I02_2 TRATTI E GENERICS

Polimorfismo: offrire comportamenti comuni ad oggetti che hanno qualcosa in comune tra di loro.

*Nota: in **c++** si parla di ereditarietà e ereditarietà multipla ma si rischia codice duplicato. Quindi, prendo cose diverse con vari codici che eventualmente posso sovrascrivere.*
*Nota: in **Java** invece è possibile solo ereditarietà singola ma fornisce la possibilità di definire varie interfacce → con metodi.*
Nota: si inizia a pensare a programmazione generica → ci sono degli algoritmi dove le operazioni sono indipendenti dal tipo di dato (es: ordinamento di lista, push, pop non dipendono dal tipo di dato presente ma ad esempio da alcune caratteristiche come confrontabilità, etc…)

- *In c++ → monoformizzazione → quando mi dici cosa ci metti dentro la lista, sintetizzo il codice specifico per il tipo effettivo a partire dal codice generico*
  - *Non è a conoscenza delle sue classi → no introspezione*
  - *Non tutti i metodi devono essere virtuali → programmatore deve specificare virtual.*
  - ***VTABLE**: tabellina intermedia che mi permette di usare metodi indiretti*
    - *Contiene array con indirizzo effettivo dei metodi virtuali implementati dalla classe*
- *In java → l'oggetto list non sa cosa ha dentro a causa del type erasure → tutto è object*
  - *Polimorfismo+metodi virtuali (tengono traccia della classe a cui effettivamente appartiene)*
- *In Rust non c'è ereditarietà ma riesce a implementare poliformismo diversamente*

Polimorfismo in c++:

```
class Alfa {
  bool b;
public:
  virtual int getValue() { return 1; }
};

class Beta: public Alfa {
  int i;
public:
  virtual int getValue() { return 2; }
};

class Gamma: public Alfa {
  char c;
public:
  virtual int getValue() { return 3; }
};
```

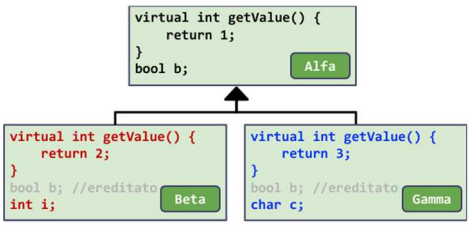
```
Alfa *ptr1 = new Alfa();
Alfa *ptr2 = new Beta();
Alfa *ptr3 = new Gamma();

ptr1-> getValue(); // 1

ptr2-> getValue(); // 2

ptr3-> getValue(); // 3
```

Beta deriva da alfa, eredita bool b ma non ci può fare nulla.
Gamma è sorella di Beta.

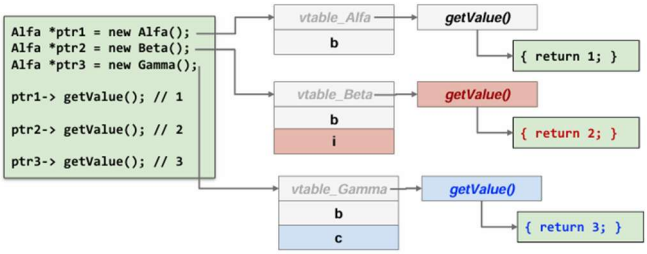


Dichiaro puntatori di tipo Alfa (ptr123), posso farlo perché i puntatori hanno tutti lo stesso peso e Beta e Gamma sono anche Alfa. Ptr punteranno ad un blocco di Heap contenenti rispettivamente un Alfa, un Beta e un Gamma.

*Quando dico Alfa ptr= new Alfa, sto puntando ad un blocco che è maggiore ad 1 byte in quanto, oltre al booleano, mi viene dato anche il **puntatore ai metodi virtuali** (getValue nero).*
*Quando scrivo ptr2=newBeta: si fa new Beta e poi si salva il puntatore in ptr2 → in beta c'è un puntatore ad un blocco con un puntatore alla vtable e poi i vari campi privati. La **vTable** di Beta presenta i metodi ereditati (nel caso sovrascritti) e quelli che implementa → la vTable di Beta conterrà solo il metodo getValue rosso.*

***GetValue** è un metodo virtuale quindi*

- *Vado su ptr*
- *Trovo puntatore*
- *Vado su tabella*
- *Prendo i-esima entry*
  - *Puntatore all'indirizzo della funzione da eseguire*



***Nota**: Nel caso in cui metodi non virtuali, si accede diretto senza passare da vtable.*

Inoltre, c++ permette di creare dei metodi virtuali astratti → metodi che non ho ma che saranno usati dalle mie sottoclassi (in java abstract davanti al metodo e alla classe). Per dichiarare una funzione virtuale astratta la si dichiara "=0;".
una classe contenente un metodo virtuale stratto, viene definita classe virtuale astratta e non è possibile istanziarla ma solo istanziare die suoi figli. In c++ è possibile dichiarare una classe vuota → classe astratta pura, con soli metodi virtual"=0;" (corrispondente delle interfacce java).

Riepilogando in c++:
Ogni istanza di una classe dotata di metodi virtuali dispone di un campo nascosto che contiene il puntatore alla VTABLE.
La VTABLE contiene un array con l'indirizzo effettivo dei metodi virtuali implementati dalla classe.
Quando un metodo virtuale viene invocato, il compilatore genere le istruzioni necessarie ad accedere alla VTABLE e prelevare l'indirizzo da chiamare.

In rust ci piace **il concetto di composizione** che indica che una parte di me è fatta così → quindi interfacce → classi virtuali astratte pure → TRATTI. Non di ereditarietà in quanto implica l'essenza e non ci piace.

Rust:

Implementa i **Tratti** → metodi che devono essere a disposizione. Nel tratto posso inserire dei metodi e una relativa implementazione che non viene considerata ereditata ma di default → se chi implementa il tratto non da una sua versione, uso quella di default.

Uso i tratti per esprimere comportamenti comuni che cose molto diverse possono avere.

- 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 **`{:?}`**

Nella maggior parte delle implementazioni in cui si usano i tratti, non si hanno costi aggiuntivi di spazio/tempo.
Ci sono alcune situazioni in cui il compilatore non può capire cosa deve fare e aggiunge qualcosa.

- In questi casi il compilatore costringe ad usare un ref dyn (**&dyn**) → reference fat(grasso) in quanto punta all'oggetto (prima parte 8byte) e VTABLE (seconda parte).

--codice scritto su rustRover → tratti

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

Implementazione del tratto si esegue dove si definisce il tratto o dove si definisce la struttura dati. `impl SomeTrait for SomeType { … }`

Nota: se si vuole usare un tratto definito in un altro file, lo devo importare tramite **use**. Ci sono alcuni tratti che non bisogna importare in modo esplicito (es: clone, iter) `use SomeNamespace::SomeTrait;`

All'interno del tratto si usa la parola chiave **self** che prende valore per chi lo implementa.

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

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

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

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

equivalenti

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

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

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

Nota: quando definisco dei tratti, posso associare dei **tipi**.
→*es: le classi che implementano T3 avranno il compito di dire cosa è Self e cosa è l'AssociatedType*

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

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

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

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

Nota: implementazione di **default** → chi implementa questi tratti può scegliere se tenere quella o crearne una nuova → *SomeType implementa default mentre OtherType sovrascrive f*

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

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

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

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

Nota: i tratti possono essere legati in una struttura di implicazione → definisco che i tipi che vogliono **implementare Subtrait devono anche implementare Supertrait.**

`trait Subtrait: Supertrait {...}`

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

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

```rust
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);
}
```

Invocare un tratto:

- **staticamente**: se tipo valore noto →compilatore identifica indirizzo della funzione e genera codice senza penalità
- **Dinamica**: Se compilatore non sa bene cosa sia → arriva un puntatore &dyn, quindi con puntatore alla Vtable, si va in Vtable e si va alla funzione
  (anche mut dyn o box)

```rust
trait Print {
  fn print(&self);
}
struct S { i: i32 }
impl Print for S {
  fn print(&self){
    println!("S {}", self.i); }
```

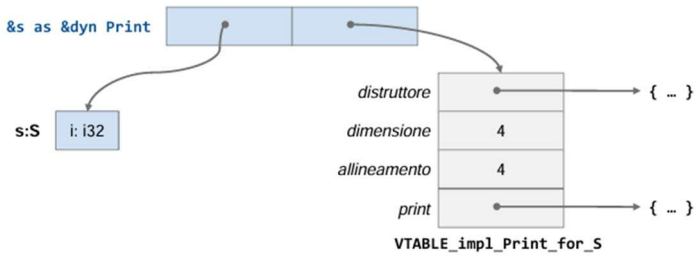
```rust
fn process(v: &dyn Print){
  v.print();
}

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

&dyn è di dimensione 16byte:i primo 8 puntano al dato effettivo, i secondi 8 puntano alla Vtable.

Vtable contiene:

- Metodo **drop** → distruttore associato al dato
  - Nel caso in cui non implementi drop → null
- **Dimensione** della struttura
- **Allineamento** → indica che nel caso di spostamento, deve essere allineato a multipli di n bytes.
- **Metodi** specifici.



NOTA: se compilatore sa cosa chiamare → va diretto altrimenti deve passare da vtable.

Tratti di libreria standard:

Rust ha un insieme di tratti base che permettono scorciatoie sintattiche.

- È possibile **confrontare due istanze** di un dato tipo con gli operatori **==** e **!=** se il tipo implementa i tratti Eq o PartialEq
  - **Eq**: implementa
    - Simmetrica: se confronto con me stesso deve essere uguale
    - Riflessiva: se a=b → b=a
    - Transitiva: se a=b, b=c → a=c
  - **PartialEq**: implementa
    - Simmetrica
    - transitiva

NaN è diverso da NaN → non vale la proprietà riflessiva → perciò in quel caso PartialEq.

- È possibile usare
  - + Add
  - – Sub
  - * Mul
  - / Div
  - % Rem
  - & BitAnd
  - ^ BitXor
  - ≪ Shl
  - ≫ Shr

- Il tipo associato **RHS** è definito, per default, come **Self**
  - In rari casi può essere necessario permettere confronti tra tipi differenti: quando lo si fa occorre fare particolare attenzione al rispetto delle proprietà suddette
- Il metodo **ne(...)** è normalmente preso dalla sua implementazione di default
  - Come opposto del risultato di **eq(...)**

- Essendoci il meccanismo del tipo associato, è possibile effettuare **confronti omogenei e non** → posso implementare un confronto tra cerchio e rettangolo se lo implemento.
  - Impl Eq for cerchio(type=Rettangolo) → fatto su rustrover

- È possibile usare < > ≤ ≥ se il tipo implementa i tratti **Ord e PartialOrd**
  - Implicano i tratti Eq e PartialEq
  - Richiedono di implementare Cmp o **PartialCmp**
    - PartialCmp ritorna un Option
  - Da PartialCmp prendono lt, le, gt, ge
  - Max
  - min
  - **clamp**: far sì che un certo valore sia dentro ad un certo intervallo → **limita un valore** tra min e max dell'intervallo.

```
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;
}
```

Per poter **stampare dei tipi custom**, vanno implementati dei tratti:

- **Display** → rappresentazione utile all'utente finale della nostra applicazione
  - Da implementare a mano, con aiuto della macro write!
  - println! + format!
    - Consentono di stampare un valore associato al segnaposto {} se tali valori implementano display

```
trait Display {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result;
}
```

- **Debug** → rappresentare una struttura dati in un modo sensato per il programmatore
  - Aggiungendo notazione #derive → ci aiuta il compilatore

```
#[derive(Debug, PartialEq, PartialOrd)]
```

Nota: se si vuole fare il confronto delle strutture dati in una maniera specifica, bisogna aggiungere un'implementazione che specifica il funzionamento:

- Se si vuole fare una uguaglianza specifica, va implementato a mano e rimuovere il derive per PartialEq

```
impl PartialEq for Cerchio{
    fn eq(&self, other: &Self) -> bool {
        self.r==other.r
    }
}
```

Ci sono **altri tratti** che implementano determinati comportamenti:

- **Clone** → possibile duplicare oggetto
  - Si può trasformare così un riferimento &T in un valore posseduto T
    - Trasformazione costosa in termini di tempo e memoria
- **Copy** → possibile fare copia
  - Originale non perde il possesso (non movimento ma copia)
  - La copia ottenuta è il semplice duplicato dei bit presenti nel valore originale
  - → implica presenza del tratto clone
- **Drop** → oggetto ha azioni da fare al rilascio
  - Mutualmente esclusivo con tratto copy → mi protegge dai doppi rilasci → non può esserci copy
  - Parente di funzione globale drop → riceve per movimento un oggetto e non ne fa nulla, quindi come viene chiusa, l'oggetto viene distrutto.

```
drop(&mut self)
```

- **Index** → trattare una cosa qualunque come fosse un array
  - Index mut → accedere in scrittura
    - Intero
    - Range → mi dà uno slice
  - Varie classi implementano i tratti Index e indexMut
  - Ci possono essere diverse implementazioni del tratto per un dato tipo
    - Es: Vec<T> consente di definire:
      - **indici numerici** che restituiscono valori di tipo T
      - indici di tipo intervallo che restituiscono **slice**

```
trait Index<Idx> {
  type Output: ?Sized;
  fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
  fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

```
// Vec<i32> implementa Index<usize, Output = i32>
let vec = vec![1, 2, 3, 4, 5];
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]);
```

- **Deref** → trasformare una qualsiasi cosa in un puntatore
  - Per poter usare deref ho in prestito self

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

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

```
struct Selector {
    elements: Vec<String>,
    current: usize
}

use std::ops::Deref;

impl Deref for Selector {
    type Target = String;
    fn deref(&self) -> &String { & self.elements[ self.current ] }
}

let mut s = Selector{elements: vec!["a".to_string(), "b".to_string()], current:0};
assert_eq!(*s, "a");

s.current = 1;
assert_eq!(*s, "b");
```

Slide non spiegate

- **RangeBounds<T>:** definire intervalli
  - 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>

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

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

- **From<T>:** permette il passaggio dal tipo T al tipo U

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

- **Into<T>**: permette il passaggio dal tipo U al tipo T
  - **Nota**: viene generato automaticamente se si implementa From

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

```rust
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 }
  }
}
```

```rust
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();
}
```

- **FromStr**: permette di gestire la coinvesione 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`)

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

Descrizione di errore: in Rust gli errori vengono ritornati attraverso Result<T,E>. Per quanto ruigayrda ilt ipo E, è preferibile usare solo tipi che implementano il tratto Error.

```rust
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> { ... }
}
```

```rust
#[derive(Debug)]
struct CustomError {
  info: String
}

impl std::fmt::Display for CustomError {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f,"{}",self.info)
  }
}

impl std::error::Error for CustomError {
  fn description(&self) -> &str {
    &self.info
  }
}
```

Derivare metodi automaticamente: 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(...)

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

```rust
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
    }
}
```

Funzioni generiche

In base a quello che ci metti effettivamente dentro → generano funzioni specializzate per quel particolare tipo a partire da codice generico → **monoformizzazione**: trasforma codice polimorfico in codice ben definito → versione specifica per il tipo.

```cpp
template <typename T>
T max(
   T t1,
   T t2) {
     return (t1 < t2 ?
        t2 : t1);
}
```
C++

```rust
fn max<T>(
   t1: T,
   t2: T) -> T where T: Ord {
     return
       if t1 < t2 { t2 }
       else { t1 }
}
```
Rust

- ***Vincolo nell'esempio*** *fn max: T deve essere qualcosa ordinabile (Ord) in quanto all'interno della funzione sto facendo un confronto*.
- ***Nota***: Il Borrow Checker si occupa di garantire che, se viene passato un valore, questo venga correttamente gestito dal punto di vista del possesso.

Compilatore verifica se esiste già la funzione con quel determinato tipo → se la trova esegue direttamente, altrimenti la crea a partire dalla generica e la fornisce.

Tipi generici

La programmazione generica può essere usata per specificare funzioni o tipi (Struct, enum e tuple)

```rust
struct MyStruct<T> where T: SomeTrait {
  foo: T,
  //…
}
```

```rust
impl<T> struct MyStruct<T>
  where T: SomeTrait {
  fn process(&self) { … }
}
```

Quando si usa quella struttura dati, si indica cosa sto inserendo dentro e quindi si crea con il tipo definito.

Sintassi: Rust offre due modi per specificare la sintassi dei vincoli sui tipi generici:

- Compatta: <T: SomeTrait>
- **Estesa**: <T> … where T: SomeTrait

  Per indicare più vincoli aggiungo il +

```rust
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
      data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

```rust
fn run_query<M,R>( data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

Nota sui tipi generici: *Quando il compilatore incontra la definizione di tipi / funzioni generici, si limita a verificare formalmente la coerenza del costrutto, senza generare alcun codice. Se in qualche parte del programma si utilizza un tipo/funzione generico legando le meta-variabili in esso contenute a tipi concreti, il costrutto generico viene istanziato, espandendo la definizione iniziale con i necessari dettagli necessari a generare il codice relativo. Se il costrutto generico, in parti diverse del programma, è legato a tipi concreti differenti, il compilatore genera ulteriori espansioni della definizione che, pur avendo una matrice comune, risulteranno indipendenti tra loro. Questo processo prende il nome di* ***monomorfizzazione*****.**

Tratti e tipi generici:

Tratti e tipi generici hanno un legame profondo. (possono coesistere)

- Tratti possono essere usati come vincoli per limitare l'utilizzo di un tipo generico
- Si può definire un tratto generico i cui metodi ricevono o restituiscono valori generici

Funzione non generica che opera su un oggetto-tratto: riceve un parametro da 16byte, il codice di dynamic_process usa la VTABLE contenuta nel *&mut dyn Write* di *w* per prendere il codice effettivo del metodo. Esisterà per una sola copia perché il tipo dell'oggetto lo capisce dalla vtable. → di conseguenza ci sarà meno codice ma anche una minor ottimizzazione.

- `fn dynamic_process(w: &mut dyn Write) { … }`
- Può prendere solo il parametro per puntatore

Funzione generica il cui parametro è vincolato da un tratto: generic_process riceve un puntatore semplice. Non esiste in un'unica istanza ma ho bisogno di più versioni compilate distinte per gestire i vari tipi di dato. → + ottimizzazioni possibili ma + codice

```rust
fn generic_process<T>(w: &mut T) where T:Write { … }
```

- Può prendere parametro per valore

NOTA*: Alcuni tratti hanno dei riferimenti statici per cui si deve implementare la funzione generica in quanto non possono essere passati ad oggetti tratto*

Alcune note aggiuntive:

- Gli oggetti-tratto richiedono l'uso di **fat pointer** per permetterne l'accesso
  - Ma non richiedono la duplicazione del codice dovuta al processo di monomorfizzazione
- L'uso di strutture dati generiche, in generale, porta a codice più efficiente
  - le chiamate alle funzioni non necessitano di transitare per la VTABLE
  - il compilatore, conoscendo il tipo concreto in fase di monomorfizzazione, può generare codice più compatto, *valutando il risultato dell'elaborazione delle parti costanti in fase di compilazione e sfruttare tecniche di code inlining per ridurre l'impatto dell'invocazione di funzioni*
- Non tutti i tratti permettono di definire oggetti-tratto
  - tratto non deve definire alcun metodo statico *(non deve usare self, &self, ..., come primo parametro)*
- Non è possibile definire un oggetto-tratto legato a più tratti disgiunti
  - è possibile, in una funzione generica, vincolare una meta-variabile ad implementare più tratti