03-21.md

1. Lezione del 21/03

Il concetto di controllo di una variabile essenzialmente esprime la scelta di cosa possa essere mutabile e cosa no. Facendo un uso frequente di questo concetto, posso anche evitare situazioni ambigue in cui per errore ho modificato una variable che invece doveva rimanere invariata

Tuttavia un sistema così costruito, e che adoperi anche un borrow checker, è molto complicato da saper usare. Nello spettro diametralmente opposto trovo invece il C, dove il concetto di possesso non è definito dal linguaggio ed inoltre la mutabilità è alla base del linguaggio, con la non mutabilità che è richiesto dal professore.

Infine il linguaggio espone anche 3 paradigmi di accesso alle risorse: accesso puntuale al dato, accesso in sola lettura al dato, accesso in scrittura vincolato al possedere il dato.

L'accesso alla riga di comando è orientato alla programmazione funzionale, a differenza del C, dove i parametri sono una matrice di caratteri, o un vettore di stringa che dir si voglia.

```
use std::env::args;
fn main() {
    let args: Vec<String> = args().skip(1).collect();
    if args.len() > 0 {
        // We have args and we can use them
    }
}
```

Notiamo l'uso della funzione `skip()` per saltare il primo parametro che rimane comunque il nome del programma chiamato e l'uso di una `collect()` per convertire i parametri da riga di comando in un vettore di stringhe.

Per l'IO, la libreria standard contiene tutto quel che è necessario per interfacciarsi ad esso. Le informazioni ritornate da tali funzioni spesso non sono il dato vero e proprio ma, come spesso accade in Rust per indicare comodamente la corretta esecuzione di una funzionalità, di una `Result<>`.

`Result` mette a disposizione due funzioni per capire istantaneamente se l'esecuzione è andata a buon fine: `is_ok()` e `is_err()`. Dopo aver verificato, possiamo accedere al dato con con la funzione `unwrap()` che dobbiamo star attenti a richiamare solo dopo la verifica, perché se si tentasse di eseguirla su su una `Result<>` che ha in realtà ritornato un errore, il programma avvierebbe un `panic!()`.

```
let mut string = String::new();
if io::stdin().read_line(&mut string).is_ok() {
    prinln!("Got {}", s.trim());
}
```

Seguono due ore di code review abbastanza dimenticabile

La programmazione ad oggetti richiede di concepire, prima dell'algoritmo, come i nostri dati sono stati strutturati. In C il costrutto `struct` permette di creare appunto strutture dal contenuto eterogeno, ma non è un linguaggio propriamente orientato agli oggetti come lo è ad esempio il C++. Notiamo che dal punto di vista del compilatore le due cose sono equivalenti se non per l'accesso ai dati: di base in una `struct` tutti i campi sono pubblici, mentre nei linguaggi ad oggetti possiamo dichiararli come privati o protetti.

Due tipi di dato presenti in C/C++ e spesso trascurati sono gli enumeratori e le union. I primi permettono di associare un etichetta ad un valore puramente numerico, mentre i secondi permettono di interpretare un dato come due tipi diversi a seconda delle esigenze.

Un esempio di union in C++:

```
union sign {
    int svar;
    unsigned int uvar;
} number;
```

Questo è un esempio è un caso molto comodo. Infatti, in questo caso, sia `int` che `unsigned int` occupano la stessa memoria. Quindi è vero che io potrei utilizzare la dichiarata variabile `number` sia come intero che intero senza segno, qualora lo reputassi necessario, ma questo avrebbe un costo nullo.

Se infatti avessi provato a definire una union tra due tipi di dato di dimensione diversa, come ad esempio un `long` che occupa più byte di un `int`, di base ogni variabile di tipo `sign` che andremmo a dichiarare, dovendo poter accogliere entrambi i tipi di dato, si allineerà alla dimensione del dato più grande.