Invocare un tratto

L'invocazione di un tratto può avvenire in due modalità

- **statica** : se il tipo del valore è noto, il compilatore può identificare l'indirizzo della funzione da chiamare e generare il codice corrispondente senza alcuna penalitàNon c'è overhead ne in termini di tempo di esecuzione ne di memoria allocata
- **dinamica** : se si dispone di un puntatore ad un valore di cui il compilatore sa solo che implementa un dato tratto, occorre eseguire una chiamata indiretta, passando per una VTABLE

Variabili o parametri destinati a contenere puntatori (riferimenti, Box, Rc, ...) ad un valore che implementa un tratto sono annotati con la parola chiave dyn

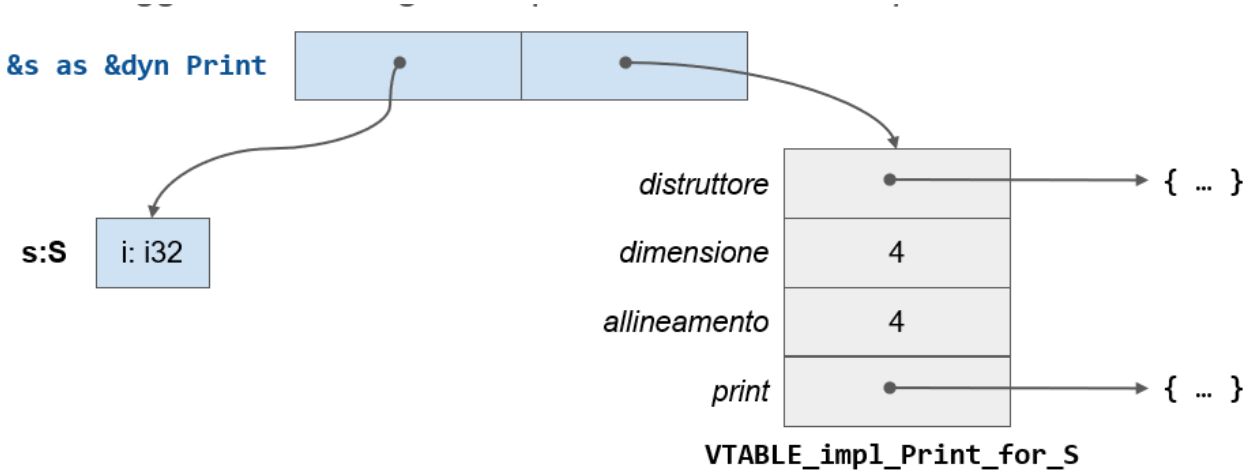
```
trait Print {
    fn print(&self);
}
struct S {i:32}
impl Print for S {
    fn print(&self){
        println!("S{}", self.i)
    }
}

fn process(v: &dyn Print){ //ricevo un puntatore dinamico
//è un qualcosa che implementa il tratto Print
//senza dyn avrei errore
//dyn è un fat pointer : prima parte replica puntatore vero  (8 byte) e proprio, secondi 8 byte puntatore alla vtable
    v.print();
}

fn main () {
    process(&s{i:o}) //conosco S solo come qualcosa che implementa print : descritta tramite fat pointer
}
```

Vtable oggetti tratto

- I riferimenti/puntatori ai tipi tratto vengono detti oggetti-tratto Possono essere condivisi o mutabili e devono rispettare le regole dell'esistenza in vita del valore a cui fanno riferimento
- Gli oggetti-tratto vengono implementati tramite fat pointer
- il primo è un puntatore alla funzione drop , il secondo campo contiene la dimensione dell'oggetto a cui stiamo puntando, il terzo elemento indica l'allineamento e poi seguono tanti puntatori quanti sono i metodi definiti dal tratto



Tratti nella libreria standard

- Usati in rust in modo pesante: definiscono per i tipi presenti nella libreria standard e anche per i custom una serie di utilizzi:
- posso confrontare due oggetti se implementano il tratto **Eq** o **PartialEq**
- Per < o <=,> o >= ecc tratto **Ord**
- Per le somme il tipo di a deve implememntare **ADD** e cosi via...

Gestire i confronti di uguaglianza

- Implementare un confronto di uguaglianza richiede il rispetto delle proprietà di riflessività, simmetria e transitività
- La relazione di riflessività non vale per numeri con la ",".
    - Ci sono tre valori particolari:

1. infinite
2. negative infinite
3. Nan

Questi numeri (1., 2.) sconvolgono i risultati delle operazioni e danno risultati particolari (es +infinito - infinito => forma indeterminata , 0/0..ecc).
Nan deve essere diverso da Nan. Nan composto con le N operazioni da origine a Nan Sempre. Potrei avere due variabili che hanno la stessa sequenza di bit ma che non devono avere uguaglianza. Introduciamo il tratto

- **partialeq** : per il caso di riflessività non garantita
- **eq** , sottotratto di partialeq (se implemento eq allora implemento anche partialeq) (tutti i tipi che implementano eq devono implementare partialeq poichè ne è sottotratto)

Ci posso essere diverse implementazioni **DI RIGHT HAND SIDE** : Rhs definisce cosa posso mettere a destra dell'eguale (di default coincidono). Ha senso anche fare confronti tra due cose diverse (es reale con complesso).
Il metodo ne(...) è normalmente preso dalla sua implementazione di default come opposto del risultato di eq(...). Il tratto eq e partial eq contengono due metodi:

- **eq(&self, other : &RHS) ->bool**,
- **ne(&self, other : &RHS)**

Gestire i confronti di ordine

I tratti **PartialOrd e Ord** permettono rispettivamente di definire relazioni d'ordine parziali e totali su un dato insieme di valori

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where Rhs: ?Sized, {
    fn partial_cmp(&self, other: &Rhs) ->
       Option<Ordering>;

    // metodi con implementazione di default
    fn lt(&self, other: &Rhs) -> bool;
    fn le(&self, other: &Rhs) -> bool;
    fn gt(&self, other: &Rhs) -> bool;
    fn ge(&self, other: &Rhs) -> bool;
}
/*-------------------------------------*/
trait Ord: Eq + PartialOrd<Self> {
  fn cmp(&self,other: &Self)
                       -> Ordering;
  // implementazione di default
fn max(self, other: Self) -> Self;
fn min(self, other: Self) -> Self;
fn clamp(self, min: Self, max: Self)
                           -> Self;
}
```

- serve coerenza tra i tratti
- i tratti li genera il compilatore con la macro **define**

Esempio Clion

```
use std::cmp::Ordering;
use std::cmp::Ordering::Equal;
#[derive(PartialEq,Eq,PartialOrd,Ord,Debug)] //qua lo fa il compilatore
struct S {
    a : i32,
    b: bool,
}/*
impl PartialEq for S  {
    fn eq(&self, other: &Self) -> bool {
        return self.a==other.a && self.b==other.b;
        //implemento eq : come faccio a dire se sono equali
    }
*/
fn main() {
    let s1=S{a:1, b:true};
    let s2=S{a:2, b: true};

    if s1==s2 {
        println!("sono uguali");
        // println!("{:?} sono uguali {:?}",s1,s2);
        // in console { a: 1, b: true } sono uguali S { a: 1, b: true }
    }
    else {
        println!("sono diversi");
    }
}
```

Visualizzare i contenuti

- Non si può derivare direttamente display
- Macro come println! e format! consentono di stampare un valore associato al segnaposto {} a condizione che tale valore implementi il tratto Display
- Tale tratto rappresenta la capacità del tipo di creare una visualizzazione di un proprio valore comprensibile ad un utente finale
- Per poter implementare display: devo implementare

```
trait Display {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result;
}
//fmt è da implementare a mano
```

Copia e duplicazione

Il tratto **Clone** indica la capacità di creare un duplicato di un valore dato : stesso contenuto logico ma non di bit.
Il tratto **Copy** è un sotto-tratto di Clone.
La presenza del tratto Copy trasforma la semantica delle operazioni di assegnazione: quello che normalmente determina un **movimento**, che rende inaccessibile il valore originale, diventa una copia. Copy è mutualmente esclusivo con drop La definizione di copy non include alcun metodo: introduce solo una proprietà. Si basa su memcpy().

Rilasciare le risorse

Se un tipo implementa il** tratto drop **vuol dire che ha operazioni particolari da eseguire al rilascio.
E' mutualmente esclusivo con copy.
Si accompagna alla funzione globale

```
fn drop<T>(_x:T){}
```

essa forza il passaggio di possesso di un valore ad una nuova variabile(_x) che uscirà di scena subito, provocandone la distruzione

Indicizzare

E' possibile utilizzare un tipo mio come Array tramite Index se si implementano i tratti Index e IndexMut

- l'espressione t[i] viene riscritta come *t.index (i) se si accede in lettura, *t.index_mut(i) se in lettura

```
trait Index<Idx> {
    //Accesso al contenuto di un oggetto in lettura
  type Output: ?Sized;
  fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    // //Accesso al contenuto di un oggetto in scrittura
  fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

Indicizzare una struttura dati

```
// Vec<i32> implementa Index<usize, Output = i32>
let vec = vec![1, 2, 3, 4, 5]; //macro per creare un vettore inizializzato con
quell'array
let num: i32 = vec[0];
let num_ref: &i32 = &vec[0];

// Ma implementa anche Index<Range<usize>, Output=[i32]>
assert_eq!(&vec[1..4], &[2, 3, 4]);
```

Deferenziare un valore

- c'è il tratto **deref**: permette di considerare un tipo qualunque come fosse un puntatore
- uso ***t per deferenziarlo**
- La sintassi *t per un tipo che implementa Deref e che non sia un riferimento né un puntatore nativo equivale a *(t.deref()) e restituisce un valore immutabile di tipo Self::Target
- Analogamente, se il tratto implementa DerefMut, *t equivale a *(t.deref_mut()) e restituisce un valore mutabile di tipo Self::Target

```
trait Deref {
  type Target: ?Sized;
  fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
  fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Definire un intervallo

- Permette di definire gli estremi di qualcosa
  - Attraverso l'utilizzo dell'operatore `..` è possibile definire intervalli di valori per tutti i tipi che implementano il tratto **`RangeBounds<T>`**
    - Quest'ultimo è implementato da diversi tipi base in Rust e consente l'utilizzo di sintassi come `..`, `a..`, `..b`, `..=c`, `d..e`, `f..=g`
    - E' necessario implementare i metodi **`end_bound(&self)`** e **`start_bound(&self)`** che ritornano entrambi il tipo **`Bound<&T>`**

```
pub trait RangeBounds<T>{
  fn start_bound(&self) -> Bound<&T>;
  fn end_bound(&self) -> Bound<&T>;
  fn contains<U>(&self, item: &U) -> bool { ... }
}
```

```
pub enum Bound<V>{
  Included(V),
  Excluded(V),
  Unbounded,
}
```

Conversione tra tipo

- I tratti **`From`** e **`Into`** permettono di effettuare conversioni di tipo, prendono possesso del valore lo convertono e ritornano il possesso al chiamante
  - I tratti sono perfettamente duali, scrivere **`T: From<i32>`** equivale a scrivere **`i32: Into<T>`**
  - Se si implementa il tratto **`From`**, il tratto **`Into`** viene generato automaticamente
  - L'implementazione del tratto **`From`** **non è simmetrica**: se è possibile passare dal tipo T al tipo U, non è affatto detto che sia possibile tornare indietro dal tipo U al tipo T

```
trait From<T>: Sized {
 fn from(other: T) -> Self;
}

trait Into<T>: Sized {
 fn into(self) -> T;
}
```

Conversione tra tipi

```
struct Point {
  x: i32,
  y: i32,
}

impl From<(i32, i32)> for Point {
  fn from((x, y): (i32, i32)) -> Self {
    Point { x, y }
  }
}

impl From<[i32; 2]> for Point {
  fn from([x, y]: [i32; 2]) -> Self {
    Point { x, y }
  }
}
```

```
fn main() {
  // from
  let p1 = Point::from((3, 1));
  let p2 = Point::from([5, 2]);

  // into
  let p3: Point = (1, 3).into();
  let p4: Point = [4, 0].into();

  // ERRORE! non vale la simmetria
  let a1 = <[i32; 2]>::from(point);
  let a2: [i32; 2] = point.into();
  let t1 = <(i32, i32)>::from(point);
  let t2: (i32, i32) = point.into();
}
```

Conversione tra tipi

- Per gestire le conversioni tra tipi che possono fallire vengono forniti i tratti **`TryFrom`** e **`TryInto`** che posseggono le stesse proprietà di **`From`** e **`Into`**
  - I metodi **`try_from`** e **`try_into`** ritornano il tipo **`Result<T,E>`** per garantire la gestione dell'errore

```
pub trait TryFrom<T>: Sized {
  type Error;
  fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T>: Sized {
  type Error;
  fn try_into(self) -> Result<T, Self::Error>;
}
```

- generano un result invece di generare direttamente il tipo T
- si generano dei panic per poi gestire l'errore in caso di errore

Conversione tra tipi

- Il tratto **`FromStr`** permette di gestire la conversione da stringa e l'eventuale fallimento
  - Il metodo **`from_str()`** viene implicitamente richiamato tutte le volte che si utilizza il metodo **`parse()`**
  - Il tratto **`FromStr`** possiede la stessa firma del tratto **`TryFrom<&str>`**
  - Non è possibile richiamare **`parse()`** su elementi che posseggono un lifetime (es. **`&i32`**)

```
pub trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

- utile se devo trasformare una stringa in qualcosa

Descrivere un errore

- In Rust gli errori non vengono lanciati ma ritornati attraverso l'utilizzo del tipo **`Result<T,E>`**
- Il tipo generico **`E`** può assumere qualsiasi valore tuttavia è preferibile utilizzare solo tipi che implementano il tratto **`Error`**

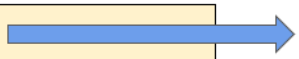
```
pub trait Error: Debug + Display {
  fn source(&self) -> Option<&(dyn Error + 'static)> { ... }
  fn backtrace(&self) -> Option<&Backtrace> { ... }
  fn description(&self) -> &str { ... }
  fn cause(&self) -> Option<&dyn Error> { ... }
}
```

Derive

Derivare metodi automaticamente

- Sebbene sia possibile implementare a mano tutti i metodi richiesti da un certo tratto, a volte è meglio affidarsi al compilatore
  - Se la definizione di una struct o di una enum è preceduta dall'attributo **`#[derive(...)]`**, il compilatore provvede ad aggiungere, automaticamente, l'implementazione dei tratti indicati all'interno del costrutto derive(...)
  - Solo un certo sottoinsieme di tratti possono essere generati automaticamente, spesso a condizione che i tipi dei dati contenuti nel tipo per cui se esegue la derivazione soddisfino opportuni vincoli (come, ad esempio, implementare a propria volta il tratto)

```
#[derive(PartialEq)]
struct Foo<T> {
    a: i32,
    b: T,
}
```



```
impl<T: PartialEq> PartialEq for Foo<T> {
    fn eq(&self, other: &Foo<T>) -> bool {
        self.a == other.a && self.b == other.b
    }

    fn ne(&self, other: &Foo<T>) -> bool {
        self.a != other.a || self.b != other.b
    }
}
```