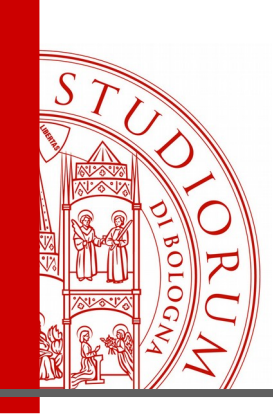


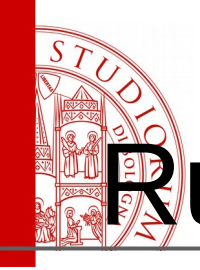
12 Rust

Rust

- 2010, Mozilla foundation (sviluppo iniziato nel 2006 da dipendente Mozilla)
- primo compilatore scritto in Ocaml; ora compilato in rust stesso; LLVM come backend
- “a language written by geniuses for geniuses” (come Haskell)
- forti influenze da OCaml, C++

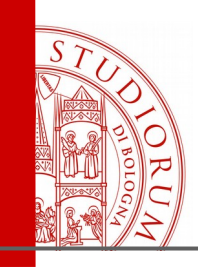
Rust

- system programming language (come C/C++, D, Go, ..)
- no (= minimal) runtime
  - usa system threads
  - no garbage collection
- zero-cost abstractions
  - polimorfismo parametrico via monomorfizzazione
  - il complesso sistema di ownership dei dati in memoria è interamente imposto a compile-time
- guaranteed memory safety
  - no memory leaks, no double deallocation, no dandling pointes, no data races
- fearless concurrency
  - il type system (+ smart pointers) minimizzano i problemi legati alla concorrenza



Rust: features non caratteristiche

- chiusure
- Algebraic Data Types + pattern matching
- no NULL values (usare option type)
- polimorfismo parametrico bounded (templates/generics con bounds via traits)
- traits (le classi inizialmente presenti sono state presto rimosse) + trait objects (per dynamic dispatch) + trait bounds + associated types
- moduli annidabili (alla OCaml, ma niente funtori e niente module types)



Rust: gestione della memoria

- Primo meccanismo:
  complesso (e molto restrittivo) sistema di ownership (tipo RAII – Resource Allocation is Initialization) con borrowing in lettura e/o scrittura
- Secondo meccanismo (anche in C++):
  smart pointers qualora il primo meccanismo diventi troppo complesso da gestire

Secondo meccanismo: gli smart pointers

Smart Pointers

- strutture dati user o system defined
- implementano uno o più traits (p.e. funzione invocata quando una stack variable di tipo smart pointer esce di scope e viene deallocata)
- si comportano come puntatori (p.e. è possibile dereferenziarli per accedere al valore puntato)
- forzano determinate politiche di gestione della memoria (e.g. allocazione nello heap Box<T>, reference counting Rc<T>, accesso atomico in lettura/scrittura, garbage collection, etc.)

Smart pointers: esempi

```
fn main() {
    let x = 5                          // allocata sullo stack
    let b = Box::new(5)                // 5 allocato nello heap
                                       // b è uno smart pointer
                                       // allocato sullo stack
     println!("x = {}, b = {}", x, b) // dereferencing di b autom.
}  // sia x che b escono di scope
   // lo stack frame della chiamata viene deallocato
   // quando b viene deallocato il dato sullo heap a cui b
   // punta viene deallocato anche lui
```

Box<T>: quando usarlo?

- i dati in Rust occupano un numero non uniforme di byte (come in C)
- Box<T> permette di allocare dati di grandi dimensioni sullo heap, facendo riferimento a essi con uno smart pointer di dimensione fissata e piccola
- di fatto necessari per implementare ADT ricorsivi!

ADT ricorsivi

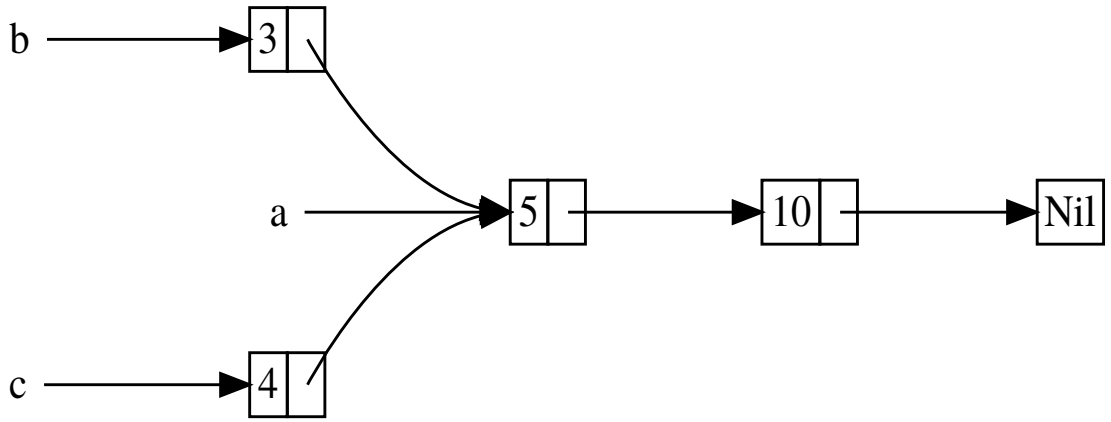
```
// Dichiarazione errata: List può avere una dimensione
// arbitraria e quindi un Cons richierebbe dimensione
// arbitraria
enum List {
    Cons(i32, List),
    Nil,
}

// Dichiarazione corretta, allocando le celle sullo heap
```

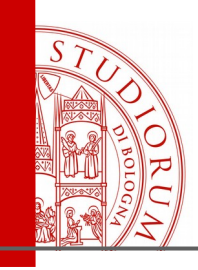
ADT ricorsivi

```
// Dichiarazione corretta
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let list = Cons(1,                  // cella allocata sullo stack
        Box::new(Cons(2,                // cella allocata sullo heap
            Box::new(Cons(3,
                Box::new(Nil))))));
} // quando list va out-of-scope, tutte le celle vengono deallocate
```

Reference counting: Rc<T>

- vogliamo implementare sharing di sotto-liste:



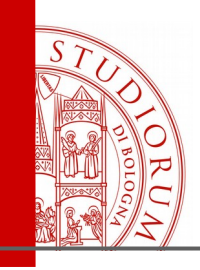


Reference counting: Rc<T>

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

// Codice errato (non compila): cosa succederebbe se b andasse
// out of scope e c no?
fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

Reference counting: Rc<T>

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;

// Codice corretto:
//   - Rc::new alloca spazio nello heap per un dato + il suo contatore
//     di riferimenti
//   - clone incrementa il reference counter
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
} // quando b e c vanno out-of-scope il counter di a viene decrementato
  // di 2 e, raggiunto lo 0, anche a viene deallocato dallo heap
```



Reference counting: Rc<T>

- come vedremo, Rust garantisce che un dato possa avere al più una reference mutable oppure un numero arbitrario di reference in sola lettura (no data races!)
- Box<T> non ha restrizioni da questo punto di vista
- Rc<T> permette solamente reference di sola lettura!

RefCell<T>

- è uno smart-pointer che A RUN-TIME verifica la condizione “al più una mutable reference o n immutable ones”
- solleva un panic! in caso contrario
- da usare con parsimonia: meglio affidarsi al controllo a compile-time di Rust

RefCell<T>

```
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}
...
fn main() {
    let value = Rc::new(RefCell::new(5));
    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10; // per mutare la cella ne acquisisco un
                               // lock in scrittura

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

Weak<T>: gestire i cicli

- la tecnica del reference counting porta a memory leaks in caso di cicli
- spesso in un ciclo vi sono dei puntatori che rappresentano che sono naturalmente forti e altri deboli
- una cella è viva quando ha almeno un puntatore forte entrante; morta altrimenti
- ovvero: i deboli non contano ai fini del RC

Weak<T>: gestire i cicli

- Esempio: alberi. Un nodo ha
  - due puntatori forti (e.g. Rc<Node>) ai figli dx e sx
  - un puntatore debole (Weak<Node>) al padre
- Ragionamento:
  - se un nodo viene sganciato da un'albero (ovvero il padre non punta più a lui) allora diventa unreachable e deve essere eliminato
  - non solo: tutti i suoi discendenti sono unreacheable e vanno reclamati
  - questo anche se il nodo e tutti i discendenti si puntano fra di loro (i figli puntano al padre, ma debolmente)

Weak<T>: gestire i cicli

- upgrade data una weak reference ritorna una Rc<T> reference se l'oggetto è ancora allocato

upgrade(r Weak<T>) → Option<Rc<T>>

- downgrade restituisce un weak pointer per un Rc<T>

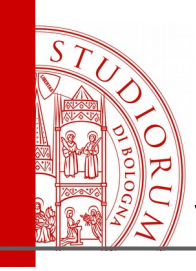
downgrade(r Rc<T>) → Weak<T>

Weak<T>: gestire i cicli

```
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node { value: 3, parent: RefCell::new(Weak::new()),
                      children: RefCell::new(vec![]), });
    assert_eq!(None, leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node { value: 5, parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]), });
    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    assert_eq!(Some &branch, leaf.parent.borrow().upgrade());
}
```

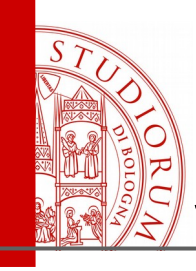
Weak pointers

- Anche nei linguaggi con garbage collector si possono creare situazioni dove la memoria non viene reclamata anche se dovrebbe
- Es: una hash-table viene utilizzata per fare memoization, ovvero associare input a output di una funzione per non ricalcolarli
- Se il puntatore dalla hash-table all'input è strong, poiché la hash-table è sempre reachable, tutti gli input usati in passato non possono essere più reclamati anche se sono unreachable in altri modi
- Soluzioni: weak pointers, weak hash tables, memonoids, etc. (googlate OCaml + questi nomi per avere esempi)



Smart pointers e concorrenza

- Numerosi altri smart pointers usati in scenari concorrenti con memoria condivisa
  - Mutex<T> con metodo bloccante lock() per ottenere reference al contenuto T
  - Arc<T> reference counting atomico da utilizzare in contesti concorrenti



Smart pointers e concorrenza

```
fn main() {
    let counter = Arc::new(Mutex::new(0)); // se non venisse usato Arc ...
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);  // … qua counter non potrebbe essere
                                             // clonato ...
        let handle = thread::spawn(move || {       // e sarebbe riportato un bug di
            let num = counter.lock().unwrap(); // ownership qui
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

User defined smart pointers

- è sufficiente definire una strattura dati che implementa un certo numero di traits (a seconda di quali features delle reference volete)
- talvolta per implementare certe funzionalità è necessario fare ricorso a costrutti unsafe del linguaggio (che manipolano la memoria bypassando i rigidi controlli di Rust)

Esempio

```
struct MyBox<T>(T);   // una tupla con un solo campo

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> { MyBox(x) }
}
……
impl<T> Deref for MyBox<T> {       // implementiamo il dereferencing, *
    type Target = T;               // associated type: * restituisce un &T,
                                   // ovvero una reference al Target che è T

    fn deref(&self) -> &T { &self.0 }   // il T è il valore dell'unico campo della tupla
}

impl<T> Drop for MyBox<T> {
    fn drop(&mut self) {
        println!("Dropping MyBox<T> with data `{}`!", self.0);
    }
}
```

Esempio (continua)

```
fn main() {
    let x = MyBox::new(4);          // crea una smart pointer per 4
    let y = *x;                     // lo dereferenzia
    println!("x = {}, y = {}", x, y);
}                                   // lo smart pointer esce di scope
```

```
claudio@zenone:~/didattica/emerging/rust$ ./smart
x = 4, y = 4
Dropping MyBox<T> with data `4`!
```

Primo meccanismo: ownership + borrowing

Ownership

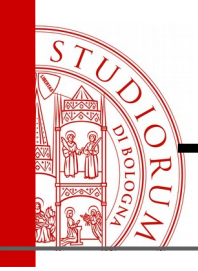
- ogni cella di memoria sullo heap ha un owner, che è responsabile per la sua deallocazione
- quando una cella sullo heap viene creata e un puntatore a esso viene assegnato a una variabile sullo stack, quest'ultima ne diventa l'owner
- quando invece il puntatore viene assegnato a un'altra cella sullo heap, questa ne diventa l'owner
- quando un owner viene deallocato (e.g. il blocco di una variabile sullo stack viene dellocato) le celle RICORSIVAMENTE possedute vengono rilasciate

Trasferimento della ownership

- una cella sullo heap ha sempre uno e un solo owner
- assegnamenti e passaggio come parametri della variabile/cella che ha l'ownership trasferiscono (move) l'ownership
- quando una variabile perde l'ownership, essa non può più essere utilizzata!

Trasferimento della ownership

```
fn main() {
    let x = 4;                          // x è unboxed, quindi sullo stack
    let s = String::from("ciao");       // s sullo stack punta a una stringa nello heap
    let y = x;
    // scommentando la prossima linea e commentando la successiva
    // s perde l'ownership della stringa
    //let t = s;
    let t = String::from("ciao");
    println!("x = {}, y = {}", x, y);
    // la prossima riga è errata se s non ha più l'ownership
    println!("s' = {}, t = {}", s, t);
}
```



Trasferimento della ownership

```
fn main() {
    let s1 = gives_ownership();                 // ownership taken
    let s2 = String::from("hello");             // ownership taken
    let s3 = takes_and_gives_back(s2);      // s2 looses ownership; s3 takes it
}   // the strings pointed to by s1 and s3 are deallocated

fn gives_ownership() -> String {
    let some_string = String::from("hello");    // allocated here
    some_string                                 // ownership transferred
}

fn takes_and_gives_back(a_string: String) -> String {  // ownership taken
    a_string                                           // ownership transferred
}
```

References

- &x è una reference a(l contenuto di) x
- &**mut** x è una reference a(l contenuto di) x che permette di modificarne il contenuto
- se x ha tipo T, &x ha tipo &T e &**mut** x ha tipo &**mut** T
- prendere una reference di una variabile implica fare borrowing della variabile
- no data races (anche concorrentemente):
  - se una variabile è borrowed mutably, nessun altro borrow è possibile e l'owner è frozen (= non può accedere alla variabile fino a quando il borrowing termina)
  - se l'ownership è mutable e la variabile viene borrowed, l'owner è frozen (= non può modificare la variabile fino a quando il borrowing termina)

Borrowing

```
fn main() {
    let x = 4;
    let y = &x;
    let t = String::from("ciao");          // takes immutable ownership
    let s = &t;                            // borrows immutably
    println!("x = {}, y = {}", x, y);
    println!("s = {}, t = {}", s, t);
} // end of borrowing (s goes out of scope) and end of ownership (t goes out of
  // scope and the string is deallocated
```

Borrowing

```
fn main() {
    let mut x = 4;
    let y = &x;
    x = 5;                    // error: assignment to borrowed `x`
}

fn main() {
    let mut x = 4;
    { let y = &x; }           // ok: the borrow ends at the end of inner block!
    x = 5;
}

fn main() {
    let x = 4;
    let z = &mut x;           // error: cannot borrow immutable local variable as mutable
}
```

Borrowing

```
fn main() {
    let mut x = 4;
    let y = &x;
    let z = &mut x;      // error: cannot borrow as mutable because it is also borrowed
                         // as immutable
}

fn main() {
    let mut x = 4;
    let y = &mut x;
    let z = &mut x;      // error: cannot borrow as mutable more than once at a time
}
```

Borrowing

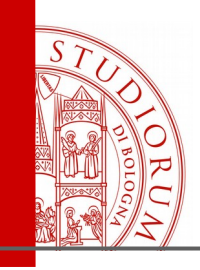
```
fn increment(x: &mut i32) {    // syntactic sugar at work! See later
    *x = *x + 1;
}

fn main() {
    let mut x = 4;              // x has mutable ownership
    increment(&mut x);          // mutable borrows begins and ends
    x = x + 1;                  // so x can be used again here
    println!("x = {}", x);
}
```

Lifetimes

- le celle di memoria hanno un lifetime che indica quando la cella verrà deallocata dall'owner
- lifetime <> scope, p.e. nel caso in cui l'ownership venga trasferita
- ogni reference ha di fatto due lifetime: quello della reference e quello di ciò a cui la reference punta
- NO DANDLING POINTERS: Rust verifica che il primo lifetime sia sempre inferiore al secondo (sintassi concreta: 'a : 'b per 'a,'b variabili di lifetime con significato "'a termina dopo 'b")

Lifetimes

- l'unico termine ground di tipo lifetime è 'static (= vivo fino al termine del programma, p.e. variabile globale)
- variabili di lifetime indicate con 'a, 'b, …
- template astratti su variabili di lifetime + bound (polimorfismo bounded)
- reference tipate con il lifetime
  Es.: &'a i32 reference a un i32 di lifetime 'a
- elisione: ove non necessari i lifetime si possono non esplicitare



Lifetimes

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle<'a>() -> &'a String {
    let s = String::from("hello");
    &s
}  // error: the lifetime of s ends here and it should end at 'a
```

Lifetimes

```
// i lifetime 'b e 'c devono terminare dopo il lifetime di 'a
fn max<'a,'b : 'a,'c : 'a>(x: &'b i32, y: &'c i32) -> &'a i32 {
    std::cmp::max(x,y)
}

fn main() {
    //let z;                  // error se z è dichiarato prima di x o y
    let x = 4;
    let y = 3;
    let z;                    // ok se z è dichiarato dopo x,y
    z = max(&x, &y);
    println!("max = {}", z);
} // i lifetime finiscono in ordine inverso di dichiarazione
```

Lifetimes

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentence };
}
```

Lifetimes

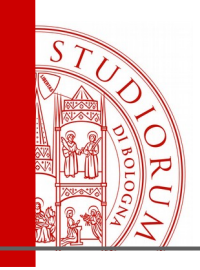
```
struct Ref<'a, T: 'a>(&'a T);  // tupla con un campo che contiene un valore di
                               // tipo T tale che tutte le reference in T vivono
                               // almeno quanto ‘a

fn print_ref<'a, T>(t: &'a T) where
    T: Debug + 'a {
    println!("`print_ref`: t is {:?}", t);
}
```

Slices

- una slice è uno smart pointer per fare borrowing mutabile o meno di una parte di una struttura
- Es: str (string slices), &[T] (vector slices)
- Es: slice = puntatore + numero di byte + capacità residua
- in quanto smart pointer le slices hanno lifetimes

Slices

```
// This function borrows a slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Fixed-size array (type signature is superfluous)
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // Slice containing last 3 elements
    analyze_slice(&xs[2 .. 4])
}
```

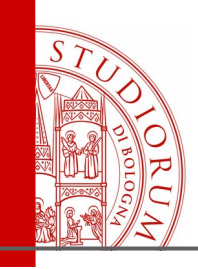


Chiusure

- Rust ha le chiusure che “catturano” le variabili libere nella chiusura
- cattura =
  - Trasferimento di ownership se la chiusura è di tipo move: (**move** |params| { body })
  - Borrowing mutabile/immutabile altrimenti
- vi è un trait per le funzioni, uno per le chiusure move, uno per le chiusure con borrowing mutabile, uno per quelle immutabili, ...

Osservazioni

- l’analisi statica di Rust aiuta a prevenire moltissimi errori comuni nella gestione esplicita della memoria e nella concorrenza con memoria condivisa
- i messaggi di errore sono spesso molto complessi da decifrare e la soluzione è spesso non ovvia (vedi Q&A sulle mailing list)
- bisogna cercare la (ri)formulazione del codice/algoritmo accettata da Rust
- necessaria ottima conoscenza degli smart pointers di libreria



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Claudio Sacerdoti Coen

Dipartimento di Informatica: Scienza e Ingegneria (DISI)

claudio.sacerdoticoen@unibo.it

www.unibo.it