Rust 치트시트

변수 및 기본 데이터 타입

- **변수 선언**: `let x = 5;` (기본적으로 불변), `let mut y = 10;` (가변 변수 선언 시 `mut` 사용).
- **섀도잉 (Shadowing)**: `let x = x + 1;`과 같이 같은 이름의 변수를 다시 선언하여 이전 변수를 가릴 수 있습니다.
- **상수**: `const MAX_POINTS: u32 = 100_000;` 상수는 항상 타입을 명시해야 합니다.
- **스칼라 타입 (Scalar Types)**:
  - **정수**: `i8`..`i128` (부호 있음), `u8`..`u128` (부호 없음), `isize`, `usize` (포인터 크기).
  - **부동 소수점**: `f32`, `f64` (기본값).
  - **불리언**: `bool` (`true`, `false`).
  - **문자**: `char` (4바이트 유니코드 포인트).
- **컴파운드 타입 (Compound Types)**:
  - **튜플 (Tuple)**: `let tup: (i32, f64, u8) = (500, 6.4, 1);`, 인덱스(`tup.0`)로 접근합니다.
  - **배열 (Array)**: `let a: [i32; 5] = [1, 2, 3, 4, 5];` 고정 크기를 가지며 스택에 할당됩니다.

제어 흐름 (Control Flow)

- **조건문 (if-else)**: `if` … `else if` … `else { ... }`. `if`는 표현식이므로 `let x = if c { 5 } else { 6 };`와 같은 할당이 가능합니다.
- **반복문 (Loops)**:
  - `loop { ... }`: 무한 루프를 실행하며, `break` 뒤에 값을 적어 루프 외부로 반환할 수 있습니다.
  - `while condition { ... }`: 조건이 참인 동안 반복합니다.
  - `for element in collection { ... }`: 컬렉션의 요소를 순회합니다 (예: `for x in 0..10 { ... }`).

소유권(Ownership), 참조 및 슬라이스

- **소유권 규칙**:
  1. 모든 값은 소유자(Owner) 변수를 가집니다.
  2. 한 번에 단 하나의 소유자만 존재할 수 있습니다.
  3. 소유자가 자신이 정의된 스코프를 벗어나면 값은 자동으로 제거(`drop`)됩니다.
- **이동 (Move)**: 스택에 저장되는 기본 타입 데이터는 복사되지만, `String`과 같은 힙 데이터는 소유권이 이전됩니다.
- **클론 (Clone)**: `let s2 = s1.clone();`을 통해 힙 데이터의 깊은 복사를 수행할 수 있습니다.
- **참조와 대여 (References & Borrowing)**:
  - `&T`: 불변 참조입니다. 여러 개가 동시에 존재할 수 있습니다.
  - `&mut T`: 가변 참조입니다. 특정 시점에 단 하나만 존재할 수 있습니다.
  - 불변 참조가 존재하는 동안에는 가변 참조를 생성할 수 없습니다.
- **슬라이스 (Slices)**: 컬렉션 전체가 아닌 일부분을 소유권 없이 참조할 때 사용합니다. (`&str`, `&[i32]`)

구조체 (Structures)

- **정의**: `struct User { username: String, email: String, active: bool }`
- **인스턴스 생성**: `let user1 = User { email: String::from("..."), ... };`
- **튜플 구조체**: `struct Color(i32, i32, i32);` (필드 이름 없이 타입만 나열)
- **유닛 라이크 구조체**: `struct AlwaysEqual;` (필드가 없는 구조체)
- **메서드 정의**: `impl User { ... }` 블록 내에 정의합니다.
  - `&self`: 불변 대여, `&mut self`: 가변 대여, `self`: 소유권 획득.
- **연관 함수 (Associated Functions)**: `self`를 매개변수로 받지 않는 함수로, `Type::function()` 형식으로 호출합니다.

열거형 (Enums) 및 패턴 매칭

- **정의**: `enum Message { Quit, Move { x: i32, y: i32 }, Write(String), ... }`
- `Option<T>`: 값이 있거나(`Some(T)`) 없는(`None`) 상태를 안전하게 처리하기 위한 표준 열거형입니다.
- `match`: 모든 가능한 경우를 엄격하게 검사하는 패턴 매칭 구문입니다.

```
match option_value {
    Some(i) if i > 5 ⇒ println!("5보다 큰 값"), // 매치 가드 활용
    Some(i) ⇒ println!("값 발견: {}", i),
    None ⇒ println!("값 없음"),
}
```

- `if let` / `while let`: 특정 패턴 하나에만 집중하여 코드를 간결하게 작성할 때 사용합니다.

모듈 시스템 (Packages, Crates, Modules)

- **크레이트 (Crate)**: 컴파일의 최소 단위로, 라이브러리나 실행 파일을 생성합니다.
- **패키지 (Package)**: 하나 이상의 크레이트를 관리하며 `Cargo.toml` 설정을 공유합니다.
- **모듈 (Module)**: `mod` 키워드를 사용하여 코드를 논리적으로 그룹화하고 가시성을 제어합니다.
- **경로 지정 (Path)**: `use` 키워드를 사용하여 외부 패키지나 모듈의 아이템을 가져옵니다.

주요 컬렉션 (Collections)

- **벡터 (Vector)**: `Vec<T>`. 크기 조절이 가능한 동적 배열입니다.
- **문자열 (String)**: `String`. UTF-8 형식의 동적 문자열로 힙 메모리에 할당됩니다.
- **해시맵 (HashMap)**: `HashMap`. 키-값(Key-Value) 쌍을 저장하는 연관 컨테이너입니다.
- **슬라이스 (Slice)**: `&[T]`. 컬렉션의 연속된 부분을 참조합니다.

타입 변환 (Type Conversions)

- `as` **키워드**: `x as u64`. 숫자 타입 간 변환이나 포인터 캐스팅에 사용합니다.
- `From` & `Into`: `impl From<A> for B`. `B::from(a)` 또는 `a.into()`로 변환 (손실 없음).
- `TryFrom` & `TryInto`: 실패 가능성이 있는 변환. `Result`를 반환합니다.
- `AsRef` & `AsMut`: 타입 `A`를 참조 `&B` 또는 `&mut B`로 저렴하게 변환합니다.
- `Deref`: `*x` 연산 시 동작 정의. 스마트 포인터가 실제 데이터처럼 동작하게 합니다.
- `String` **변환**: `s.parse::<i32>()` (문자열 → 숫자), `x.to_string()` (숫자 → 문자열).

이터레이터 (Iterators)

- **생성**:
  - `iter()`: 불변 참조 (`&T`) 순회.
  - `iter_mut()`: 가변 참조 (`&mut T`) 순회.
  - `into_iter()`: 소유권 이동 (`T`) 또는 적절한 타입 순회.
- **어댑터 (Lazy)**:
  - `map(|x| ...)`: 각 요소 변환.
  - `filter(|x| ...)`: 조건에 맞는 요소만 선택.
  - `enumerate()`: 인덱스와 함께 반환 `(index, value)`.
  - `zip(other)`: 두 이터레이터를 하나로 묶음.
  - `take(n)`, `skip(n)`: 개수 제한 및 건너뛰기.
- **소비 (Consumer)**:
  - `collect()`: 이터레이터를 컬렉션(Vec 등)으로 변환.
  - `fold(init, |acc, x| ...)`: 하나의 값으로 축약.
  - `for_each(|x| ...)`: 각 요소에 대해 실행.
  - `find(|x| ...)`, `any(|x| ...)`: 요소 검색 및 조건 확인.

에러 처리 (Error Handling)

- `panic!`: 복구가 불가능한 치명적인 에러가 발생했을 때 프로그램을 즉시 종료합니다.
- `Result\<T, E\>`: 성공(`Ok(T)`) 또는 실패(`Err(E)`)를 나타내는 열거형으로, 복구 가능한 에러 처리에 사용됩니다.
- `?` 연산자: 에러 발생 시 해당 에러를 호출자에게 즉시 전달(전파)하여 코드를 간결하게 만듭니다.

제네릭(Generics), 트레잇(Traits), 라이프타임(Lifetimes)

- **제네릭**: 타입을 추상화하여 다양한 데이터 타입에 동작하는 유연한 코드를 작성합니다.
- **트레잇 (Traits)**:
  - `Clone` / `Copy`: 값의 복제 및 복사 동작 정의.
  - `Debug` / `Display`: 포맷팅 출력 (`{:?}` vs `{}`).
  - `Default`: 기본값 생성 (`Default::default()`).
  - `Drop`: 메모리 해제(소멸자) 시 동작 정의.
  - `PartialEq` / `Eq`: 값의 비교 동작 정의.
- **라이프타임**: 참조자가 유효한 범위를 명시하여 메모리 안전성을 보장합니다.

스마트 포인터 (Smart Pointers)

- `Box<T>`: 값을 힙 메모리에 할당하고 소유권을 관리합니다.
- `Rc<T>`: 단일 스레드 환경에서 여러 소유자를 허용하는 참조 카운팅 스마트 포인터입니다.
- `Arc<T>`: 여러 스레드에서 안전하게 소유권을 공유할 수 있는 원자적 참조 카운팅 포인터입니다.
- `RefCell<T>` / `Mutex<T>`: 런타임에 빌림 규칙을 검사하여 내부 가변성(Interior Mutability)을 제공합니다.

고급 Rust 문법 및 활용

제네릭과 트레잇 바운드 심화

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

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

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

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

라이프타임 활용 심화

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

// 라이프타임 엘리전(생략) 규칙: 일정한 패턴에서는 명시하지 않아도 컴파일러가 유추함
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[..]
}
```

매크로 시스템 (Macros)

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

// 프로시저 매크로 (Procedural Macros): 코드 생성 및 메타프로그래밍 지원
#[proc_macro_derive(MyTrait)]
pub fn my_trait_derive(input: TokenStream) -> TokenStream { ... }
```

비동기 프로그래밍 (Async/Await)

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

// 비동기 함수 정의
async fn fetch_data() -> String {
    sleep(Duration::from_secs(1)).await;
    "데이터 로드 완료".to_string()
}

// 여러 비동기 작업 동시 실행 (Join)
async fn parallel_work() -> (String, String) {
    let (res1, res2) = tokio::join!(fetch_data(), fetch_data());
    (res1, res2)
}
```

고급 에러 처리 패턴

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

impl std::error::Error for MyError {}

// From 트레잇을 구현하여 에러 타입 자동 변환 지원
impl From<std::io::Error> for MyError {
    fn from(err: std::io::Error) -> Self { MyError::Io(err) }
}
```

함수형 프로그래밍 패턴 활용

```
// 이터레이터 체이닝을 통한 선언적 데이터 처리
fn process_numbers(numbers: Vec<i32>) -> Vec<i32> {
    numbers
        .into_iter()
        .filter(|&x| x > 0)
        .map(|x| x * 2)
        .collect()
}
```

메모리 안전성 및 성능 최적화

- Zero-Cost Abstractions: 추상화를 사용하더라도 런타임 오버헤드가 거의 없는 최적화된 기법을 제공합니다.
- 모노모피즘 (Monomorphization): 제네릭 함수를 실제 사용되는 구체 타입별로 컴파일 시점에 생성하여 정적 디스패치 성능을 보장합니다.

웹 개발 실전 (Warp 프레임워크)

```
use warp::Filter;

async fn handler() -> Result<impl warp::Reply, warp::Rejection> {
    Ok(warp::reply::json(&"Hello, Warp!"))
}

#[tokio::main]
async fn main() {
    let route = warp::path("hello").and_then(handler);
    warp::serve(route).run(([127, 0, 0, 1], 3030)).await;
}
```

시스템 프로그래밍 활용

- 안전하지 않은 블록(`unsafe`)을 통한 저수준 제어 및 외부 라이브러리 연동

```
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) }
    }
}
```

테스팅 및 벤치마킹

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() { assert_eq!(2 + 2, 4); }
}
```

패키지 관리 및 Cargo 활용

- `Cargo.toml`을 통해 의존성, 빌드 프로필(Release/Debug), 프로젝트 메타데이터를 통합 관리합니다.
- **크로스 컴파일**: `rustup target add`와 `cargo build --target` 명령어를 통해 다양한 플랫폼 환경의 바이너리를 생성할 수 있습니다.