



Polimorfismo

Tratti e programmazione generica

Polimorfismo

- **Polimorfismo**, deriva dal greco πολύς (molto) e μορφή (forma), dunque significa **multiforme**.
- In informatica:
 - Associare comportamenti comuni a oggetti i cui tipi sono molto diversi tra di loro.



Polimorfismo

- Sebbene il processo di analisi di un dominio applicativo tenda a favorire la sua suddivisione in una molteplicità di tipi distinti (classi, strutture, funzioni, ecc.), la necessità di minimizzare il codice scritto spinge verso l'identificazione di pattern comuni
 - Che possano essere condivisi tra tali tipi per unificare la struttura del codice
 - Principio DRY - *Don't Repeat Yourself*
- La soluzione individuata è il **polimorfismo**: capacità offerta dai linguaggi di associare **comportamenti comuni ad un insieme di tipi differenti**
 - Può basarsi sul paradigma della **programmazione generica**, dove è possibile formulare funzioni o strutture dati in cui uno o più tipi non sono specificati per nome, ma tramite simboli astratti, abilitando così la scrittura di codice in grado di operare con una molteplicità di tipi
 - Oppure può sfruttare la definizione di **interfacce** comuni, che possono essere implementate dai singoli tipi: questo consente di fare riferimento all'interfaccia invece che al tipo concreto
 - Nei linguaggi che implementano il concetto di **ereditarietà**, il polimorfismo si può realizzare derivando più classi concrete da una super-classe in cui è definito il comportamento comune

Polimorfismo in C e C++

- Il linguaggio C non ha nessun supporto sintattico specifico per l'implementazione del polimorfismo
 - Tuttavia, è possibile ricorrere ad opportuni pattern di programmazione per ottenere il comportamento richiesto
 - <https://stackoverflow.com/questions/8194250/polymorphism-in-c>
- Il linguaggio C++ supporta il concetto di ereditarietà (multipla) e il concetto di metodo virtuale
 - Un metodo così etichettato viene chiamato in modo indiretto, passando attraverso una struttura intermedia detta VTABLE
 - Questa contiene un array con l'indirizzo effettivo dei metodi virtuali che la classe implementa
 - Ogni istanza di una classe dotata di metodi virtuali dispone di un campo nascosto che contiene il puntatore alla VTABLE (**costo in termini di memoria**)
 - Quando un metodo virtuale viene invocato, il compilatore genera le istruzioni necessarie ad accedere alla VTABLE e prelevare l'indirizzo da chiamare (**costo in termini di tempo**)

Polimorfismo in C++

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

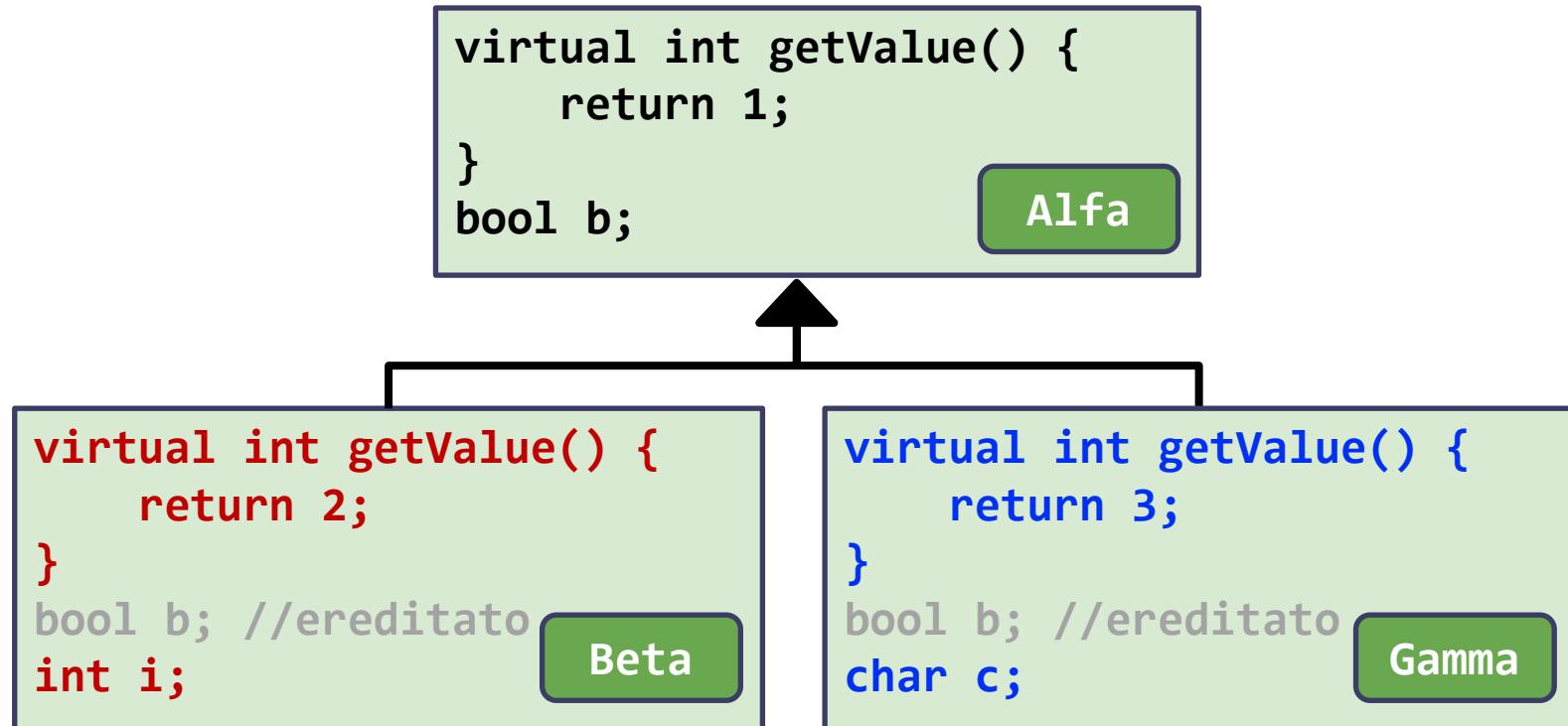
class Beta: public Alfa {
    int i;
public:
    virtual int getValue() { return 2; }
};

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
```

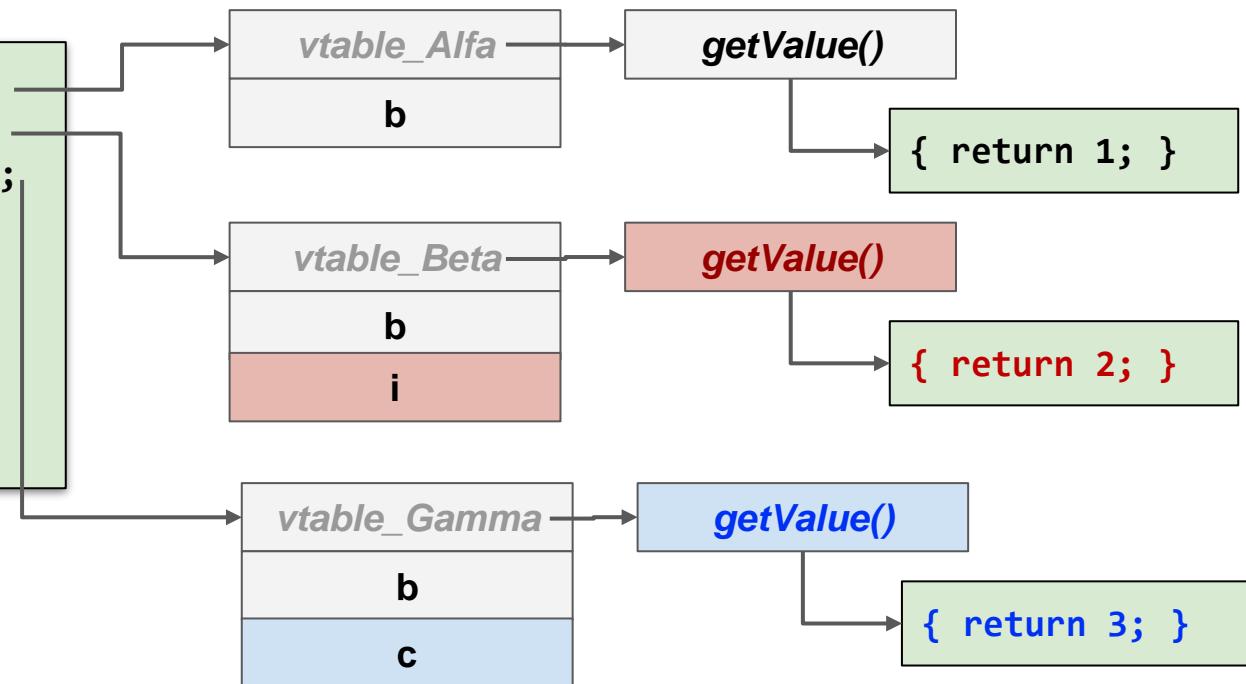
Polimorfismo in C++



Polimorfismo in C++

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

ptr1-> getValue(); // 1
ptr2-> getValue(); // 2
ptr3-> getValue(); // 3
```





Polimorfismo in C++

- Solo le funzioni membro denominate “virtual” sono polimorfiche
 - La presenza di metodi virtuali comporta una penalità in termini di spazio (ogni istanza contiene un puntatore alla VTABLE) e di tempo (ogni chiamata deve essere risolta passando tramite la VTABLE)
 - Una funzione membro non virtuale non ha costi aggiuntivi di chiamata
- E' possibile omettere il corpo di una funzione virtuale, dichiarandola “ = 0; ”
 - Questo rende la **funzione virtuale astratta**
- Se una classe contiene almeno una funzione virtuale astratta diventa una **classe astratta**
 - Classi di questo tipo non possono essere istanziate direttamente, ma possono essere usate come classi base da cui derivare sottoclassi concrete, purché dotate di un'implementazione per tutti i metodi astratti
- Una **classe astratta pura** contiene solo funzioni virtuali astratte
 - Equivalenti a quella che in altri linguaggi di programmazione si chiamano **interfacce** (es. Java, C#)

Tratto

- Collezione di **metodi** che definiscono un comportamento che un **tipo** può implementare.

Tratti

- I tratti in Rust costituiscono l'equivalente delle interfacce di Java e C# o delle classi astratte pure in C++
 - **Un tratto definisce un insieme di metodi**, eventualmente associando loro un'implementazione di default
- 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
 - Tale costo si presenta solo quando si crea esplicitamente un riferimento dinamico (`&dyn TraitName`)

Definire ed 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
 - L'implementazione del tratto può essere fatta o dove viene definito il tratto o deve viene definita la struttura dati.

Definire ed 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}  
}
```

Definire ed usare un tratto

- 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 Salutabile {  
    fn saluta(&self);  
}
```

```
struct Persona { nome: String }  
struct Azienda { nome: String, numero: i32 }
```

```
impl Salutabile for Persona {  
    fn saluta(&self) {  
        println!("Ciao, sono {}!", self.nome);  
    }  
}  
impl Salutabile for Azienda {  
    fn saluta(&self) {  
        println!("Salve {} il tuo numero è {}!", self.nome, self.numero);  
    }  
}  
fn main() {  
    let persona = Persona { nome: String::from("Mario") };  
    let azienda = Azienda { nome: String::from("ItalComputer"), numero: 1247};  
    persona.saluta();  
    azienda.saluta();  
}
```

trait1.rs

```
trait HasArea {  
fn get_area(&self) -> f64;  
}  
  
struct Circle { radius: f64}  
impl HasArea for Circle {  
fn get_area(&self) -> f64 {  
    std::f64::consts::PI * (self.radius * self.radius)  
}  
}  
  
struct Rectangle { width: f64, height: f64, }  
impl HasArea for Rectangle {  
fn get_area(&self) -> f64  
{  
    self.width * self.height  
}  
}  
fn main() {  
let circle = Circle { radius: 3.0 };  
let rectangle = Rectangle { width: 4.0, height: 5.0 };  
println!("L'area del cerchio è: {}", circle.get_area());  
println!("L'area del rettangolo è: {}", rectangle.get_area());  
}
```



Definire ed usare un tratto

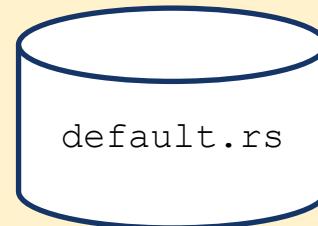
- Se una funzione tra quelle definite da un tratto non usa, come primo parametro, né **self** né un suo derivato (**&self**, **&mut self**, ...), questa non è legata all'istanza del tipo che la implementa
 - Può essere invocata usando come prefisso il nome del tratto o il nome del tipo che la implementa

```
trait Default {  
  
    fn default() -> Self  
  
}
```

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

```
#[derive(Default)]
struct Person {
    name: String,
    age: u32,
}

fn main() {
    // Utilizzando il valore predefinito fornito dal tratto Default
    let default_person: Person = Default::default();
    println!("Name: {}", default_person.name); // Stampa: Name:
    println!("Age: {}", default_person.age); // Stampa: Age: 0
}
```



```
trait Inizializzabile {
    fn inizializza() -> Self;
}

struct Punto {
    x: i32,
    y: i32,
}

impl Inizializzabile for Punto {
    fn inizializza() -> Self {
        Punto { x: 0, y: 0 }
    }
}

fn main() {
    let punto = Punto::inizializza();
    println!("Punto inizializzato: ({}, {})", punto.x, punto.y);
}
```



```
trait Inizializzabile {
    fn inizializza() -> Self;
}

#[derive(Default)]
struct Punto {
    x: i32,
    y: i32,
}

impl Inizializzabile for Punto {
    fn inizializza() -> Self {
        Punto { x: 0, y: 0 }
    }
}

fn main() {
    let punto = Punto::inizializza();
    let punto1:Punto = Default::default();
    println!("Punto inizializzato: ({}, {})", punto.x, punto.y);
    println!("Punto inizializzato con Default: ({}, {})", punto1.x, punto1.y);
}
```



```
trait Descrivibile {
    fn descrizione() -> &'static str;
}
#[derive(Debug)]
struct Libro {
    titolo: String,
    autore: String,
}
#[derive(Debug)]
struct Disco {
    titolo: String,
    cantante_gruppo: String,
    anno: i32
}
impl Descrivibile for Libro {
    fn descrizione() -> &'static str
    { "=> Questo è un libro" }
}
impl Descrivibile for Disco {
    fn descrizione() -> &'static str
    {"=> Questo è un disco"}
}
```



```
fn main() {
    let libro = Libro {
        titolo: String::from("Il Signore degli Anelli"),
        autore: String::from("J.R.R. Tolkien"),
    };
    let disco = Disco {
        titolo: String::from("Out of Time"),
        cantante_gruppo: String::from("R.E.M."),
        anno: 1991
    };
    println!("{} {}", libro, Libro::descrizione());
    println!("{} {}", disco, Disco::descrizione());
}
```

Definire ed usare un tratto

- 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!");  
}
```

```
trait Convertibile {  
    type Output;  
    fn converti(&self) -> Self::Output;  
}  
struct NumeroIntero { valore: i32}  
impl Convertibile for NumeroIntero {  
    type Output = f64;  
    fn converti(&self) -> Self::Output {  
        self.valore as f64 }  
}  
struct NumeroReale { valore: f64}  
impl Convertibile for NumeroReale {  
    type Output = i32;  
    fn converti(&self) -> Self::Output {  
        self.valore as i32 }  
}  
fn main() {  
    let numero = NumeroIntero { valore: 42 };  
    let valore_convertito: f64 = numero.converti();  
    println!("Valore convertito: {}", valore_convertito);  
  
    let numero = NumeroReale { valore: 42.3 };  
    let valore_convertito: i32 = numero.converti();  
    println!("Valore convertito: {}", valore_convertito);  
}
```



Definire ed usare un tratto

- 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  
}
```

```
trait Moltiplicabile {
    fn moltiplica(&self, factor: i32) -> i32;
    fn moltiplica_per_due(&self) -> i32 {
        self.moltiplica(2)
    }
}
struct Numero { valore: i32 }

impl Moltiplicabile for Numero {
    fn moltiplica(&self, factor: i32) -> i32 {
        self.valore * factor
    }

    fn moltiplica_per_due(&self) -> i32 {
        println!("Moltiplicazione per 2");
        self.moltiplica(2) // Ridefinizione del metodo con un comportamento diverso
    }
}

fn main() {
    let numero = Numero { valore: 5 };
    println!("Risultato: {}", numero.moltiplica_per_due());
}
```



```
trait Moltiplicabile {
    fn moltiplica(&self, factor: i32) -> i32;
    fn moltiplica_per_due(&self) -> i32 {
        self.moltiplica(2)
    }
}
struct Numero {valore: i32}
struct Numero2 { valore: i32, altro: i32 }
impl Moltiplicabile for Numero {
    fn moltiplica(&self, factor: i32) -> i32 {
        self.valore * factor
    }
    fn moltiplica_per_due(&self) -> i32 {
        println!("Moltiplicazione per 2");
        self.moltiplica(2) // Ridefinizione del metodo con un comportamento diverso
    }
}
impl Moltiplicabile for Numero2 {
    fn moltiplica(&self, factor: i32) -> i32 {
        self.valore * factor
    }
}
fn main() {
    let numero = Numero2 { valore: 5, altro: 1 };
    println!("Risultato: {}", numero.moltiplica_per_due());
}
```



Sotto-tratti e super-tratti

- 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 Sup(&self) {println!("In super");}  
    fn g(&self) {}  
}  
  
trait Subtrait: Supertrait {  
    fn Sub(&self) {println!("In sub");}  
    fn h(&self) {}  
}
```

```
struct SomeType;  
impl Supertrait for SomeType {}  
impl Subtrait for SomeType {}  
  
fn main() {  
    let s = SomeType;  
    s.Sup();  
    s.Sub();  
}
```

```

trait Animali { fn fanno_verso(&self); }

trait Mammiferi: Animali { fn si_cibano(&self); }
trait Uccelli: Animali { fn volano(&self); }

struct Cani;
struct Crow;

impl Animali for Cani {
fn fanno_verso(&self) { println!("Bau Bau"); }
}
impl Mammiferi for Cani {
fn si_cibano(&self) { println!("I cuccioli di cani sono allattati."); }
}

impl Animali for Crow {
fn fanno_verso(&self) { println!("Cra Cra"); }
}
impl Uccelli for Crow {
fn volano(&self) { println!("I corvi volano."); }
}

```



```

fn main() {
    let dog = Cani;
    let corvo = Crow;
    dog.fanno_verso();
    dog.si_cibano();
    corvo.fanno_verso();
    corvo.volano();
}

```

Sotto-tratti e super-tratti (II)

```
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);  
}
```

Invocare un tratto

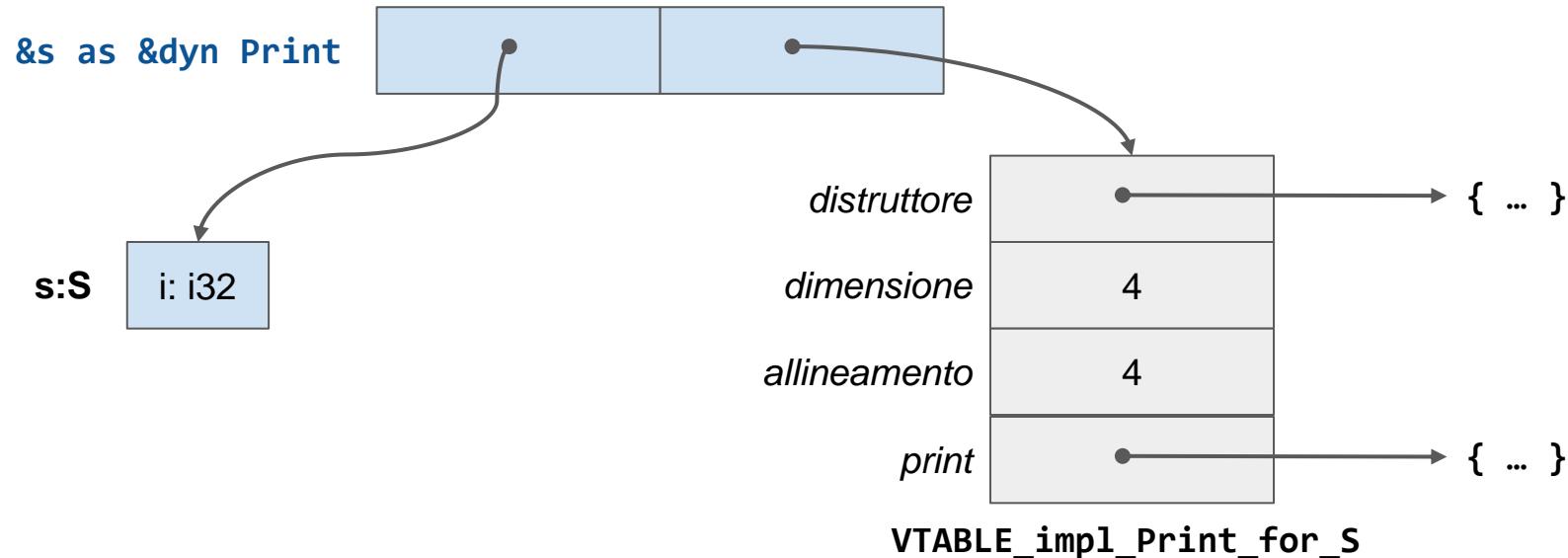
- L'invocazione dei metodi di un tratto può avvenire in due modalità distinte
 - Invocazione **statica**: se il tipo del valore è noto, il compilatore può identificare l'indirizzo della funzione da chiamare e generare il codice corrispondente senza alcuna penalità
 - Invocazione **dinamica**: se si dispone 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});  
}
```

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*



```
trait Shape { fn area(&self) -> f64; }

struct Circle { radius: f64}
impl Shape for Circle {
    fn area(&self) -> f64 { std::f64::consts::PI * self.radius * self.radius }
}

struct Rectangle {
    width: f64,
    height: f64
}
impl Shape for Rectangle {
    fn area(&self) -> f64 { self.width * self.height }
}

fn main() {
    let circle = Circle { radius: 5.0 };
    let rectangle = Rectangle { width: 3.0, height: 4.0 };

    let mut shapes: Vec<&dyn Shape> = Vec::new();
    shapes.push(&circle);
    shapes.push(&rectangle);
    for shape in shapes {
        println!("Area: {}", shape.area());
    }
}
```



```
trait Shape {
    fn area(&self) -> f64; }
trait Drawable: Shape { fn draw(&self); }

struct Rectangle {
    width: f64,
    height: f64,
}
impl Shape for Rectangle {
    fn area(&self) -> f64 { self.width * self.height }
}

impl Drawable for Rectangle {
    fn draw(&self) {
        println!("Drawing a rectangle with width {} and height {}.", self.width, self.height);
    }
}

struct Square { side: f64, }
impl Shape for Square {
    fn area(&self) -> f64 { self.side * self.side }
}

impl Drawable for Square {
    fn draw(&self) {
        println!("Drawing a square with side {}.", self.side);
    }
}
```

```
fn draw_shape(shape: &dyn Drawable) {
    shape.draw();
    println!("Area: {}", shape.area());
}

fn main() {
    let rectangle =
        Rectangle {width:3.0,height: 4.0};
    let square = Square { side: 2.0};

    draw_shape(&rectangle);
    draw_shape(&square);
}
```



Tratti nella libreria standard

- Rust definisce un insieme di tratti base la cui eventuale implementazione da parte di un dato tipo abilita una serie di scorciatoie sintattiche legate agli operatori presenti nel linguaggio
 - Sostituendo, a tutti gli effetti, il meccanismo di **operator overloading** presente nel linguaggio C++
- E' possibile confrontare due istanze di un dato tipo con gli operatori **==** e **!=** se il tipo implementa i tratti **Eq** o **PartialEq**
 - E' possibile un confronto basato su **<**, **>**, **<=** e **>=** se il tipo implementa i tratti **Ord** o **PartialOrd**
- Gli operatori binari **+**, **-**, *****, **/**, **%**, **&**, **^**, **<<**, **>>** sono rispettivamente associati ai tratti **Add**, **Sub**, **Mul**, **Div**, **Rem**, **BitAnd**, **BitXor**, **Shl**, **Shr**
- Le operazioni unarie **-** e **!** corrispondono a **Neg** e **Not**

```

use std::ops::Add;

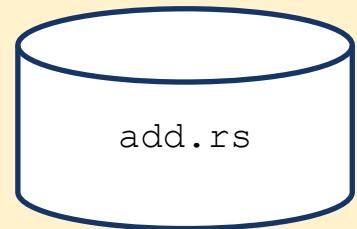
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self::Output {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 0 };
    let p2 = Point { x: 2, y: 3 };
    let p3 = p1 + p2; // Usa l'implementazione di Add
    println!("p3.x: {}, p3.y: {}", p3.x, p3.y);
}

```



Gestire i confronti di uguaglianza

- Implementare un confronto di uguaglianza richiede il puntuale rispetto di alcune proprietà logiche essenziali
 - Riflessività, simmetria, transitività
- Rust introduce due tratti per esprimere questa capacità
 - **PartialEq** ed **Eq**, entrambi definiti dai metodi `eq(&self, other: &RHS) -> bool` e `ne(&self, other: &RHS) -> bool`
 - **PartialEq** richiede che siano garantite, dall'implementazione, sia la proprietà simmetrica sia quella transitiva
 - **Eq** (che è un sotto-tratto di **PartialEq**) impone anche il rispetto della proprietà riflessiva: i tipi floating-point (f32 e f64) NON implementano questo tratto (NaN != NaN)
 - Il tratto **Eq** è un tratto senza metodi e la sua dichiarazione solo per specificare che l'uguaglianza gode anche della proprietà riflessiva
- Il tipo associato **RHS (Right Hand Side)** 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(...)**

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl PartialEq for Point {
    fn eq(&self, other: &Self) -> bool {
        self.x == other.x && self.y == other.y
    }
}

fn main() {
    let point1 = Point { x: 1, y: 2 };
    let point2 = Point { x: 1, y: 2 };
    let point3 = Point { x: 3, y: 4 };

    println!("point1 == point2: {}", point1 == point2); // true
    println!("point1 == point3: {}", point1 == point3); // false
}
```



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
    }
}
```

```
#[derive(Debug, PartialEq, Eq)]
struct Point {
    x: i32
    y: i32,
}

fn main() {
    let point1 = Point { x: 1, y: 2 };
    let point2 = Point { x: 1, y: 2 };
    let point3 = Point { x: 3, y: 4 };

    println!("point1 == point2: {}", point1 == point2); // true
    println!("point1 == point3: {}", point1 == point3); // false
}
```

2 Point sono uguali se tutti i corrispondenti valori dei loro campi sono uguali

partialeq2.rs

```
struct FloatingPointNumber {
    value: f64,
}

impl PartialEq for FloatingPointNumber {
    fn eq(&self, other: &Self) -> bool {
        let epsilon = 0.00001;
        let difference = (self.value - other.value).abs();
        difference <= epsilon
    }
}

fn main() {
    let a = FloatingPointNumber { value: 1.0 };
    let b = FloatingPointNumber { value: 1.01 };
    let c = FloatingPointNumber { value: 1.00000001 };

    println!("a == b: {}", a == b);
    println!("a == b: {}", a == c);
}
```



```
#[derive(PartialEq)]  
  
struct FloatingPointNumber {  
    value: f64,  
}  
  
fn main() {  
    let a = FloatingPointNumber { value: 1.0 };  
    let b = FloatingPointNumber { value: 1.01 };  
    let c = FloatingPointNumber { value: 1.000000001 };  
    let d = FloatingPointNumber { value: 1.000000001 };  
  
    println!("a == b: {}", a == b);  
    println!("a == c: {}", a == c);  
    println!("c == d: {}", c == d);  
}
```



Gestire i confronti di uguaglianza

- **PartialEq** ed **Eq**, entrambi definiti dai metodi:
 - `eq(&self, other: &Rhs) -> bool`
 - `ne(&self, other: &Rhs) -> bool`
- Il tipo associato **Rhs (Right Hand Side)** 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

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl PartialEq<i32> for Point {
    fn eq(&self, other: &i32) -> bool {
        self.x == *other && self.y == *other
    }
}

fn main() {
    let point = Point { x: 5, y: 5 };

    // Confronto tra un punto e un valore i32
    println!("point == 5: {}", point == 5); // true
}
```



partialeq5.rs

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;
}
```

```
#[derive(Debug, PartialEq)]
struct Point {
    x: f64,
    y: f64,
}

impl PartialOrd for Point {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        if self.x < other.x && self.y < other.y {
            Some(std::cmp::Ordering::Less)
        } else if self.x > other.x && self.y > other.y {
            Some(std::cmp::Ordering::Greater)
        } else if self.x == other.x && self.y == other.y {
            Some(std::cmp::Ordering::Equal)
        } else {
            None // In caso di confronto non definito
        }
    }
}

fn main() {
    let point1 = Point { x: 1.0, y: 4.0 };
    let point2 = Point { x: 3.0, y: 2.0 };

    match point1.partial_cmp(&point2) {
        Some(ordering) => { println!("point1 è {:?} di point2", ordering); }
        None => { println!("Confronto non definito tra point1 e point2"); }
    }
}
```

partialord1.rs

```
#[derive(Debug, PartialEq, PartialOrd)]
struct Point {
    x: f64,
    y: f64,
}

fn main() {
    let point1 = Point { x: 1.0, y: 2.0 };
    let point2 = Point { x: 3.0, y: 4.0 };

    println!("point1 < point2: {}", point1.lt(&point2));
    println!("point1 > point2: {}", point1.gt(&point2));
}
```



```
#[derive(Debug, Eq, PartialEq, PartialOrd)]  
  
struct Point {  
    x: i32,  
    y: i32,  
}  
impl Ord for Point {  
    fn cmp(&self, other: &Self) -> std::cmp::Ordering {  
        if self.x != other.x {  
            self.x.cmp(&other.x)  
        } else {  
            self.y.cmp(&other.y)  
        }  
    }  
}  
fn main() {  
    let point1 = Point { x: 3, y: 5 };  
    let point2 = Point { x: 3, y: 8 };  
  
    let max_point = point1.max(point2);  
    println!("Il punto con coordinate massime è {:?}", max_point);  
}
```



```

use std::cmp::Ordering;

#[derive(Debug, Eq, PartialEq, PartialOrd)]
struct Person {
    name: String,
    age: i32,
}

impl Ord for Person {
    fn cmp(&self, other: &Self) -> Ordering {
        other.age.cmp(&self.age)
    }
}

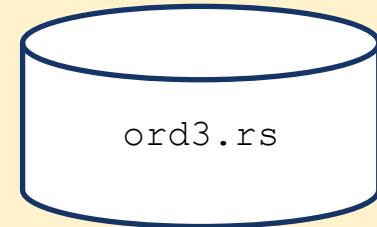
fn main() {
    let john = Person { name: String::from("John"), age: 25 };
    let emma = Person { name: String::from("Emma"), age: 23 };
    let peter = Person { name: String::from("Peter"), age: 25 };

    println!("John is {:?}, compared to Emma.", john.cmp(&emma));
        // Stampa: John is Greater, compared to Emma.
    println!("John is {:?}, compared to Peter.", john.cmp(&peter));
        // Stampa: John is Equal, compared to Peter.
}

```



```
fn main() {  
  
    let min = 50;  
    let max = 100;  
  
    let num1 = 200;  
    let num2 = 25;  
  
    let clamped_number = num1.clamp(min, max);  
    println!("Clamped number: {}", clamped_number); // Stampa: Clamped number: 100  
  
    let clamped_number = num2.clamp(min, max);  
    println!("Clamped number: {}", clamped_number); // Stampa: Clamped number: 50  
  
}
```



Visualizzare i contenuti

- 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**
 - Da un punto di vista pratico, la sua implementazione richiede la definizione di un solo metodo

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

- Il tratto **Debug** ha la stessa firma di **Display**
 - Il suo scopo, tuttavia, è differente: creare una rappresentazione comprensibile al programmatore del valore
 - Viene attivato usando il segnaposto **{:?}**
 - Può essere sintetizzato automaticamente attraverso l'annotazione **#[derive(Debug)]**

Copia e duplicazione

- Il tratto **Clone** indica la capacità di un tipo di creare un duplicato di un valore dato
 - Questo può essere usato per trasformare un riferimento ($\&T$) in un valore posseduto (T)
 - Tale trasformazione, ovviamente, può essere molto costosa in termini di tempo e memoria, tuttavia è sempre sotto il controllo del programmatore perché tale funzionalità non è mai attivata automaticamente
- Il tratto **Copy** è un sotto-tratto di **Clone**
 - Esso implica il fatto che la copia ottenuta sia - a tutti gli effetti - il solo duplicato dei bit presenti nel valore originale, senza nessuna trasformazione
 - La sua implementazione può essere veloce ed efficiente (si basa sulla funzione `memcpy(...)`)
 - Questo tratto può essere implementato solo da strutture dati i cui campi implementino il tratto stesso
- 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**

Rilasciare le risorse

- Il tratto **Drop** permette di definire il metodo **drop(&mut self)** che fa le veci del distruttore nel linguaggio C++
 - Il compilatore Rust garantisce che tale metodo sarà chiamato nel momento in cui la variabile che contiene un valore di questo tipo uscirà dal proprio scope sintattico o subito prima che le venga assegnato un nuovo valore
 - Questo permette di rilasciare eventuali risorse possedute dalla struttura dati o abilitare comportamenti duali a quelli della creazione dell'oggetto, tipici del pattern RAI - Resource Acquisition Is Initialization
- Questo tratto è mutuamente esclusivo con il tratto **Copy**
 - Questo vincolo elimina, in assenza di blocchi **unsafe { ... }**, la possibilità di avere un doppio rilascio delle risorse presenti sullo heap
- Questo tratto 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 una struttura dati

- E' possibile utilizzare **sintatticamente** una struttura dati come fosse un array, abilitando la notazione **t[i]** se si implementano i tratti **Index** e **IndexMut**
 - L'espressione **t[i]** viene riscritta dal compilatore come ***t.index(i)**, se si accede in lettura al risultato dell'espressione, o come ***t.index_mut(i)**, se si accede in scrittura
- Questi due tratti sono definiti nel seguente modo

```
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;
}
```

```
use std::ops::Index;
struct MyVector<T> {
    data: Vec<T>,
    matricola: i32
}
impl<T> MyVector<T> {
    fn new(matr: i32) -> MyVector<T> {
        MyVector { data: Vec::new(),
                   matricola: matr}
    }
    fn push(&mut self, item: T) {self.data.push(item);}
}
impl<T> Index<usize> for MyVector<T> {
    type Output = T;
    fn index(&self, index: usize) -> &Self::Output {
        print!("Accesso at index {} => ", index);
        &self.data[index]
    }
}
fn main() {
    let mut my_vector = MyVector::new(250_000);
    my_vector.push(23);
    my_vector.push(25);
    println!("{} {}", my_vector[0]);
    println!("{} {}", my_vector[1]);
}
```



```

use std::ops::{Index, IndexMut};

#[derive(Debug)]
struct StringSet {
    data: Vec<String>,
}

impl StringSet {
    fn new() -> Self {
        StringSet { data: Vec::new() }
    }
    fn add(&mut self, s: String) {self.data.push(s)}
}
impl Index<usize> for StringSet {
    type Output = String;

    fn index(&self, index: usize) -> &Self::Output {
        &self.data[index]
    }
}
impl IndexMut<usize> for StringSet {
    fn index_mut(&mut self, index: usize) -> &mut Self::Output {
        &mut self.data[index]
    }
}

```



```

fn main() {
    let mut string_set=StringSet::new();
    string_set.add("apple".to_string());
    string_set.add("banana".to_string());
    string_set.add("orange".to_string());
    println!("Original {:?}",string_set);

    string_set[1] = "grape".to_string();
    println!("Modified {:?}",string_set);
}

```

Indicizzare una struttura dati

- Varie classi della libreria standard implementano i tratti Index e IndexMut
 - Poiché sia l'indice usato che il tipo associato restituito dall'operazione di indicizzazione sono parametrizzati, è lecito avere implementazioni multiple del tratto per un dato tipo
 - Vec<T>, ad esempio, consente di definire sia indici numerici (usize) che restituiscono valori di tipo T, che indici di tipo intervallo (Range) che restituiscono Slice<T>

```
// 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]);
```

```

use std::ops::Index;
struct MyVec{
    data: Vec<i32>,
    year: i32
}
impl MyVec {
    fn new(data: Vec<i32>) -> Self {
        MyVec{data: data, year: 2024}
    }
}
impl Index<std::ops::Range<usize>> for MyVec {
    type Output = [i32];

    fn index(&self, index: std::ops::Range<usize>) -> &Self::Output {
        &self.data[index]
    }
}
fn main() {
    let my_vec = MyVec::new(vec![1, 2, 3, 4, 5]);

    // Accesso tramite intervallo di indici
    let slice = &my_vec[1..4];
    println!("Slice: {:?}", slice); // Stampato: [2, 3, 4]
}

```



Dereferenziare un valore

- E' possibile trattare una struttura dati come se fosse sintatticamente un puntatore implementando i tratti **Deref** e **DerefMut**
 - 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;  
}
```



Dereferenziare un valore

- Il tipo `Self::Target` deve essere qualcosa contenuto, posseduto o referenziato in `Self`
 - Nel caso di `Box<T>`, `Target` è il tipo `T` contenuto nel `Box` e allocato sullo heap
- La notazione `*t` prende a prestito, in modo condiviso o esclusivo, il valore `t` per poter eseguire il metodo del tratto corrispondente
 - I riferimenti restituiti da `deref(&self)` o da `deref_mut(&mut self)` prolungano, come conseguenza delle regole sul tempo di vita, il prestito di `self` fino al termine del loro utilizzo
- I due tratti svolgono anche un secondo importante ruolo: permettono la conversione automatica dal tipo `&Self` al tipo `&Self::Target` e da `&mut Self` a `&mut Self::Target`. Questa proprietà viene detta ***deref coercion***
 - Questo meccanismo non viene applicato nella risoluzione di tipi generici

```
use std::ops::Deref;
#[derive(Debug)]
struct CustomStruct {
    number: i32,
    boxed_value: Box<i32>,
}

impl Deref for CustomStruct {
    type Target = i32;

    fn deref(&self) -> &Self::Target {
        &self.number
    }
}
fn main() {
    let boxed_value = Box::new(42);
    let custom_struct = CustomStruct { number: 10, boxed_value };

    let custom_struct_ref = &custom_struct;

    println!("Dereferencing custom_struct: {:?}", *custom_struct_ref);
        // Prints: Dereferencing custom_struct: 10
    println!("Dereferencing boxed_value: {:?}", *custom_struct_ref.boxed_value);
        // Prints: Dereferencing boxed_value: 42
}
```



Deref Coercion

- Consente l'interoperabilità, ad esempio, tra &String e &str, per cui è lecito invocare su un valore di tipo String metodi definiti per str

```
fn main() {  
    let mut s: String = "Hello, World!".to_string();  
    let mut s_ref: &str = &s; // Deref coercion: &String -> &str  
  
    println!("{}", s_ref); // Prints: Hello, World!  
}
```



Dereferenziare un valore

```
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”);
```



Deref Coercion

- Il compilatore esegue la conversione implicita per chiamare il metodo deref, restituendo il tipo Target.

```
fn main() {  
  
    let b:Box<i32> = Box::new(10);  
  
    println!("{}", b);  
}
```



```
use std::ops::Deref;

struct MyData {value: i32,}

struct Container {data: Box<MyData>,}

impl Deref for Container {
    type Target = MyData;

    fn deref(&self) -> &Self::Target {
        &self.data
    }
}

fn main() {
let my_data = MyData { value: 42 };

let container = Container { data: Box::new(my_data) };

// dereferenziazione automatica per accedere al valore all'interno di Container
println!("Value inside Container: {}", container.value);
}
```



```
use std::ops::{Deref, DerefMut};

#[derive(Debug)]
struct MyBox<T>(T);

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

impl<T> DerefMut for MyBox<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}

fn main() {
    let mut my_box = MyBox(5);

    // Usando deref coercion: converte automaticamente &mut MyBox<i32> a &mut i32
    *my_box += 1;

    println!("Valore dopo l'incremento: {:?}", my_box);
}
```





deref6.rs

```
fn main() {  
    let mut my_string = String::from("Hello");  
  
    // Utilizzo del dereferenziamento mutevole per modificare la stringa  
    let my_string_ref: &mut String = &mut my_string;  
    (*my_string_ref).push_str(", world!");  
  
    println!("{}", my_string_ref);  
}
```

```
use std::ops::{Deref, DerefMut};

struct CustomString { value: String}

impl Deref for CustomString {
    type Target = String;

    fn deref(&self) -> &Self::Target {
        &self.value
    }
}

impl DerefMut for CustomString {
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.value
    }
}

fn main() {
    let mut custom_string = CustomString {value: "Original value".to_string(),};

    let mut custom_str_ref = custom_string.deref_mut();
    *custom_str_ref = "Mutated value".to_string();

    println!("{}", custom_string.value); // Prints: Mutated value
}
```



Definire un intervallo

Approfondimento

- Attraverso l'utilizzo dell'operatore .. è possibile definire intervalli di valori per tutti i tipi che implementano il trait **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,
}
```

Approfondimento

```
use std::ops::RangeBounds;
fn main() {
    let range = 1..; // Definizione di un range
    let end = range.end_bound(); // Ottenere il limite superiore del range

    match end {
        std::ops::Bound::Included(&upper) => println!("Limite superiore incluso:{}" , upper),
        std::ops::Bound::Excluded(&upper) => println!("Limite superiore escluso:{}" , upper),
        std::ops::Bound::Unbounded => println!("Il range non ha limite superiore"),
    }
}
```

rangel.rs

```

use std::ops::{Bound, RangeBounds};

struct MyRange<T> {
    start: T,
    end: T,
}

impl<T> RangeBounds<T> for MyRange<T> where T: Copy {
    fn start_bound(&self) -> Bound<&T> {
        Bound::Included(&self.start)
    }
    fn end_bound(&self) -> Bound<&T> {
        Bound::Excluded(&self.end)
    }
}
fn main() {
    let my_range = MyRange { start: 1, end: 5 };

    match my_range.end_bound() {
        Bound::Included(&upper) => println!("Il limite superiore è incluso: {}", upper),
        Bound::Excluded(&upper) => println!("Il limite superiore è escluso: {}", upper),
        Bound::Unbounded => println!("Il range non ha limite superiore"),
    }
}

```

Approfondimento

range2.rs

Approfondimento



```
fn main() {
    let range = 0..10;

    println!("Does the range 0..10 contain 5? {}", range.contains(&5)); // true
    println!("Does the range 0..10 contain 10? {}", range.contains(&10)); // false

    let inclusive_range = 0..=10;
    println!("Does the inclusive range 0..=10 contain 10? {}", inclusive_range.contains(&10));
    // true
}
```

range3.rs

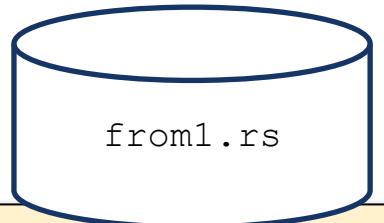
Conversione tra tipi

- 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

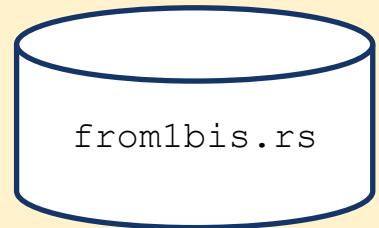
```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
impl From<(i32, i32)> for Point {  
    fn from((x, y): (i32, i32)) -> Self {  
        Point { x, y }  
    }  
}  
  
impl From<[i32; 2]> for Point {  
    fn from([x, y]: [i32; 2]) -> Self {  
        Point { x, y }  
    }  
}
```



```
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();  
}
```

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32, }
impl From<(i32, i32)> for Point {
    fn from((x, y): (i32, i32)) -> Self { Point { x, y } }
}
impl From<Point> for (i32, i32) {
    fn from(Point { x, y }: Point) -> Self { (x, y) }
}
impl From<[i32; 2]> for Point {
    fn from([x, y]: [i32; 2]) -> Self { Point { x, y } }
}
impl From<Point> for [i32; 2] {
    fn from(Point { x, y }: Point) -> Self { [x, y] }
}
fn main () {
    let point = Point::from((0, 0));
    let point: Point = (0, 0).into();
    let tuple: (i32, i32) = point.into();
    let point = Point::from([1, 1]);
    let point: Point = [0, 0].into();
    let array = <[i32; 2]>::from((2,2));

    println!(" {:?} {:?} {:?}", point, tuple, array);
}
```

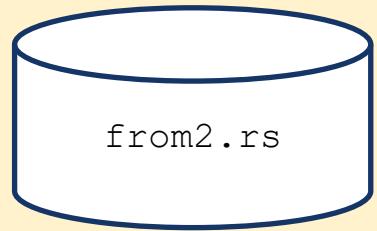


```
struct Centimeters(f64);

// Definiamo una struttura Inches
struct Inches(f64);

// Implementiamo il tratto From per convertire da Inches a Centimeters
impl From<Inches> for Centimeters {
    fn from(inches: Inches) -> Self {
        Centimeters(inches.0 * 2.54)
    }
}

fn main() {
    let inches = Inches(10.0);
    let centimeters: Centimeters = Centimeters::from(inches);
    println!("10 pollici sono equivalenti a {:.2} centimetri.", centimeters.0);
}
```

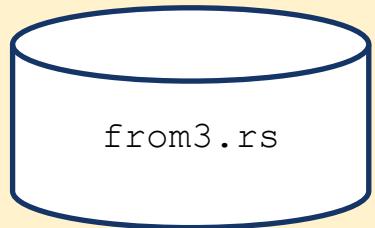


```
struct Person {
    name: String,
    age: u32,
}

struct Employee {
    name: String,
    position: String,
}

impl From<Person> for Employee {
    fn from(person: Person) -> Self {
        Employee {
            name: person.name,
            position: String::from("Sviluppatore"),
        }
    }
}

fn main() {
    let person = Person {
        name: String::from("Mario Rossi"),
        age: 30,
    };
    let employee: Employee = Employee::from(person);
    println!("Nome: {}", employee.name);
    println!("Posizione: {}", employee.position);
}
```



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>;  
}
```

```
use std::convert::TryFrom;
struct PositiveInteger(u32);
impl TryFrom<u32> for PositiveInteger {
    type Error = &'static str;
    fn try_from(value: u32) -> Result<Self, Self::Error> {
        if value > 0 {
            Ok(PositiveInteger(value))
        } else {
            Err("il valore deve essere maggiore di zero")
        }
    }
}
fn verifica (value: u32, result: Result<PositiveInteger, &str>)
{   match result {
        Ok(pos_int) => println!("Valore positivo: {}", pos_int.0),
        Err(err) => println!("Errore: ho ricevuto {} {}", value, err),
    }
}
fn main() {
    let valid_value = 42;
    let result_valid = PositiveInteger::try_from(valid_value);
    verifica(valid_value, result_valid);
    let invalid_value = 0;
    let result_invalid = PositiveInteger::try_from(invalid_value);
    verifica(invalid_value, result_invalid);
}
```



```
use std::convert::TryFrom;
struct EvenNumber(i32);
impl TryFrom<i32> for EvenNumber {
    type Error = &'static str;

    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value % 2 == 0 {
            Ok(EvenNumber(value))
        } else {
            Err("il numero non è pari.")
        }
    }
}
fn verifica (value: i32, result: Result<EvenNumber, &str>)
{
    match result {
        Ok(even) => println!("Numero pari: {}", even.0),
        Err(err) => println!("Errore: ho ricevuto {}, ma {}", value, err),}
}
fn main() {
    let even_number = 42;
    let result = EvenNumber::try_from(even_number);
    verifica(even_number, result);
    let odd_number = 37;
    let result_odd = EvenNumber::try_from(odd_number);
    verifica(odd_number, result_odd);
}
```



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>**

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

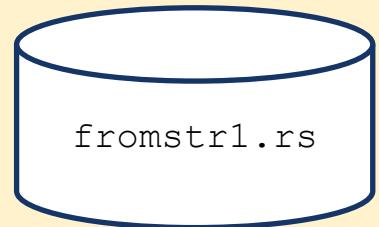
```
use std::num::ParseIntError;

use std::str::FromStr;

fn verifica (value: &str, result: Result<i32, ParseIntError>)
{
    match result {
        Ok(num) => println!("Numero: {}", num),
        Err(err) => println!("Errore: ho ricevuto {} {}", value, err),
    }
}

fn main() {
    // Utilizziamo il metodo parse() fornito dal trait FromStr
    // per convertire la stringa in un numero intero
    let number_str = "42";
    let number: Result<i32, ParseIntError> = number_str.parse();
    verifica(number_str, number);

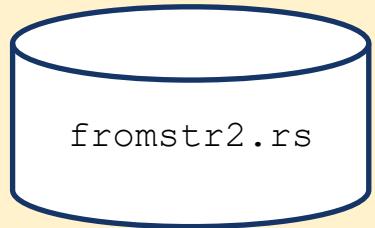
    let number_str2 = "quarantadue";
    let number2: Result<i32, ParseIntError> = number_str2.parse();
    verifica(number_str2, number2);
}
```



```
use std::str::FromStr;

fn main() {
    let string_boolean = "true".to_string();
    let result: Result<bool, _> = bool::from_str(&string_boolean);

    match result {
        Ok(value) => println!("String converted to boolean: {}", value),
        Err(_) => println!("Failed to convert string to boolean"),
    }
}
```



Scrivere un errore

- In Rust gli errori vengono 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 trait **Error**

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

```
use std::error::Error;
use std::fmt;

// Definiamo un tipo di errore personalizzato
#[derive(Debug)]
struct CustomError {
    message: String,
}
impl fmt::Display for CustomError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.message)
    }
}
impl Error for CustomError{} // Funzione che restituisce un errore personalizzato
fn do_something() -> Result<(), Box
```



Tipi generici

- Un sistema di tipi stringente facilita la creazione di codice robusto
 - Identificando, in fase di compilazione, quei frammenti di codice che violano il sistema dei tipi
- In certe situazioni, per non violare il sistema dei tipi, occorre replicare una grande quantità di codice generando problemi in fase di manutenzione
 - Occorre, infatti, garantire che eventuali modifiche ad una versione del codice vengano propagate a tutte le altre
- Sia il C++ che Rust permettono di estendere il sistema dei tipi utilizzando una forma di meta-programmazione (detta *template programming* in C++ e *generics* in Rust) grazie alla quale è possibile descrivere strutture dati e funzioni che contengono dati (o che operano su dati) il cui tipo è rappresentato da una meta-variabile
 - Permettendo così di esprimere dei concetti più generali

Funzioni generiche

- In C++, come in Rust, si può definire una funzione in modo che operi su un tipo di dato non ancora precisato

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

C++

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

Rust

Funzioni generiche

- Nel caso del C++, la funzione definita può operare con qualsiasi tipo di dato, a patto che supporti:
 - L'operatore “`<`” tra argomenti omogenei
 - Il costruttore di copia, per trasformare l'argomento ricevuto (che potrebbe essere il risultato di un'espressione) in un valore posseduto dal parametro formale
- Nel caso di Rust, i vincoli sono più esplicativi
 - Il tipo T che può essere passato alla funzione è soggetto al vincolo di implementare il trait Ord
 - Il Borrow Checker si occupa di garantire che, se viene passato un valore, questo venga correttamente gestito dal punto di vista del possesso
- Nel punto in cui una funzione generica viene invocata, il compilatore provvede a dedurre cosa debba essere sostituito al segnaposto T per rendere accettabile il codice e genera una versione specializzata della funzione per tale tipo, se non è già stata generata (**monomorfizzazione**).

Template C++

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

int i = max(10, 20); // T → int

std::string s, s1 = ..., s2 = ...;
s = max(s1, s2);      // T → std::string

max(2, 3.141593);    // ERRORE
// il compilatore non sa cosa scegliere!
max<float>(2, 3.141593) // T → float
```

```
template <typename T>
class wrapper {
    T data;
public:
    wrapper(T d): data(d) {}
    T get() { return data; }
};

// le classi richiedono ESPLICITAMENTE
// il tipo
wrapper<int> w1(1);           // T → int

wrapper<const char*> w2("hello");
// T → const char*
```

- A partire dalla versione C++20, la parola chiave “`typename`” può essere sostituita con il nome di un concept
 - Insieme di vincoli esplicativi sul tipo generico che limitano i tipi concreti che possono essere sostituiti alla meta-variabile, riprendendo in buona misura il meccanismo dei tratti di Rust

Concetti in C++

```
#include <concepts>
#include <vector>
template <typename T>
concept incrementable = requires(T t) { ++t; } && std::copyable<T>;  
  
template <incrementable T>
class Counter {
    T val;
public:
    Counter(T start): val(start) {}
    T increment() { return ++val; }
    T value() { return val; }
};  
  
Counter<int> c1(0);           // T → int
```

Tipi generici in Rust

- Come nel caso del C++, anche in Rust si possono usare costrutti generici per generalizzare i parametri / tipo di ritorno di una funzione o per definire un tipo composto (struct, tupla, enum) basato su parti variabili
 - Si definisce una funzione generica basata sulle meta-variabili T, U, V, ... indicando tali meta-variabili all'interno di parentesi angolari < > dopo il nome della funzione e prima dell'elenco dei parametri formali
 - Ciascuna meta-variabile può essere soggetta ad eventuali restrizioni, indicate come l'insieme dei tratti che deve implementare e/o come il tempo di vita e deve garantire
- Se una struttura generica implementa dei metodi, i nomi delle meta-variabili ed i vincoli cui sono soggette vengono ripetuti nel blocco impl

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

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

Tipi generici (in generale)

- 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**
- L'uso di tipi generici facilita il processo di astrazione da parte del programmatore
 - Affidando al compilatore il compito di rifinire i dettagli necessari a gestire specifici tipi di dato

Trait Bound

- Rust offre due modi per specificare i vincoli sui tipi generici
 - Una versione compatta `<T: SomeTrait>`
 - La versione estesa `<T> ... where T: SomeTrait`
- In entrambi i casi, se è necessario indicare che il tipo deve implementare più tratti, questi possono essere combinati con il segno `+`

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

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

```
fn get_value<T>(condition: bool, value: T) -> Option<T> {
    if condition {
        Some(value)
    } else {
        None
    }
}
```

```
fn main() {
    let condition = true;
    let value = "Hello, world!";
    match get_value(condition, value) {
        Some(v) => println!("Value: {}", v),
        None => println!("No value found"),
    }
}
```

generic1.rs

```
fn duplicate<T: Clone>(a: T) -> (T, T) {  
    (a.clone(), a.clone())  
}
```

```
fn main() {  
    let bau = String::from("bau");  
    let pair = duplicate(bau);  
    println!("{}:{}", pair);  
}
```

generic2.rs

```
fn sum<T: std::ops::Add<Output = T> + Default + Copy> (items: &[T]) -> T {
    let mut result = T::default();
    for item in items {
        result = result + *item;
    }
    result
}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    println!("Somma: {}", sum(&numbers));

    let floats = vec![1.1, 2.2, 3.3, 4.4, 5.5];
    println!("Somma: {}", sum(&floats));
}
```



```
fn max<T: PartialOrd>(items: &[T]) -> Option<&T> {
    if items.is_empty() {
        None
    } else {
        let mut max = &items[0];
        for item in items.iter() {
            if item > max { max = item; }
        }
        Some(max)
    }
}

fn my_max <T: std::fmt::Display> (value: Option<&T>)
{ match value {
    Some(max) => println!("Massimo: {}", max),
    None => println!("Sequenza vuota")
}
}

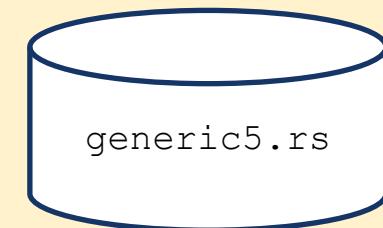
fn main() {
    let numbers = vec![1, 3, 5, 2, 4];
    my_max(max(&numbers));
    let strings = vec!["apple", "banana", "orange", "grape"];
    my_max(max(&strings));
}
```



```
#[derive(Debug)]
struct Pair<T, U> {
    first: T,
    second: U,
}
impl<T, U> Pair<T, U> {
    fn new(first: T, second: U) -> Self {
        Pair { first, second }
    }
}
fn main() {
    // Crea una coppia di interi
    let integer_pair = Pair::new(10i32, 20i32);
    println!("Integer pair: {:?}", integer_pair);

    // Crea una coppia di stringhe
    let string_pair = Pair::new("Hello".to_string(), "World".to_string());
    println!("String pair: {:?}", string_pair);

    // Crea una coppia di valori misti
    let mixed_pair = Pair::new(10i32, "Hello".to_string());
    println!("Mixed pair: {:?}", mixed_pair);
}
```



```
trait Calcolabile { fn calcola(&self) -> f64; }
struct Cerchio { raggio: f64, }
struct Quadrato { lato: f64, }
impl Calcolabile for Cerchio {
    fn calcola(&self) -> f64 {
        std::f64::consts::PI * self.raggio * self.raggio
    }
}
impl Calcolabile for Quadrato {
    fn calcola(&self) -> f64 {
        self.lato * self.lato
    }
}
fn calcola_area<T: Calcolabile>(forma: T) -> f64 { forma.calcola() }
fn main() {
    let cerchio = Cerchio { raggio: 5.0 };
    let quadrato = Quadrato { lato: 4.0 };
    println!("Area del cerchio: {}", calcola_area(cerchio));
    println!("Area del quadrato: {}", calcola_area(quadrato));
}
```



Tratti e tipi generici

- Sebbene tratti e tipi generici siano due modi per implementare il polimorfismo, tra i due esiste un legame molto più profondo
 - Uno o più tratti possono essere usati come vincoli per limitare l'utilizzo di un tipo generico ai soli casi in cui ha senso farlo
 - Si può definire un **tratto generico**, i cui metodi, cioè, ricevono o restituiscono valori generici, eventualmente vincolati da ulteriori tratti.

```
trait Confrontabile<T> {
    fn confronta(&self, other: &T) -> bool;
}

// Implementazione del traito Confrontabile per i tipi di dati numerici
impl<T> Confrontabile<T> for T
where
    T: std::cmp::PartialOrd,
{
    fn confronta(&self, other: &T) -> bool {
        self >= other
    }
}

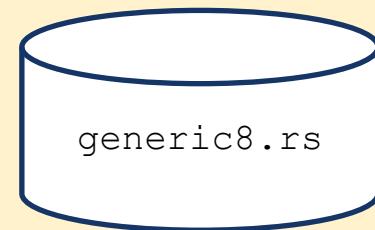
fn main() {
    let x = 5;
    let y = 10;
    println!("x è maggiore o uguale a y? {}", x.confronta(&y));
}
```



Tratti e tipi generici

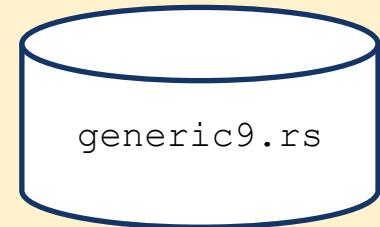
- Occorre comprendere le somiglianze e le differenze tra una funzione non generica che opera su oggetto-tratto e un'altra generica il cui parametro è vincolato da un tratto e scegliere quando usare una forma o l'altra
 - `fn dynamic_process(w: &mut dyn Write) { ... }`
 - `fn generic_process<T>(w: &mut T) where T:Write { ... }`
- Gli oggetti-tratto richiedono l'uso di un *double 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
 - Non solo perché le chiamate alle funzioni non necessitano di transitare per la VTABLE, ma perché 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
 - Occorre infatti che il tratto non definisca alcun metodo statico (ovvero che non utilizza `self`, `&self`, ..., come primo parametro)
- Non è possibile definire un oggetto-tratto legato a più tratti disgiunti
 - Mentre è possibile, in una funzione generica, vincolare una meta-variabile ad implementare più tratti

```
trait Suona { fn suona(&self); }
struct Chitarra;
struct Pianoforte;
impl Suona for Chitarra {
    fn suona(&self) {
        println!("Chitarra: Strum!");
    }
}
impl Suona for Pianoforte {
    fn suona(&self) {
        println!("Pianoforte: Ding!");
    }
}
fn esegui_melodia<T: Suona>(strumento: & T) { strumento.suona(); }
fn main() {
    let chitarra = Chitarra;
    let pianoforte = Pianoforte;
    esegui_melodia(&chitarra);
    esegui_melodia(&pianoforte);
}
```



```
trait Suona {
    fn suona(&self);
}

struct Chitarra;
struct Pianoforte;
impl Suona for Chitarra {
    fn suona(&self) {
        println!("Chitarra: Strum!");
    }
}
impl Suona for Pianoforte {
    fn suona(&self) {
        println!("Pianoforte: Ding!");
    }
}
fn esegui_melodia(strumento: &dyn Suona) { strumento.suona(); }
fn main() {
    let chitarra = Chitarra;
    let pianoforte = Pianoforte;
    esegui_melodia(&chitarra);
    esegui_melodia(&pianoforte);
}
```





Per saperne di più

- What is generic programming?
 - <https://oswalt.dev/2020/08/what-is-generic-programming/>
- Using generic types in Rust
 - <https://oswalt.dev/2021/06/using-generic-types-in-rust/>
- Polymorphism in Rust
 - <https://oswalt.dev/2021/06/polymorphism-in-rust/>
- Rust Traits: Defining Behavior
 - <https://oswalt.dev/2020/07/rust-trait-defining-behavior/>
- Tour of Rust's standard library traits
 - <https://github.com/pretzelhammer/rust-blog/blob/master/posts/tour-of-rusts-standard-library-traits.md>