Rust 치트시트

1. 변수, 스칼라 & 컴파운드 타입

- 변수: `let x = 5;` (불변), `let mut y = 10;` (가변).
- 섀도잉: `let x = x + 1;` 새로운 변수를 선언하여 이전 변수를 가림.
- 상수: `const MAX_POINTS: u32 = 100_000;` 타입 명시 필수.
- 스칼라 타입:
  - 정수: `i8`..`i128`, `u8`..`u128`, `isize`, `usize`.
  - 부동소수점: `f32`, `f64`.
  - 불리언: `bool` (`true`, `false`).
  - 문자: `char` (4바이트 유니코드).
- 컴파운드 타입:
  - 튜플: `let tup: (i32, f64, u8) = (500, 6.4, 1);`, `tup.0`으로 접근.
  - 배열: `let a: [i32; 5] = [1, 2, 3, 4, 5];` 고정 크기, 스택 할당.

2. 제어 흐름

- if-else: `if ... else if ... else { ... }`. `if`는 표현식이므로 `let x = if c { 5 } else { 6 };` 가능.
- 루프:
  - `loop { ... }`: 무한 루프. `break`로 값 반환 가능: `let result = loop { break 10; };`.
  - `while condition { ... }`
  - `for element in collection { ... }`: `for x in 0..10 { ... }`

3. 소유권, 참조, 슬라이스

- 소유권 규칙:
  1. 모든 값은 소유자(owner) 변수를 가짐.
  2. 한 번에 단 한 명의 소유자만 있음.
  3. 소유자가 스코프를 벗어나면 값은 `drop`됨.
- 이동 (Move): 스택에만 있는 데이터(e.g., `i32`)는 복사, 힙 데이터(e.g., `String`)는 소유권 이동.
- 클론 (Clone): `let s2 = s1.clone();` 힙 데이터의 깊은 복사.
- 참조와 대여 (References & Borrowing):
  - `&T`: 불변 참조. 여러 개 존재 가능.
  - `&mut T`: 가변 참조. 오직 하나만 존재 가능.
  - 불변 참조와 가변 참조는 공존할 수 없음.
  - `let r1 = &s; let r2 = &s;` (O)
  - `let r1 = &mut s; let r2 = &mut s;` (X)
- 슬라이스 (Slices): 컬렉션의 일부를 소유권 없이 참조. `&str`, `&[i32]`.

4. 구조체 (Structs)

- 정의: `struct User { username: String, email: String, active: bool }`
- 인스턴스화: `let user1 = User { email: String::from("..."), ... };`
- 튜플 구조체: `struct Color(i32, i32, i32);`
- 유닛 라이크 구조체: `struct AlwaysEqual;`
- 메서드: `impl User { fn new(...) -> User { ... } fn describe(&self) { ... } }`
  - `&self`: 불변으로 빌림, `&mut self`: 가변으로 빌림, `self`: 소유권 가져옴.
- 연관 함수 (Associated Functions): `impl` 블록 내에 있지만 `self`를 받지 않는 함수. `String::from`처럼 `::`으로 호출.

5. 열거형 (Enums) 및 패턴 매칭

- 정의: `enum Message { Quit, Move { x: i32, y: i32 }, Write(String), ChangeColor(i32, i32, i32), }`
- Option: `enum Option<T> { Some(T), None, }` 값이 있거나 없음을 표현.
- match: 모든 경우를 처리해야 하는 강력한 제어 흐름 연산자.

```
match option_value {
    Some(i) if i > 5 => println!("Greater than 5"), // 매치 가드
    Some(i) => println!("Got an int: {}", i),
    None => println!("It's None"),
}
```

- if let: 하나의 패턴만 매칭할 때 사용. `if let Some(value) = option_value { ... }`
- while let: 루프가 패턴에 맞는 동안 계속 실행.

6. 패키지, 크레이트, 모듈

- 크레이트 (Crate): 라이브러리(`lib.rs`) 또는 실행 파일(`main.rs`).
- 패키지 (Package): 하나 이상의 크레이트를 포함. `Cargo.toml`로 관리.
- 모듈 (Module): `mod` 키워드로 코드를 그룹화. `super`로 부모 모듈 참조.
- 경로 (Path):
  - `use std::collections::HashMap;`: 절대 경로.
  - `use self::kinds::PrimaryColor;`: 상대 경로.
  - `use std::io::{self, Write};`: 중첩 경로.
  - `use std::collections::*;`: Glob 연산자.

7. 주요 컬렉션

- 벡터 (Vector): `Vec<T>`. `let v: Vec<i32> = Vec::new(); let v = vec![1, 2, 3];`
- 문자열 (String): `String`. UTF-8 인코딩, 힙 할당.
- 해시맵 (HashMap): `HashMap<K, V>`. `use std::collections::HashMap;`

8. 에러 처리

- `panic!`: 복구 불가능한 에러. 프로그램을 즉시 종료.
- `Result<T, E>`: 복구 가능한 에러. `enum Result<T, E> { Ok(T), Err(E), }`
- `?` 연산자: `Result`를 반환하는 함수 내에서 에러 전파를 단순화. `Err`일 경우 즉시 반환.

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

9. 제네릭, 트레잇, 라이프타임

- 제네릭 (Generics): `fn largest<T: PartialOrd + Copy>(list: &[T]) -> T { ... }`
- 트레잇 (Traits): 특정 타입이 가질 수 있는 공유 동작을 정의. (인터페이스와 유사)

```
pub trait Summary {
    fn summarize(&self) -> String;
}
impl Summary for NewsArticle { ... }
```

- 라이프타임 (Lifetimes): 참조가 유효한 스코프를 명시하여 댕글링 포인터를 방지. `fn longest<'a>(x: &'a str, y: &'a str) -> &'a str { ... }`

10. 스마트 포인터

- `Box<T>`: 힙에 값을 할당.
- `Rc<T>`: 다중 소유권을 가능하게 하는 참조 카운팅 스마트 포인터. (단일 스레드)
- `Arc<T>`: `Rc<T>`의 원자적(스레드 안전) 버전.
- `RefCell<T>`/`Mutex<T>`: 내부 가변성(interior mutability) 패턴을 제공. 런타임에 빌림 규칙을 검사.

11. 고급 Rust 개념

제네릭과 트레잇 바운드 심화

```
// 트레잇 바운드
fn process<T: Clone + Debug>(item: T) -> T {
    println!("{:?}", item);
    item.clone()
}

// where 절 사용
fn complex_function<T, U>(t: T, u: U) -> String
where
    T: Display + Clone,
    U: Debug + PartialEq,
{
    format!("{} {:?}", t, u)
}

// 트레잇 객체
trait Drawable {
    fn draw(&self);
}

struct Circle { radius: f64 }
struct Rectangle { width: f64, height: f64 }

impl Drawable for Circle {
    fn draw(&self) {
        println!("Drawing circle with radius {}", self.radius);
    }
}

impl Drawable for Rectangle {
    fn draw(&self) {
        println!("Drawing rectangle {}x{}", self.width, self.height);
    }
}

// 트레잇 객체 사용
```

```
fn draw_shapes(shapes: &[Box<dyn
Drawable>]) {
    for shape in shapes {
        shape.draw();
    }
}
```

라이프타임 심화

```
// 명시적 라이프타임
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }

    fn announce_and_return_part(&self,
announcement: &str) -> &str {
        println!("Attention please: {}",
announcement);
        self.part
    }
}

// 라이프타임 엘리전 규칙
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in
bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}

// 고차 라이프타임 바운드
fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: std::fmt::Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

매크로 시스템

```
// 선언적 매크로
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec =
Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}

// 사용: let v = vec![1, 2, 3];

// 프로시저 매크로 (derive 매크로 예제)
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input,
DeriveInput};

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input:
TokenStream) -> TokenStream {
    let ast = parse_macro_input!(input
as DeriveInput);
    let name = &ast.ident;

    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro!
My name is {}!", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

비동기 프로그래밍 (async/await)

```
use tokio::time::{sleep, Duration};

// 비동기 함수
async fn fetch_data() -> String {
    sleep(Duration::from_secs(1)).await;
    "Data fetched".to_string()
}

async fn process_data(data: String) ->
String {
    sleep(Duration::from_secs(1)).await;
    format!("Processed: {}", data)
}

// 비동기 함수 조합
async fn fetch_and_process() -> String {
    let data = fetch_data().await;
    process_data(data).await
}

// 병렬 실행
async fn parallel_execution() ->
(String, String) {
    let (result1, result2) =
tokio::join!(
        fetch_data(),
        fetch_data()
    );
    (result1, result2)
}

// 비동기 스트림
use futures::stream::{self, StreamExt};

async fn process_stream() {
    let mut stream = stream::iter(1..=5)
        .map(|i| async move { i * 2 })
        .buffer_unordered(3);

    while let Some(result) =
stream.next().await {
        println!("Result: {}", result);
    }
}
```

에러 처리 고급 패턴

```
use std::error::Error;
use std::fmt;

// 커스텀 에러 타입
#[derive(Debug)]
enum MyError {
    Io(std::io::Error),
    Parse(std::num::ParseIntError),
    Custom(String),
}

impl fmt::Display for MyError {
    fn fmt(&self, f: &mut
fmt::Formatter) -> fmt::Result {
        match self {
            MyError::Io(err) => write!
(f, "IO error: {}", err),
            MyError::Parse(err) =>
write!(f, "Parse error: {}", err),
            MyError::Custom(msg) =>
write!(f, "Custom error: {}", msg),
        }
    }
}

impl Error for MyError {}

impl From<std::io::Error> for MyError {
    fn from(err: std::io::Error) ->
MyError {
        MyError::Io(err)
    }
}

impl From<std::num::ParseIntError> for
MyError {
    fn from(err:
std::num::ParseIntError) -> MyError {
        MyError::Parse(err)
    }
}

// 에러 체이닝
fn read_and_parse_file(filename: &str) -
> Result<i32, MyError> {
    let contents =
std::fs::read_to_string(filename)?;
    let number =
contents.trim().parse::<i32>()?;
    Ok(number)
}

// anyhow 크레이트 사용
```

```rust
use anyhow::{Context, Result};

fn read_config() -> Result<String> {
    let config =
std::fs::read_to_string("config.toml")
        .context("Failed to read config
file")?;
    Ok(config)
}
```

함수형 프로그래밍 패턴

```rust
// 클로저와 고차 함수
fn apply_twice<F>(f: F, x: i32) -> i32
where
    F: Fn(i32) -> i32,
{
    f(f(x))
}

// 함수 조합
fn compose<F, G>(f: F, g: G) -> impl
Fn(i32) -> i32
where
    F: Fn(i32) -> i32,
    G: Fn(i32) -> i32,
{
    move |x| f(g(x))
}

// 이터레이터 체이닝
fn process_numbers(numbers: Vec<i32>) ->
Vec<i32> {
    numbers
        .into_iter()
        .filter(|&x| x > 0)
        .map(|x| x * 2)
        .collect()
}

// 커링
fn add(x: i32) -> impl Fn(i32) -> i32 {
    move |y| x + y
}

// 사용: let add_five = add(5); let
result = add_five(3);
```

메모리 안전성과 성능

Zero-Cost Abstractions

```rust
// 컴파일 타임에 최적화되는 추상화
trait Drawable {
    fn draw(&self);
}

struct Circle { radius: f64 }
struct Rectangle { width: f64, height:
f64 }

impl Drawable for Circle {
    fn draw(&self) {
        println!("Circle: {}",
self.radius);
    }
}

impl Drawable for Rectangle {
    fn draw(&self) {
        println!("Rectangle: {}x{}",
self.width, self.height);
    }
}

// 모노모피즘: 각 타입에 대해 별도의 함수 생성
fn draw_all<T: Drawable>(items: &[T]) {
    for item in items {
        item.draw();
    }
}
```

메모리 풀 패턴

```rust
use std::collections::VecDeque;

struct MemoryPool<T> {
    pool: VecDeque<T>,
    factory: fn() -> T,
}

impl<T> MemoryPool<T> {
    fn new(factory: fn() -> T) -> Self {
        Self {
            pool: VecDeque::new(),
            factory,
        }
    }

    fn get(&mut self) -> T {

self.pool.pop_front().unwrap_or_else(self.factory)
    }

    fn return_item(&mut self, item: T) {
        self.pool.push_back(item);
    }
}
```

웹 개발 (Warp 웹 프레임워크)

```rust
use warp::Filter;
use std::collections::HashMap;
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize)]
struct User {
    id: u32,
    name: String,
    email: String,
}

// 라우트 정의
fn user_routes() -> impl Filter<Extract
= impl warp::Reply, Error =
warp::Rejection> + Clone {
    let users = warp::path("users");

    let get_users = users
        .and(warp::get())
        .and_then(get_users_handler);

    let create_user = users
        .and(warp::post())
        .and(warp::body::json())
        .and_then(create_user_handler);

    get_users.or(create_user)
}

async fn get_users_handler() ->
Result<impl warp::Reply,
warp::Rejection> {
    let users = vec![
        User { id: 1, name:
"Alice".to_string(), email:
"alice@example.com".to_string() },
        User { id: 2, name:
"Bob".to_string(), email:
"bob@example.com".to_string() },
    ];
    Ok(warp::reply::json(&users))
}

async fn create_user_handler(user: User)
-> Result<impl warp::Reply,
warp::Rejection> {
    // 사용자 생성 로직
    Ok(warp::reply::json(&user))
}

#[tokio::main]
async fn main() {
    let routes = user_routes();
    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

시스템 프로그래밍

```rust
use std::fs::File;
use std::io::{self, Read, Write};
use std::os::unix::io::{AsRawFd, RawFd};

// 파일 디스크립터 직접 조작
fn duplicate_fd(fd: RawFd) ->
io::Result<RawFd> {
    unsafe {
        let new_fd = libc::dup(fd);
        if new_fd == -1 {

Err(io::Error::last_os_error())
        } else {
            Ok(new_fd)
        }
    }
}

// 메모리 매핑
use memmap2::{Mmap, MmapOptions};

fn memory_map_file(filename: &str) ->
io::Result<Mmap> {
    let file = File::open(filename)?;
    unsafe
{ MmapOptions::new().map(&file) }
}
```

```rust
// 시그널 처리
use signal_hook::{iterator::Signals,
SIGINT, SIGTERM};

fn setup_signal_handlers() ->
io::Result<()> {
    let mut signals =
Signals::new(&[SIGINT, SIGTERM])?;

    std::thread::spawn(move || {
        for sig in signals.forever() {
            match sig {
                SIGINT => println!
("Received SIGINT"),
                SIGTERM => println!
("Received SIGTERM"),
                _ => unreachable!(),
            }
        }
    });

    Ok(())
}
```

테스팅과 벤치마킹

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_calculator() {
        assert_eq!(add(2, 3), 5);
        assert_eq!(multiply(4, 5), 20);
    }

    #[test]
    #[should_panic(expected = "division
by zero")]
    fn test_division_by_zero() {
        divide(10, 0);
    }

    #[test]
    fn test_with_result() -> Result<(),
String> {
        if add(2, 2) == 4 {
            Ok(())
        } else {
            Err(String::from("2 + 2
should equal 4"))
        }
    }
}

// 벤치마킹
use criterion::{criterion_group,
criterion_main, Criterion};

fn benchmark_fibonacci(c: &mut
Criterion) {
    c.bench_function("fibonacci 20", |b|
b.iter(|| fibonacci(20)));
}

criterion_group!(benches,
benchmark_fibonacci);
criterion_main!(benches);
```

패키지 관리와 Cargo

```toml
# Cargo.toml 예제
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"
authors = ["Your Name
<your.email@example.com>"]
description = "A sample Rust project"
license = "MIT"

[dependencies]
tokio = { version = "1.0", features =
["full"] }
serde = { version = "1.0", features =
["derive"] }
warp = "0.3"
anyhow = "1.0"

[dev-dependencies]
criterion = "0.4"

[profile.release]
opt-level = 3
lto = true
codegen-units = 1
panic = "abort"
```

크로스 컴파일

```
# Rust 크로스 컴파일 설정
rustup target add x86_64-unknown-linux-
gnu
rustup target add aarch64-unknown-linux-
gnu

# 크로스 컴파일러 설치
sudo apt-get install gcc-x86-64-linux-
gnu
sudo apt-get install gcc-aarch64-linux-
gnu

# 크로스 컴파일 실행
cargo build --target x86_64-unknown-
linux-gnu --release
```