını String veya i32 için uygun olanını çağırıyor, compiler bu işlemi run-time'da yapıyor.
- Static dispatch'e göre yavaştır, bu nedenle daha az tercih edilir. Run-time'da cost'a neden olur.

```

40 //2-DYNAMIC DISPATCH
41 fn DD_print_it(z:&Printable){
42     println!("{}",z.format());
43 }
44
45 fn dynamicDispatch(){
46     let a=123;
47     let b="hello".to_string();
48
49     DD_print_it(&a);
50     DD_print_it(&b);
51 }

```

-
- Az tercih edilmesi gerekmesine rağmen, bazı durumlarda dynamic dispatch kullanmak zorundayızdır, örneğin "Shape" class'ı kullanarak Circle, Square gibi farklı objelere erişim sağlayabiliriz, yani biz eğer "shape"lerden oluşan bir array tanımlarsak, içerisine de Circle Square gibi farklı objeler yerleştirirsek, bunlara özgü fonksiyonları kullanmak istediğimizde mecbur dynamic dispatch kullanılacaktır.

```

46 struct Circle{ radius:f64 }
47 struct Square { side: f64 }
48
49 trait Shape{
50     fn area(&self)->f64;
51 }
52 impl Shape for Square{
53     fn area(&self)->f64{
54         self.side*self.side
55     }
56 }
57 impl Shape for Circle{
58     fn area(&self)->f64{
59         self.radius*self.radius*std::f64::consts::PI
60     }
61 }
62
63 fn dynamicDispatch(){
64     let a=123;
65     let b="hello".to_string();
66
67     DD_print_it(&a);
68     DD_print_it(&b);
69
70     let shapes:[&Shape;4]= [&Circle{radius:1.0},&Square{side:3.0},&Circle{radius:2.0},&Square{side:4.0}];
71
72     for (i,shape) in shapes.iter().enumerate(){
73         println!("Shape #{} has area {}",i,shape.area());
74     }
75 }

```

Vectors Of Different Objects

- Vector'leri hep aynı tipteki elemanları tutmak için kullandık şimdiye kadar, farklı elemanları tutmak için 2 farklı yol var.
- Birinci yol vector'ün enum'lardan oluşmasıdır. Enum'lara parametre olarak farklı objeleri (struct objeleri) argüman olarak verebiliriz, böylece farklı elemanlardan oluşan bir vector elde ederiz.
- Bunun sıkıntısı bütün vector elemanları için direk aynı fonksiyonu çağırıyoruz, enum type'larına göre ayırt edip fonksiyonlarını çağırabiliriz. Bir de tekrardan enum tanımlamak fazlalık gibi oluyor.

```
42 → enum Creature{
43     Human(Human),
44     Cat(Cat)
45 }
46 fn main(){
47
48     // let mut creatures=Vec::new(); //let mut creatures:Vec<Human>=Vec::new()
49     // creatures.push(Human{name:"John"});
50     // //creatures.push(Cat{name:"Misha"}); //Gives Error
51
52     let mut creatures=Vec::new();
53     creatures.push(Creature::Human(Human{name:"John"}));
54     creatures.push(Creature::Cat(Cat{name:"Misha"}));
55
56     for c in creatures{
57         //c.talk();//We can not write like that, it gives Error, because c is Creature which Enumeration type
58         match c{
59             Creature::Human(h) => h.talk(),
60             Creature::Cat(c) => c.talk()
61         }
62     }
```

- Not: Kodun önceki kısımları "trait" konu başlığındaki trait Animal, struct Human, struct Cat, Cat impl ve Human impl'in aynısıdır.
- Dynamic dispatch'teki gibi bir Animal vectorü oluşturup içerisine hem Human hem Cat koyarız gibi düşünebiliriz, fakat bu sefer de Animal'ın hangisi olacağını bilmediğimiz için içerisindeki objelerin compile-time'da size'ının belirsiz olmasından ötürü compiler hata veriyor ve belirgin bir size istiyor. Bu durumu "Box" keywordü ile çözeceğiz. Human ve Cat objelerimizi Box objelerinin içine yerleştireceğiz. Box'u ise vectore yerleştireceğiz. Box'un size'ı belirli olduğundan compiler hata vermeyecektir. Buradaki tek gereklilik objemizin "trait"inin olması gerektir. Animal objesinin trait'i var olduğu için Box ile kullanabildik.
- Box değişkenlerimizin stack yerine heap'e konmasını sağlayan bir tiptir. Smart pointer olarak da geçer.
- Bu yöntemle birlikte vectorün tüm elemanlarına enum'daki match'e ihtiyaç duymadan erişebildik.

```
64 let mut animals:Vec<Box<Animal>> = Vec::new();
65 animals.push(Box::new(Human{name:"John"}));
66 animals.push(Box::new(Cat{name:"Misha"}));
67
68 for a in animals.iter(){
69     a.talk();
70 }
```


Lifetime and Memory

Ownership

- Rust için en önemli temel olan şey memory safety'dir. Bu bölümde bununla ilgili Rust özelliklerinden bahsedilecektir.
- "let v=vec![1,2,3]" diye kullandığımızda v vektörde ayrılmış memory'e sahip olur, v o memory alanını point eder. Ayrıca variable stack'te tutulurken vektördeki data'lar ise heap'te tutulur. Böyle bir durumda biz "let v2=v" dersek v2 bu sefer o vector alanını point etmeye başlar ve owner v2 olur. Yani Rust'da "shallow copy" yok, "move" var aslında. Pointerını kopyalamıyor direk diğerine taşıyor eğer ki "borrow" etmiyorsak.
- Rust'da aynı lokasyonu gösteren sahiplik bildiren birden fazla pointer olamaz memory safety için, aksi halde "race condition" olma durumu var. Bu nedenle v2 pointer'lık görevini aldıktan sonra v artık iş görmez olur. "v"yi kullanarak vektörü yazdırmak istersek compile hatası alacağız. Aynı durum closure'ın variable'ı vektör'ü sahiplenmesi durumunda da geçerlidir. Closure dışında "v"nin sahipliği kalmamıştır, kullanmaya çalışınca hata verecektir.
- "Box", "Box" ile kurduğumuz variable'larda da pointer görevi görür, bu nedenle onlarda da benzer durumlar geçerlidir. Ama biz normal bir i32 variable'ı kopyalar gibi diğer elemana atmış olsak, bu sefer bu pointer olmadığı için sadece normal temel değişken kopyalama olur, değişkenlerde bir kaybolma olmaz.
- Sonuç olarak eğer heap'teki datayı tutan bir pointer'ımız varsa ve biz bunu başka bir variable'a atarsak/eşitlesek, eski variable'ın pointerlığı kaybolur, işlevsiz hale gelir.

```
3  fn ownership(){
4      let v=vec![1,2,3]; //v owns the memory which is stored in vector
5      //let v2=v; //v2 is owner now.
6      //println!("{:?}",v); //Gives Error, because v is already moved
7
8      let foo = |v:Vec<i32>| ();
9      foo(v);
10     //println!("{:?}",v); //It gives Error because v is already moved
11
12     let t=1; //i32 Copy
13     let y=t; //Only copying a variable
14     println!("t={}",t); //It is OK
15
16     let u=Box::new(1); //i32
17     let u2=u;
18     //println!("u={}",*u); //It gives Error because u is already moved
```

- Eğer closure/lambda function return ediyorsa aşağıdaki örnekteki gibi, o zaman return ettiği vektörü kullanabiliriz, dönenin ownership'liğini tekrar bir variable'la almamız.

```

20 let print_vector = |x:Vec<i32>| -> Vec<i32>
21 {
22     println!("{:?}",x);
23     x
24 };
25
26 let vv=print_vector(v);
27 println!("{}",vv[0]); //It is OK because it returns again and we take it.
28 //println!("{}",v[0]); //It gives Error

```

- Ownership kuralları: 1- Her değerin(value) bir owner vardır. 2-Aynı zamanda sadece bir tane owner olabilir. 3- Owner scope dışında kalırsa, değerimiz düşürülür(dropped).
- Aklımıza bir variable'ı fonksiyona referanssız argüman olarak verince neden ownership'lığı kaybolabilir gibi soru gelebilir, şöyle bir örnek verebiliriz. Bir data'mız var ve biz onu fonksiyona argüman olarak veriyoruz. Biz bu datayı fonksiyona verince bu fonksiyonun içinde local bir değişkene "let v2=v" gibi bu eşitlenir ve move olur ve bu v2 stack'te tutulur. Fonksiyon bitince de scope bittiği için stack'ten v2 atılır. Böylece biz v nesnemizi return edilmezse kaybetmiş oluruz.

Borrowing

- Bir fonksiyona variable'ımızı argüman olarak verirken ownership'lığını eğer kaybetmek istemiyorsak, o zaman "borrowing" yani ödünç olarak vermemiz gerekiyor. Bunun için variable'ımızın adresini yani variable'ımızı reference yaparak o fonksiyona argüman olarak vermeliyiz.
- Dolaylı olarak bu memory işlemlerinin ve mutability'nin amacı database'i korumaktır, yani yanlışlıkla asıl variable'ın değerinin değişmesini engellemektir. Birisinin read ederken diğerinin yazmasını engellemektir.

```

33 fn borrowing(){
34     let v=vec![1,2,3];
35     let print_vector = |x:&Vec<i32>|
36     {
37         println!("{:?}",x);
38         println!("x[0]={}",x[0]);
39     };
40
41     print_vector(&v);
42     println!("{:?}",v);//Ok
43     println!("v[0]={}",v[0]); //Ok

```

- Eğer bir variable'ımızın lokasyonunu mutable olarak borrow/ödünç verdiğimizde o anda orijinal variable'ımızı kullanamayız, borrow'un bitmesi gerekiyor ki kullanabilelim. Bunu normal scope'da gözlemleyebilmek için borrow işleminin başlayıp bitmesi için onu ayrı bir scope'a alırsak "{}" ile. Bu işlem daha güvenilir olması için kullanıcının daha sonra variable'ları karıştırmaması için gerekir.
- Aslında biz bir variable'ı kullanmaya devam etmezsek en son kullanıldığı yerde reference'lı olan variable'ın scopeu bitmiş sayılır, onun yerine başka bir reference variable'ı kullanmaya kalkarsak artık bu variable'ın devri başlamış diyebiliriz. "{}" kullanmadan da yani yapabiliydik.

- Eğer immutable olarak borrow verirsek burada c'ye verdiğimiz gibi, bundan sonra herhangi birine mutable olarak ödünç verdikten sonra c'yi kullanmamıza izin vermeyecektir. Yani read için birden fazla kişiye ödünç verebilirken, write için sadece o anda tek bir kişiye ödünç verebiliriz. Write için ödünç verdiğimizde read için ödünç verdiklerimiz, variable'ı kullanamaz olur. Yani birisi database'e yazma yetkisi aldığı an, okuyucular hep pasif duruma düşecektir taki yazma yetkisi olan yetkisini bırakana kadar. Önceki immutable borrow'lar da geçersiz sayıldı.

```
a=40
g=40
c=40
a=40
b=42
a=42
a=45
```

- ```

45 let mut a=40;
46 let c= &a; //immutable reference
47 let g= &a; //immutable reference
48
49 println!("{}",a);//ok
50 println!("{}",g);//ok
51 println!("{}",c);//ok
52 println!("{}",a);//ok
53 {
54 let b=&mut a; //mutable reference
55 *b += 2;
56 //println!("{}",a); //gives Error because it is borrowed to b now! Because b is used later.
57 //a=45; //gives Error because it is borrowed to b now! Because b is used later.
58 *println!("{}",b); //If we dont use it there, we can use the previous lines for a.
59 }
60 println!("{}",a);
61 a=45;
62 println!("{}",a);
63 //println!("{}",c); //Gives Error, it loses immutable reference after mutable reference

```

- Rust'da safe için for loop'la vector içerisinde normal olarak iterate ederken vector mutable olsa bile içeriğini değiştirmemize izin vermez. Iterate ederken yanlışlıkla içeriğinin değiştirilmesini önlemek içindir.

```

66 let mut z=vec![3,2,1];
67 for i in &z
68 {
69 println!("{}",i);
70 //z.push(5); //it gives Error, when we iterating the vector, it doesnot allow us to change vector inside.
71 }

```

- Reference verme kuralı: 1-Herhangi bir zamanda ortamda ya sadece 1 tane mutable reference olabilir ya da sayısız immutable reference olabilir. Aynı anda hem mutable hem immutable referans olamaz. 2- Reference'lar her zaman geçerli olmalıdır.

## Lifetime

- "&'static str" yazdığımızda " 'static " keywordü lifetime'ı belirtir. Static lifetime'ı olan variable'lar program bitene kadar yaşarlar. " ' + lifetime " keywordü ile variable'ların lifetime'ını belirtiriz.

- Rust geçersiz reference'lara karşı ve kontrollü olduğu için, reference olarak verilen bir variable'ın lifetime'ının da belirtilmesini ister. Böylece ömrü biterse reference'lığını sürdürmez, anlamsız bir lokasyona refer etmez. Örnek olarak bir arrayın tamamına veya bir bölgesine referans olan slice'lar için lifetime'lar anlamlı olacaktır.
- Lifetime'ları kendimizin oluşturduğu şekilde de verebiliriz, hazır lifetime keywordlerini de kullanabiliriz.
- Aslında bazı durumlarda eğer biz bir lifetime belirlemezsek arka planda Rust'ın belirlediği lifetime'lar çalışır, bu nedenle compiler hata verir. Örneğin scope bittikten sonra oradaki variable'lara erişmeye çalışırsak, Rust'ın belirlediği lifetime'ların bitmesinden dolayı compiler erişirmez ve hata verir. Örneğin biz "get\_ref\_name"i normal şekilde tanımladık, ama o arka planda ona uygun "a" ile lifetime'larını belirledi, self kaybolursa return'ün adresi de kaybolur gibi düşünebiliriz.

```

78 struct Person{
79 name:String
80 }
81 impl Person{
82 fn get_ref_name(&self) -> &String{ //Compiler see this function as below
83 //fn get_ref_name<'a>(&'a self) -> &'a String{ //Real function declaration of previous one
84 &self.name
85 }
86 }
87 struct Company<'z>
88 {
89 name:String,
90 ceo:&'z Person //If we only write &Person, compiler gives error when using later
91 }
92 fn lifetime(){
93 let boss = Person {name:String::from("ElonMusk")};
94 let tesla =Company{name:String::from("Tesla"),ceo: &boss};
95
96 let mut z:&String;
97 {
98 let p=Person{name:String::from("John")};
99 z=p.get_ref_name();
100 }
101 //println!("{}",z); //It gives error because p doesnot live now due to real function declaration above
102 }

```

- Struct'ın eğer bir lifetime'ı varsa ona implementation yazdığımız zaman compiler implementation içinde bir lifetime olmasını istiyor. Bu nedenle impl'a da bir lifetime veriz. Genelde convention olarak struct ve impl'in lifetime'ları aynı harfle gösterilir.

```

106 struct Person2<'a>
107 {
108 name:&'a str
109 }
110
111 //impl<'b> Person2<'b>{ //It is OK
112 impl<'a> Person2<'a>{ //Convention: struct and impl have the same key for lifetime.
113 fn talk(&self){
114 println!("Hi my name is {}",self.name);
115 }
116 }
117
118 fn lifetimeInStructureImp(){
119 let person=Person2{name: "Dmitri"};
120 person.talk();
121 }

```

## Reference-Counted Variables(Rc)

- Normalde bir stringi argüman olarak bir fonksiyona verirse, string’de bir çeşit vector olduğu için ownership’ini fonksiyon alacak, variable move olacak ve o variable’a tekrardan fonksiyon dışında erişemeyecektik. Aşağıdaki örnekte de normal kullandığımız gibi kullanılmış ve daha sonra print ile “name”e erişmeye çalışınca hata alınmıştır aynı nedenden dolayı.

```
4 struct Person{
5 name: String,
6 num:i32
7 }
8 impl Person{
9 fn new(name:String,num:i32)->Person{
10 Person{name:name,num:num}
11 }
12 fn greet(&self){
13 println!("Hi, my name is {}",self.name);
14 }
15 }
16 fn rc_demo()
17 {
18 let name="John".to_string();
19 let num=5;
20 let person=Person::new(name,num);
21
22 person.greet();
23 //println!("Name:{}",name);//it gives error,because normal "name" is moved with Person::new function, before.
24 println!("Num:{}",num); //OK
25 }
```

- Bu durumun çözülmesi için bir yöntem geliştirilmiş(Rc), bu sayede variable move olmadan ve memory safety’lerine dikkat ederek kullanabileceğiz.
- Rc’yi kullanabilmek için öncelikle “use std::rc::Rc” isimli modülü eklememiz gerekiyor.
- Rc ile variable’ımızın reference sayısı (strong pointer sayısı) tutulur, kodda variable’ın reference edildiği tüm lokasyonların sayısını tutar. “0” olursa anlar ki variable silinebilir, böylece lifetime üzerinde de kontrol sağlayabiliriz.
- Variable’larımızın “move” olmadan yani erişimini bir yere verip ona erişimimizi kaybetmeden kullanabilmek için Rc’nin “clone” methodunu kullanırız. Clone methodu hem reference count’unu artırır, hem de variable’ın “move” olmadan kullanılabilmesini sağlar.
- “String” type’ındaki önemli olan variableımızı “Rc<String>” olarak yazarak kullanacağız. Böylece Rc methodlarını da kullanabileceğiz o variable üzerinde. Ayrıca yeni string yaratırken “Rc::new()” içinde yaratacağız.
- Reference sayısını öğrenmek için “Rc::strong\_count(&variable\_name)” şeklinde kullanırız, bize variable’ın strong pointer sayısını döner. Aşağıdaki örnekte variable’ı reference eden pointer sayısının scope içinde “clone”dan dolayı bir artıp sonra bir azaldığı görülecektir.
- Rc ve Arc sayesinde hem değişkenimize erişimi kaybetmeyeceğiz yani move olmayacak hem de clone’lansa bile kullanıldığı,referanslandığı sayıyı bilebilmiş olacağız yani onu takip edebilmiş olacağız.

```

Name=John, name has 1 strong pointers
Name=John, name has 2 strong pointers
Hi, my name is John
Name=John, name has 1 strong pointers
Name:John
Num:5

```

```

4 use std::rc::Rc;
5
6 struct Person{
7 name: Rc<String>,
8 num:i32
9 }
10 impl Person{
11 fn new(name:Rc<String>,num:i32)->Person{
12 Person{name:name,num:num}
13 }
14 fn greet(&self){
15 println!("Hi, my name is {}",self.name);
16 }
17 }
18 fn rc_demo()
19 {
20 let name=Rc::new("John".to_string()); //Rc::new(String)
21 let num=5;
22 println!("Name={}, name has {} strong pointers",name,Rc::strong_count(&name));
23 {
24 let person=Person::new(name.clone(),num); //give it with clone
25 println!("Name={}, name has {} strong pointers",name,Rc::strong_count(&name));
26 person.greet();
27 }
28 println!("Name={}, name has {} strong pointers",name,Rc::strong_count(&name));
29 //println!("Name:{}",name);//it gives error,because normal "name" is moved with Person::new function, before.
30 println!("Name:{}",name);//Ok, because we use Rc, we use clone, there is no "move"
31 println!("Num:{}",num); //OK
32 }

```

## Atomic Reference-Counted Variables (Arc)

- “Rc” variable, tek bir thread üzerinde çalışıyor, thread’ler arasında gönderilemez. Eğer birden fazla thread’e variable’ı vereceksek, “Arc” (atomic) kullanmamız gerekiyor. Arc ile yazılmış variable’da birden fazla thread’de kullanılan reference’ler count olarak sayılıyor. Daha safe bir thread kullanımı sunuyor.
- “use std::sync::Arc” ve “use std::thread” ile gerekli modüllerini ekleriz kullanabilmek için.

```

38 struct Person2{
39 name: Arc<String>,
40 }
41 impl Person2{
42 fn new(name:Arc<String>)->Person2{
43 Person2{name:name}
44 }
45 fn greet(&self){
46 println!("Hi, my name is {}",self.name);
47 }
48 }
49 fn arc_demo(){
50 let name=Arc::new("John".to_string());
51 let person=Person2::new(name.clone());
52
53 let t=thread::spawn(move || { //create a thread from its template
54 person.greet();
55 });
56 println!("Name={}",name);//OK.
57 t.join().unwrap();//wait thread to finish
58 }

```

### Using Mutex for Thread Safe Mutability

- Mutable variable'ların aynı anda birden fazla thread'de değiştirilmesini önlemek için mutex'ler kullanılmalıdır.
- Eğer variable'ımızı birden fazla thread'de yalnız okumak için kullanacaksak "Arc" kullanmalı, onu threadler içerisinde değiştireceksek de karışıklıkları önlemek için "Mutex ve Arc" kullanılmalıdır.
- Aşağıdaki örnekte bir thread içinde mutable "state"i değiştirip okuduk, aynı zamanda main thread içinde tekrardan okuduk.
- Mutex'leri "lock" ile kitledik. "Mutex::new(...)" ile mutex halinde variable oluşturduk.

```

Name=John,State=bored
Hi, my name is John and I am excited

```

```

63 struct Person3{
64 name: Arc<String>,
65 state: Arc<Mutex<String>>
66 }
67 impl Person3{
68 fn new(name:Arc<String>,state:Arc<Mutex<String>>)->Person3{
69 Person3{name:name,state:state}
70 }
71 fn greet(&self){
72
73 let mut state=self.state.lock().unwrap();
74 state.clear();
75 state.push_str("excited");
76
77 println!("Hi, my name is {} and I am {}",self.name,state.as_str());
78 }
79 }
80 fn mutex_arc_demo(){
81 let name=Arc::new("John".to_string());
82 let state= Arc::new(Mutex::new("bored".to_string()));
83 let person=Person3::new(name.clone(),state.clone());
84
85 let t=thread::spawn(move || { //create a thread from its template
86 person.greet();
87 });
88 println!("Name={},State={}",name,state.lock().unwrap().as_str());//OK.
89 t.join().unwrap();//wait thread to finish
90 }

```

## Circular References

- 2 objenin birbirine reference etmesidir.
- Buradaki sorunlardan birisi, Immutable olarak bir variable function'a borrow edilmişse, mutable olarak borrow edilmesine compiler izin vermiyor. Bunu "RefCell" ile çözeceğiz.
- Diğer problem ise "drop" edilme sırasına göre birbirine refer eden variable'ları tutan variable'ların boşa düşmesi halindeki compiler hatasıdır, yani kısaca lifetime ile ilgilidir. Bunu da "reference counted variables (Rc)" ile çözeceğiz.
- İlk olarak sorunların çözülmemiş halindeki kodlarını daha sonra çözülmüş halini belirteceğiz.



```

2 struct Student<'a>{ //lifetime is shown with 'a
3 name:String,
4 courses:Vec<&'a Course<'a>>
5 }
6 impl<'a> Student<'a>
7 {
8 fn new(name: &str) ->Student<'a>
9 {
10 Student{
11 name:name.into(), //&str->String
12 courses:Vec::new()
13 }
14 }
15 }
16

```

```

17 struct Course<'a>
18 {
19 name:String,
20 students: Vec<&'a Student<'a>>
21 }
22 impl<'a> Course<'a>
23 {
24 fn new(name: &str)-> Course<'a>{
25 Course{
26 name:name.into(),
27 students:Vec::new()
28 }
29 }
30 fn add_student(&'a mut self,student:&'a mut Student<'a>){
31 student.courses.push(self);
32 self.students.push(student);//it is not allowed for normal case.
33 //RefCell
34 }
35 }
36
37 fn main(){
38 let john=Student::new("John");
39 let course=Course::new("Rust Course");
40 course.add_student(john); //Rd
41 }

```

- 
- Bu bölümde kullanılan kalıplar yeterli olmayacak şekilde açıklamasız kaldı şimdilik, o nedenle ilerleyen süreçte, bilgiler pekiştikçe tekrar bu alana dönmek daha iyi olacak.

```

48 ///SECOND PART WITH SOLVED PROBLEMS
49 // with Rc- RefCell
50 |
51 use std::rc::Rc;
52 use std::cell::RefCell;
53
54 struct Student{
55 name: String,
56 courses:Vec<Rc<RefCell<Course>>>
57 }
58
59 impl Student
60 {
61 fn new(name: &str) ->Student
62 {
63 Student{
64 name:name.into(), //&str->String
65 courses:Vec::new()
66 }
67 }
68 }

```

```

70 struct Course
71 {
72 name:String,
73 students: Vec<Rc<RefCell<Student>>>
74 }
75 impl Course
76 {
77 fn new(name: &str)-> Course{
78 Course{
79 name:name.into(),
80 students:Vec::new()
81 }
82 }
83 fn add_student(course:Rc<RefCell<Course>>,student:Rc<RefCell<Student>>){
84 student.borrow_mut().courses.push(course.clone());
85 course.borrow_mut().students.push(student);
86 }
87 }

```

```

89 fn main(){
90 let john=Rc::new(RefCell::new(Student::new("John")));
91 let jane=Rc::new(RefCell::new(Student::new("Jane")));
92
93 let course=Course::new("Rust Course");
94 let magic_course=Rc::new(RefCell::new(course));
95
96 //course.add_student(john); //Rc
97 Course::add_student(magic_course.clone(),john);//add clone not to move for next line
98 Course::add_student(magic_course,jane);
99 }

```

- Rc ve RefCell ile birlikte birbirini reference eden şeyler için gerekli durumları oluşturabiliriz. Fakat bu durumda compiler'ın bize sağladığı "borrow checker" özelliğini kaybetmiş oluyoruz. Bu nedenle circular reference'sız bir şekilde durumu çözmek daha iyi olacaktır (üçüncü kısımda olduğu gibi).
- Structların içindeki variablelarımızı reference (&) olarak tanımlıyoruz ki "moving" olmasın, aynı reference'i kullanabilsin diye.
- Eğer variable'lar birlikte anlamlı olacaksa onlara lifetime veririz. Yani birisinin yok olduktan sonra diğerinin var olmasının bir anlamı yoksa ikisine aynı lifetime'ı vermemiz gerekir.

```

104 ///THIRD PART- WITH "ENROLLMENT Struct"
105 //students-course-Vec<Enrollment{course,student}>
106
107 struct Student{
108 name:String
109 }
110 impl Student{
111 //shows the courses which is taken by student
112 fn courses(&self,platform:Platform)->Vec<String>{
113 platform.enrollments.iter() //e=enrollment
114 .filter(|&e| e.student.name==self.name)//filter the student
115 .map(|e| e.course.name.clone())//take each course
116 .collect()//put everything in vector
117 }
118 }
119
120 struct Course{
121 name:String
122 }
123 struct Enrollment<'a>{ //Put a lifetime because all together are meaningfull.
124 student:&'a Student, //Put a reference not to "move"
125 course:&'a Course
126 }
127 impl<'a> Enrollment<'a>{
128 fn new(student: &'a Student, course: &'a Course)->Enrollment<'a>{
129 Enrollment(student,course)
130 }
131 }

```

```

133 struct Platform<'a>{ //table of enrollments
134 enrollments: Vec<Enrollment<'a>>
135 }
136 impl<'a> Platform<'a>{
137 fn new()->Platform<'a>{
138 Platform{ enrollments: Vec::new() }
139 }
140 fn enroll(&mut self,student: &'a Student,course: &'a Course){
141 self.enrollments.push(Enrollment::new(student,course));
142 }
143 }
144
145 fn main(){
146 let john=Student{name:"John".into()};
147 let course=Course{name:"Intro to Rust".into()};
148 let mut p=Platform::new();
149
150 p.enroll(&john,&course);
151
152 for c in john.courses(p){
153 println!("John takes {}",c);
154 }
155 }

```

## Concurrency

- Thread, paralel olarak aynı zamanda gerçekleşen görevler bütünüdür.
- “std::thread ve std::time” modüllerinin eklenmesi gerekir.
- Thread yaratmak için “spawn” keywordü kullanılır ve içerisine closure içinde işlemleri verilir.
- “sleep” keywordü ile thread’in beklemesi sağlanır.
- Thread’in işlerinin bitmesini beklemek için “join” keywordü kullanılır. Eğer main thread dışındaki threadlerin bitmesini beklemezsek o zaman main thread bittiği zaman diğer threadler daha işini bitirmeden sonlandırılacaktır. Main thread için herhangi bir beklemeye gerek yoktur, o zaten normal şekilde çalışacaktır. Join yazıp yazmadığımız durumların farkını aşağıda iki sonuçtan anlaşılabilir. Join yazmazsak yazılması gereken “+”lar yazılmadan program bitiyor.

```
5 use std::thread;
6 use std::time;
7
8 fn main(){
9
10 let handle=thread::spawn(|| {
11 for _ in 1..10{
12 print!("+");
13 thread::sleep(time::Duration::from_millis(500));
14 }
15 });
16
17 for _ in 1..10{
18 print!("_");
19 thread::sleep(time::Duration::from_millis(300));
20 }
21
22 handle.join();
23 }
```

```
++_+_+_+
[Done] exited with c
[Running] cd "c:\Use
++_+_+_+
++_+_+_+
```

## Consuming Crates

- “crates.io” sitesinden kominitenin oluşturduğu modüllerden istediğimizi arayarak indirip kullanabiliriz. Tek şart “Cargo.toml” dosyasına bir satır olarak “dependency” eklememiz gerektir.
- Eğer görünürde “Cargo.toml” dosyanız yoksa (örneğin “CodeRunner” extension’ı kullanılınca ayrıca bu dosyayı oluşturmadan derleme yapıyor. Bu durumda bu extension’ı kullanmadan oluşturmak gerekiyor.) bu dosyayı proje dizininde oluşturmak gerekiyor. “package” yazan kısmı standart olarak doldururken “dependency” yazan kısmı üstteki gibi ekleme yapacağımız zaman doldururuz.

```
main.rs Cargo.toml 1
Cargo.toml
1 [package]
2 name="consumingCrates"
3 version="0.1.0"
4 author=["Bugrahan Kara <bugrakara@x.com.tr>"]
5
6 [dependencies]
7 rand= {path="C:/Users/bugrakara/Desktop/Rust_Denemeler/_0_General/Modules/rand", version= "0.8.5"}
```

- Online ortamda “Cargo.toml”i oluşturup dependency’i ekleyip, o dizinde cmd açıp veya VSCode terminalini kullanarak “cargo build” dersek gerekli modülleri indirecektir. Offline ortamda ise gerekli modülleri github üzerinden zip olarak indirdikten sonra dependency’de path’leri verilip, o modülleri build edip “cargo build --offline” denilerek bu durum çözülebilir. \*Denendiğinde birkaç farklı hata ile karşılaşıldı, bu nedenle ilerleyen süreçte tekrar detaylı incelenip denenecektir.

Detaylı bilgi için:

<https://stackoverflow.com/questions/57336771/how-do-i-use-cargo-to-build-a-project-while-offline>

- “Bir diğer yol da “cargo-vendor” kullanımı, offline olarak tüm crate’leri modülleri içerip kullanılmasına olanak tanır.
- Dependency verdiğimiz “random” crate’ini kullanmak için “extern crate rand” diye belirtiriz.

```
main.rs
1 extern crate rand;
2 use rand::Rng;
3
4 fn main(){
5 let mut rng=rand::thread_rng();
6 let b:bool =rng.gen();
7 }
```

## Building Modules and Crates

- “Crates” modülleri içeren yardımcı library gibi dökümanlardır. “Module”ler ise birbirleriyle ilişkili işlevleri, fonksiyonları barındıran kümeler gruplardır. “mod” olarak gösterilirler, içerisinde birden fazla direk kullanabileceğimiz functionlar bulunur.
- Yeni bir crate (ismi “phrases” olsun.) oluşturup, bunu kendi projemizde kullanmak amacıyla oluşturacağımız bir klasör içinde (ismi “Phrases” olsun) .rs dosyasını (ismi “lib.rs” olsun şimdilik) açarız. İçerisinde “mod”lardan oluşan modüllerimizi yazarız. Burada farklı dillerde selamlama için modüller ekledik. Bu fonksiyon ve modüllere dışarıdan ulaşmak isteyeceğimiz için bunları public yani “pub” yaptık.

```
Phrases > src > lib.rs
1 pub mod greetings
2 {
3 pub mod english{
4 pub fn hello() -> String{"hello".to_string()}
5 pub fn goodbye() -> String{"goodbye".to_string()}
6 }
7 pub mod turkish{
8 pub fn hello() -> String{"selam".to_string()}
9 pub fn goodbye() -> String{"gulegule".to_string()}
10 }
11 }
```

- Eğer düzenli olsun istersek src içinde lib.rs'in yanında modülleri farklı .rs dosyaları içerisinde bölebiliriz. Bunun için ana modülün ismi ile "src" içinde bir klasör("greetings") oluştururuz. Bunun içerisinde ise submodule'ümüzün isminde bir .rs dosyası oluştururuz("english.rs"). Daha sonra onları lib.rs içerisinde dosya ismi ile çağırırız.

```
Phrases > src > english.rs
1 pub fn hello() -> String{"hello".to_string()}
2 pub fn goodbye() -> String{"goodbye".to_string()}
```

```
Phrases > src > lib.rs
1 pub mod greetings
2 {
3 pub mod english;
4 pub mod turkish{
5 pub fn hello() -> String{"selam".to_string()}
6 pub fn goodbye() -> String{"gulegule".to_string()}
7 }
8 }
```

- Crate için ayrıca basit bir "Cargo.toml" dosyası oluştururuz, içeri doldururuz. Ayrıca Crate'mizin klasörü içinde bir "src" isimli bir klasör daha oluşturur, ana klasördeki lib.rs'i ve diğerlerini bunun içine atarız. Daha sonra ana klasörümüzün path'inde "cargo build" diyerek crate'imizi "Cargo" aracılığı ile kullanıma hazır hale getiririz.

```
Phrases > Cargo.toml
1 [package]
2 name="phrases"
3 version="0.1.0"
4 authors=["Bugrahan Kara <bugrakara@x.com.tr>"]
```

- Daha sonra kendi projemizi oluştururuz, onun Cargo.toml dosyasına crate'imizi dependency olarak pathini veririz ve main.rs'imiz içinde crate'imizi extern ederiz. Çalıştırdıktan sonra oluşturduğumuz crate'i kullandığını görebiliriz.

```

Cargo.toml
1 [package]
2 name = "_59_60_BuildingModulesCrates_Testing"
3 version = "0.1.0"
4 edition = "2021"
5
6 # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
7
8 [dependencies]
9 phrases={path="./Phrases"}
10
11 [[bin]]
12 name = "_59_60_BuildingModulesCrates_Testing"
13 path = "main.rs"
14

```

- “use” kullanarak modüllerin isimlerini uzun uzun yazmak yerine kısaca gösterebiliriz.

```

main.rs
1 extern crate phrases;
2
3 use phrases::greetings::turkish;
4
5 fn main(){
6 println!("English: {}, {}",phrases::greetings::english::hello(),phrases::greetings::english::goodbye());
7 println!("Turkish: {}, {}",turkish::hello(),turkish::goodbye());
8 }

```

- Not: Bunu VSCode üzerinden kullanacaksanız “CodeRunner” extension’ını kapatıp ilk bahsedilen çalıştırma yöntemi ile denemelisiniz.

## Testing

- Rust “unit-test”i desteklemektedir. Bu nedenle bunu kolaylaştıracak araçlar sunmuştur.
- Bu örnekte yazdığımız crate’i lib.rs’i test edeceğiz. Bu nedenle lib.rs içerisinde test için gerekli fonksiyonumuzu çağıracağız.
- Attribute (Rust Macro’su) olarak “[test]” yazarak test fonksiyonu olduğunu belirtiriyoruz. “assert\_eq!(...)” keywordünü kullanarak test kontrolünü yaparız. Test fonksiyonlarını yazdıktan sonra terminalden lib.rs’in bulunduğu “Phrases” klasörü adresindeyken “cargo test” diyerek unit-test’leri çağırıyoruz cargo aracılığıyla, bize bu aşağıdaki şekilde dönüyor.

```

Finished test [unoptimized + debuginfo] target(s) in 0.31s
Running unittests (target\debug\deps\phrases-7bd0ee3eb27c245f.exe)

running 1 test
test english_greeting_correct ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Doc-tests phrases

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

```

```
Phrases > src > lib.rs
1 pub mod greetings
2 {
3 pub mod english;
4 pub mod turkish{
5 pub fn hello() -> String{"selam".to_string()}
6 pub fn goodbye() -> String{"gulegule".to_string()}
7 }
8 }
9
10 #[test]
11 fn english_greeting_correct() {
12 assert_eq!("hello", greetings::english::hello());
13 }
```

- Eğer test'in fail olacağını biliyorsak ve bunu test edeceksek, "[should\_panic]" diyerek attribute'da bunu belirtiriz. Ona göre testin geçip geçmediğini ayarlar.

```
10 #[test]
11 #[should_panic]
12 fn english_greeting_correct() {
13 assert_eq!("hellodsf", greetings::english::hello());
14 }

running 1 test
test english_greeting_correct - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 me
```

- Eğer spesifik olarak o testin yapılmamasını istiyorsak "[ignore]" attribute'ı yazılır.

```
running 1 test
test english_greeting_correct ... ignored

test result: ok. 0 passed; 0 failed; 1 ignored;
```

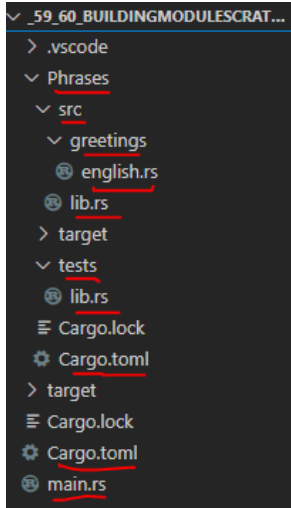
- Eğer testlerimizi ayrı bir dosya içerisine yazmak istersek, önce "Phrases" klasörünün içine "tests" isimli bir klasör oluştururuz. Bunun içerisine de gene "lib.rs" isimli bir dosya oluşturduk. Test dosyamızı da aslında bir modül haline getirmeye çalışıyoruz gibi düşünebiliriz. Daha sonra test satırlarımızı bu dosya içine yapıştırıp bunları bir modüle içine aldık. Dışarıdan "phrases" crate'ine yani test edeceğimiz unit'i de extern yapıp ulaşıyoruz. Son olarak da "[cfg(test)]" attribute'ını ekleyerek test modülü olduğunu belirtiriz.

```
Phrases > tests > lib.rs
1 #[cfg(test)]
2 mod tests{
3
4 extern crate phrases;
5
6 #[test]
7 #[should_panic]
8 #[ignore]
9 fn english_greeting_correct() {
10 assert_eq!("hellodsf", phrases::greetings::english::hello());
11 }
12 }
```

- "Phrases" klasör uzantısında "cargo build" ve "cargo test" deyip çalıştırırız.

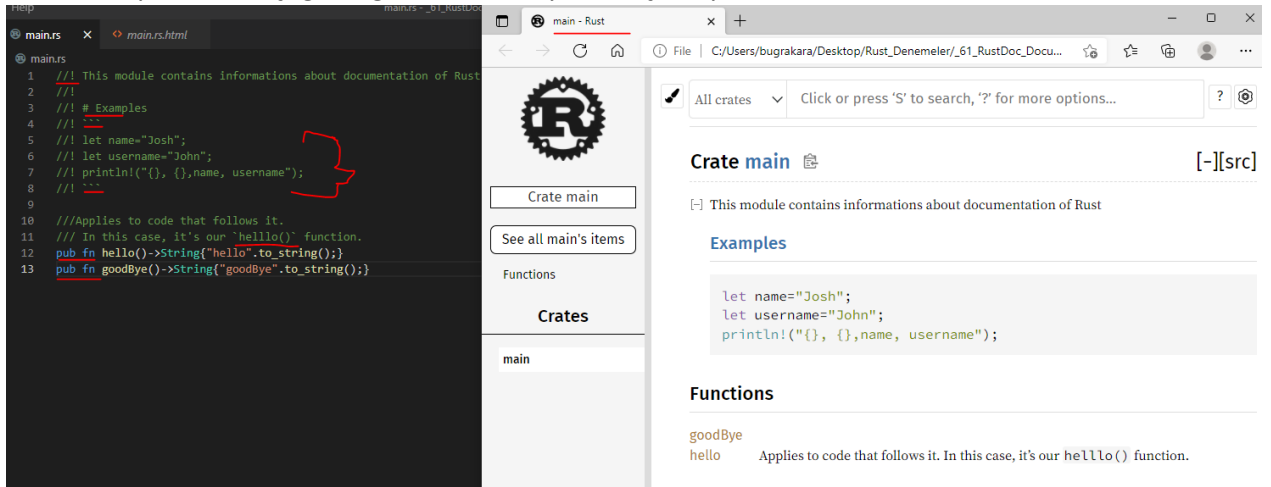
```
PS C:\Users\bugrakara\Desktop\Rust_Denemeler_59_60_BuildingModulesCrates_Testing\Phrases> cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
PS C:\Users\bugrakara\Desktop\Rust_Denemeler_59_60_BuildingModulesCrates_Testing\Phrases> cargo test
```

- Bu örnekteki Klasör ve dosyaların ağaç yapısı aşağıdaki şekildedir.



## RustDoc – Documentation

- Rust doküman yapmak için bazı araçlar sunar. Bunlardan temeli “rustdoc”tur.
- Yorum satırına alırken bazı farklılıklar vardır. Örneğin “//!”, “/\*!”, “/\*”, buna göre doküman farklı şekilde oluşturulur. Dokümanı yazarken bu şekilde yorum satırında belirttiğimiz takdirde, “rustdoc filename.rs” diye terminale yazılınca, “index.html” diye bir dosya oluşturuyor. Bu dosya internet sayfasındaki aşağıdaki gibi bir html sayfası oluşturuyor.



```
PS C:\Users\bugrakara\Desktop\Rust_Denemeler_61_RustDoc_Documentation> rustdoc main.rs
PS C:\Users\bugrakara\Desktop\Rust_Denemeler_61_RustDoc_Documentation> cd doc
PS C:\Users\bugrakara\Desktop\Rust_Denemeler_61_RustDoc_Documentation\doc> cd main
PS C:\Users\bugrakara\Desktop\Rust_Denemeler_61_RustDoc_Documentation\doc\main> .\index.html
```