Polimorfismo

- la necessità di minimizzare il codice scritto spinge verso l'identificazione di pattern comuni
- La soluzione individuata è il **polimorfismo**: capacità offerta dai linguaggi di associare comportamenti comuni ad un insieme di tipi differenti
- si può implementare con programmazione generica, interfacce o ereditarietà

Polimorfismo nei vari linguaggi

- in c non c'è supporto sintattico specifico per implementare il polimorfismo
- il linguaggio ***C++*** supporta il concetto di ereditarietà e il concetto di metodo virtuale

Tratti

- Non c'è l'ereditarietà in rust
- Equivalente in rust delle interfacce di java
- Un tratto esprime la capacità di un tipo di eseguire una certa funzionalità :
    - Un tipo che implementa

    ```
    std::io::Write
    ```

    può scrivere dei byte.

Un tipo che implementa std::iter::Iterator può produrre una sequenza di valori. Un tipo che implementa std::clone::Clone può creare copie del proprio valore. Un tipo che implementa std::fmt::Debug può essere stampato tramite println!() usando il formato {:?}.

- A differenza di quanto accade in C++ o Java, se si invoca su un valore una funzione relativa ad un tratto, **non si ha - normalmente - un costo aggiuntivo.** Né gli oggetti che implementano tratti hanno una penalità in termini di memoria per ospitare il puntatore alla VTABLE (ad esempio enum non ha vtable). Tale costo si presenta solo quando si crea esplicitamente un riferimento dinamico (**&dyn TraitName**). Serve quando non voglio ritornare una funzione ma un Tratto (ad esempio qua si ritorna qualcosa su cui si potrà scrivere). In questo caso si ritorna un **fat pointer**

Definire e usare un tratto

- Si definisce un tratto con la sintassi

```
trait SomeTrait { fn someOperation(&mut self) -> SomeResult; … }
```

- Una struttura dati concreta, come struct od enum, può esplicitamente dichiarare di implementare un dato tratto attraverso il blocco seguente

```
impl SomeTrait for SomeType { … }
```

- Dato un valore il cui tipo implementa un tratto, è possibile invocare su tale valore i metodi del tratto, con la normale sintassi basata sul '.'
    - A condizione che il tratto sia stato dichiarato nello stesso crate o che sia stata importato attraverso il costrutto

```
use SomeNamespace::SomeTrait;
```

- Alcuni tratti (come Clone e Iter) non necessitano di essere importati esplicitamente in quanto fanno parte di una porzione di codice della libreria standard (il cosiddetto preludio) che viene importato automaticamente in ogni crate

Definire e usare un tratto

- La parola chiave **`Self`**, nella definizione di un tratto, si riferisce al tipo che lo implementerà

```
trait T1 {
  fn returns_num() -> i32;    //ritorna un numero
  fn returns_self() -> Self;  //restituisce un'istanza del tipo che lo implementa
}
```

```
struct SomeType;
impl T1 for SomeType {
  fn returns_num() -> i32 { 1 }
  fn returns_self() -> Self {SomeType}
}
```

```
struct OtherType;
impl T1 for OtherType {
  fn returns_num() -> i32 { 2 }
  fn returns_self() -> Self {OtherType}
}
```

- Self è una metavariabile : si riferisce al tipo che la implementerà

- Se una funzione tra quelle definite da un tratto non usa, come primo parametro, né **`self`** né un suo derivato (**`&self`**, **`&mut self`**, …), questa non è legata all'istanza del tipo che la implementa
  - Può essere invocata usando come prefisso il nome del tratto o il nome del tipo che la implementa

```
trait Default {

  fn default() -> Self

}
```

```
fn main() {
  let zero: i32 = Default::default();
  let zero_again = i32::default();
}
```

- Non so cosa ritornare : interi ritornano zero, stringhe una stringa vuota

- Un metodo è una funzione che utilizza come primo parametro la parola chiave **`self`** o una sua variazione (**`&self`**, **`&mut self`**, …)
  - Il tipo del parametro self può anche essere **`Box<Self>`**, **`Rc<Self>`**, **`Arc<Self>`**, **`Pin<Self>`**
- I metodi sono invocati con l'operatore **`.`** (punto) sul tipo che li implementa

```
trait T2 {
  fn takes_self(self);
  fn takes_immut_self(&self);
  fn takes_mut_self(&mut self);
}
```

equivalenti

```
trait T2 {
  fn takes_self(self: Self);
  fn takes_immut_self(self: &Self);
  fn takes_mut_self(self: &mut Self);
}
```

```
trait ToString {
  fn to_string(&self);
}
```

```
fn main() {
  let five =  5.to_string();
}
```

- Un tratto può avere uno o più tipi associati
  - Questo permette alle funzioni del tratto di fare riferimento, in modo astratto, a tali tipi che dovranno essere poi specificati nel contesto del tipo che implementa il tratto stesso

```
trait T3 {
  type AssociatedType;
  fn f(arg: Self::AssociatedType);
}
```

```
struct SomeType;
impl T3 for SomeType {
  type AssociatedType = i32;
  fn f(arg: Self::AssociatedType) {}
}
```

```
struct OtherType;
impl T3 for OtherType {
  type AssociatedType = &str;
  fn f(arg: Self::AssociatedType) {}
}
```

```
fn main() {
  SomeType::f(1234);
  OtherType::f("Hello, Rust!");
}
```

-Chi implementa il tratto : deve anche dire cos'è il tipo per lui

- Nella definizione di un tratto è lecito indicare, per una data funzione, un'implementazione di default
  - Le funzioni che implementano il tratto saranno libere di adottarla o potranno sovrascriverla con altro codice, purché venga rispettata la firma delle funzione (tipo dei parametri e del valore di ritorno)
  - Questo è particolarmente comodo in quelle situazioni in cui un dato metodo può essere implementato in funzione di altri metodi del tratto

```
trait T4 {
  fn f() { println!("default"); }
}
```

```
struct SomeType;
impl T4 for SomeType { }
//uso dell'implementazione di default
```

```
struct OtherType;
impl T4 for OtherType {
  fn f() { println!("Other"); }
}
```

```
fn main() {
  SomeType::f();  // default
  OtherType::f(); // Other
}
```

- La notazione **`trait Subtrait: Supertrait {...}`** indica che i tipi che implementano *Subtrait* devono implementare anche *Supertrait*
  - Le due implementazioni sono tra loro indipendenti ed è possibile che, per un dato tipo, una si avvalga dell'altra o viceversa

```
trait Supertrait {
  fn f(&self) {println!("In super");}
  fn g(&self) {}
}

trait Subtrait: Supertrait {
  fn f(&self) {println!("In sub");}
  fn h(&self) {}
}
```

```
struct SomeType;
impl Supertrait for SomeType {}
impl Subtrait for SomeType {}

fn main() {
  let s = SomeType;
  s.f(); //Errore: chiamata ambigua
  <SomeType as Supertrait>::f(&s);
  <SomeType as Subtrait>::f(&s);
}
```

- L'invocazione dei metodi di un tratto può avvenire in due modalità distinte
  - Invocazione **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 penalità
  - Invocazione **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: i32 }
impl Print for S {
  fn print(&self){
    println!("S {}", self.i); }
```

```
fn process(v: &dyn Print){
  v.print();
}

fn main() {
  process(&S{i: 0});
}
```