MyIterator<T> in Rust

Edmund Cape
with contributions from the Rust user-forum

v6.1 Last updated Feb 16, 2021

MyCollection<T> and MyIterator<T> ...

A collection can be stored values, or a function that generates values.

Iterator hosts a dynamic view of the elements in the collection.

MyCollection<T>

```
struct MyCollection<_, T> {
  data: _ Data<T>,
}
impl<_, T> MyCollection<_, T> {
  /* allocate, add, remove */
}
```

MyIterator<T>

```
struct MyIterator<_, T> {
  cursor: Position,
  link_to_data: _ MyCollection<T>
}
impl<_, T> MyIterator<_, T> {
  fn next(&mut self) -> Option<_, T> {
    // view the data to which the cursor "points"
    // advance the cursor
  }
}
```

... are related, but mutually exclusive

→ Storing data is conceptually distinct from viewing the data

→ The relationship is not strictly 1:1

→ The required mutability of the iterator is unrelated to the mutability of the data

MyIterator<T> features are encoded using mutable state, a method and a [reference + type] to the data

Iterator is a mutable view of the elements in the collection.

MyIterator<_, T>

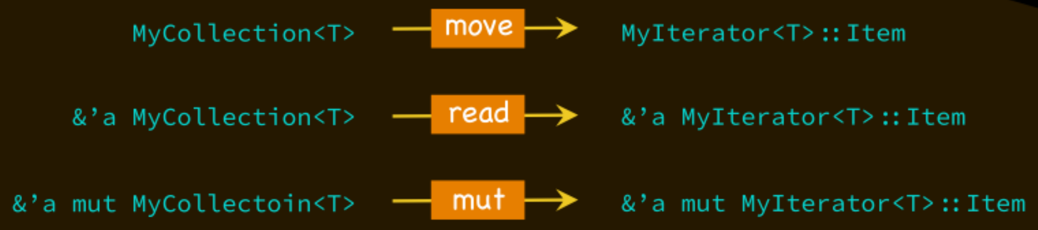
```
struct MyIterator<_, T> {
  cursor: Position,
  link_to_data: _ MyCollection<T>
}
impl<_, T> MyIterator<_, T> {
  fn next(&mut self) → Option<_, T> {
    // view the data to which the cursor "points"
    // advance the cursor
  }
}
```

Encoding of iterator features

→ Next position: current position + some increment

→ View of position: reference to the data + position

→ View privileges: data reference type



MyCollection<T> → move → MyIterator<T>::Item

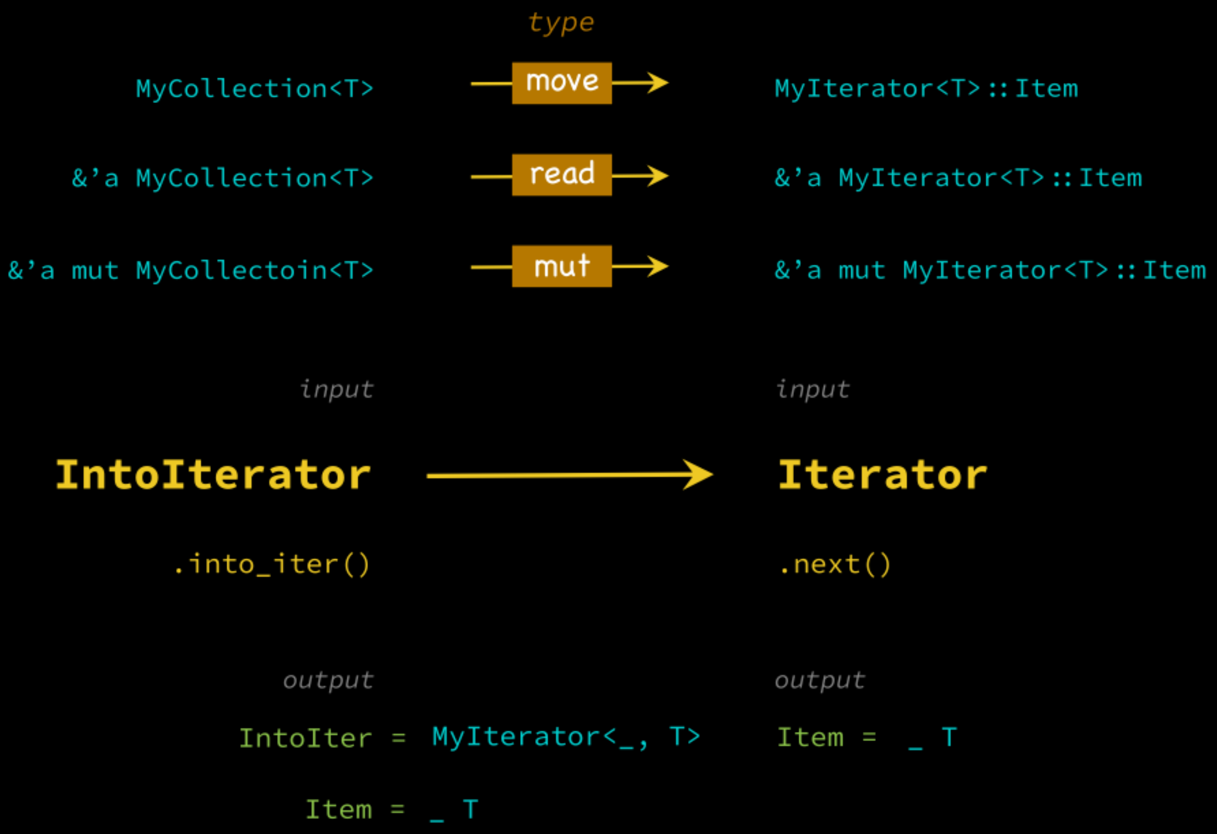
&'a MyCollection<T> → read → &'a MyIterator<T>::Item

&'a mut MyCollectoin<T> → mut → &'a mut MyIterator<T>::Item

There is a 1:1 correspondence between the "view privileges" of the collection and that of the iterator

For any given type of _ MyCollection<T> there is a single _ MyIterator<T>

IntoIterator provides the interface to align the reference type
owned by the collection with that of the iterator



type

MyCollection<T> → move → MyIterator<T>::Item

&'a MyCollection<T> → read → &'a MyIterator<T>::Item

&'a mut MyCollectoin<T> → mut → &'a mut MyIterator<T>::Item

input

IntoIterator → **Iterator**

input

.into_iter()

.next()

output

IntoIter = MyIterator<_, T>

Item = _ T

output

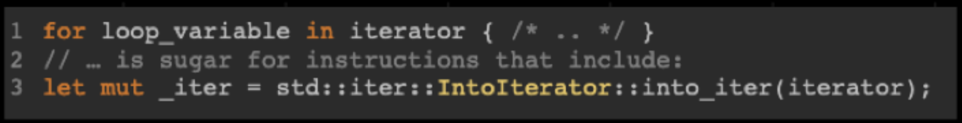
Item = _ T

"The IntoIterator trait serves the purpose of providing specialized Iterator-implementing types, iterator types which have the opportunity to steal the contents of, or point back to, the original collection being iterated over, while also providing any private state necessary for performing the iteration. This decouples iteration from storage."

source: @H2CO3 Rust user-forum

v6.2 Last updated Feb 18, 2021

Implement IntoIterator to specify a “default" iterator for MyCollection<T>



```rust
for loop_variable in iterator { /* .. */ }
// … is sugar for instructions that include:
let mut _iter = std::iter::IntoIterator::into_iter(iterator);
```

std::iter::IntoIterator::into_iter(iterator);

GO

for item in
my_collection

MyCollection<T>

as self

1 as IntoIterator.into_iter()

2 → IntoIter

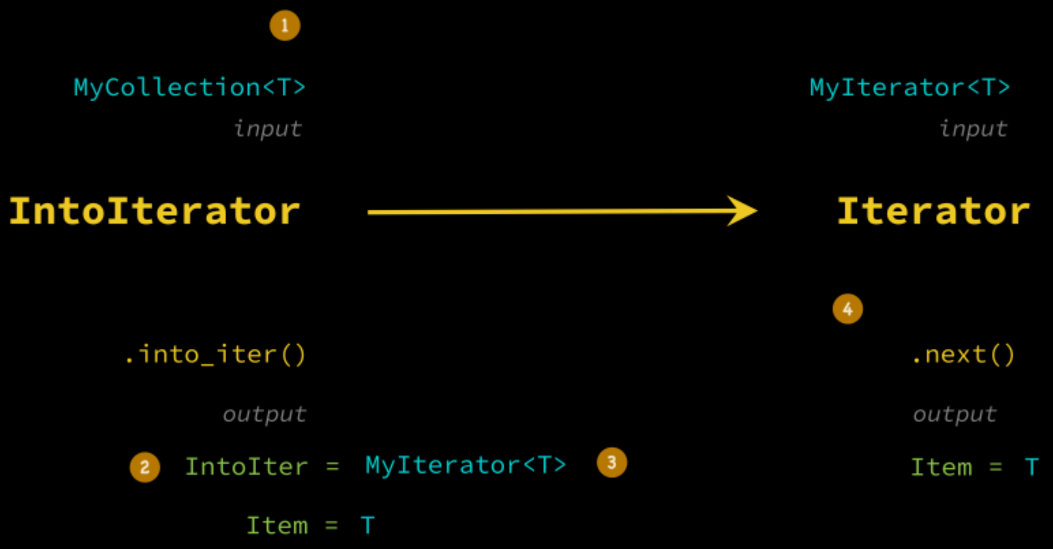
3 as MyIterator<T>

4 as Iterator

.next()

implement

IntoIterator, Iterator
and MyIterator



1

MyCollection<T>

input

IntoIterator

.into_iter()

output

2 IntoIter = MyIterator<T> 3

Item = T

MyIterator<T>

input

Iterator

4

.next()

output

Item = T

consumers of the “default” into_iter()
MyCollection<T> → Iterator

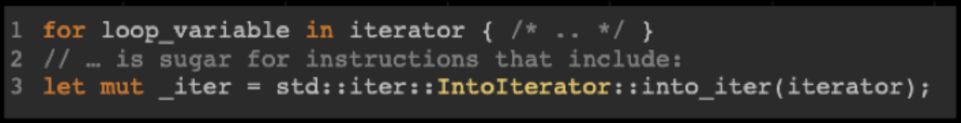
for _ in MyCollection() sugar

Iterator::chain → Chain
Iterator::cmp_by → Ordering
Iterator::zip → Zip
etc…

std::iter::Iterator

v6.2 Last updated Feb 18, 2021

Implement a method MyCollection<T> -> MyIterator<T> and an instance of Iterator



```
for loop_variable in iterator { /* .. */ }
// … is sugar for instructions that include:
let mut _iter = std::iter::IntoIterator::into_iter(iterator);
```

std::iter::IntoIterator::into_iter(iterator);

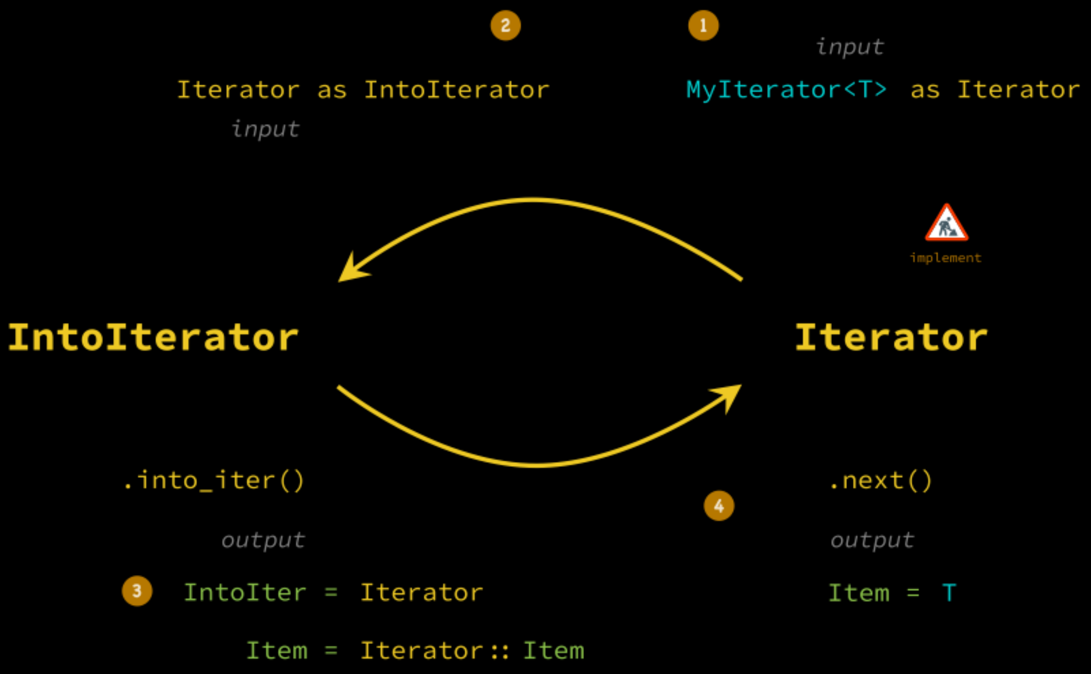
GO

for item in my_collection

```
    MyCollection.iter()
    → MyIterator<T>
    as self
1   as Iterator
2   as IntoIterator.into_iter()
3   → IntoIter
4   as Iterator

          .next()
```

implement

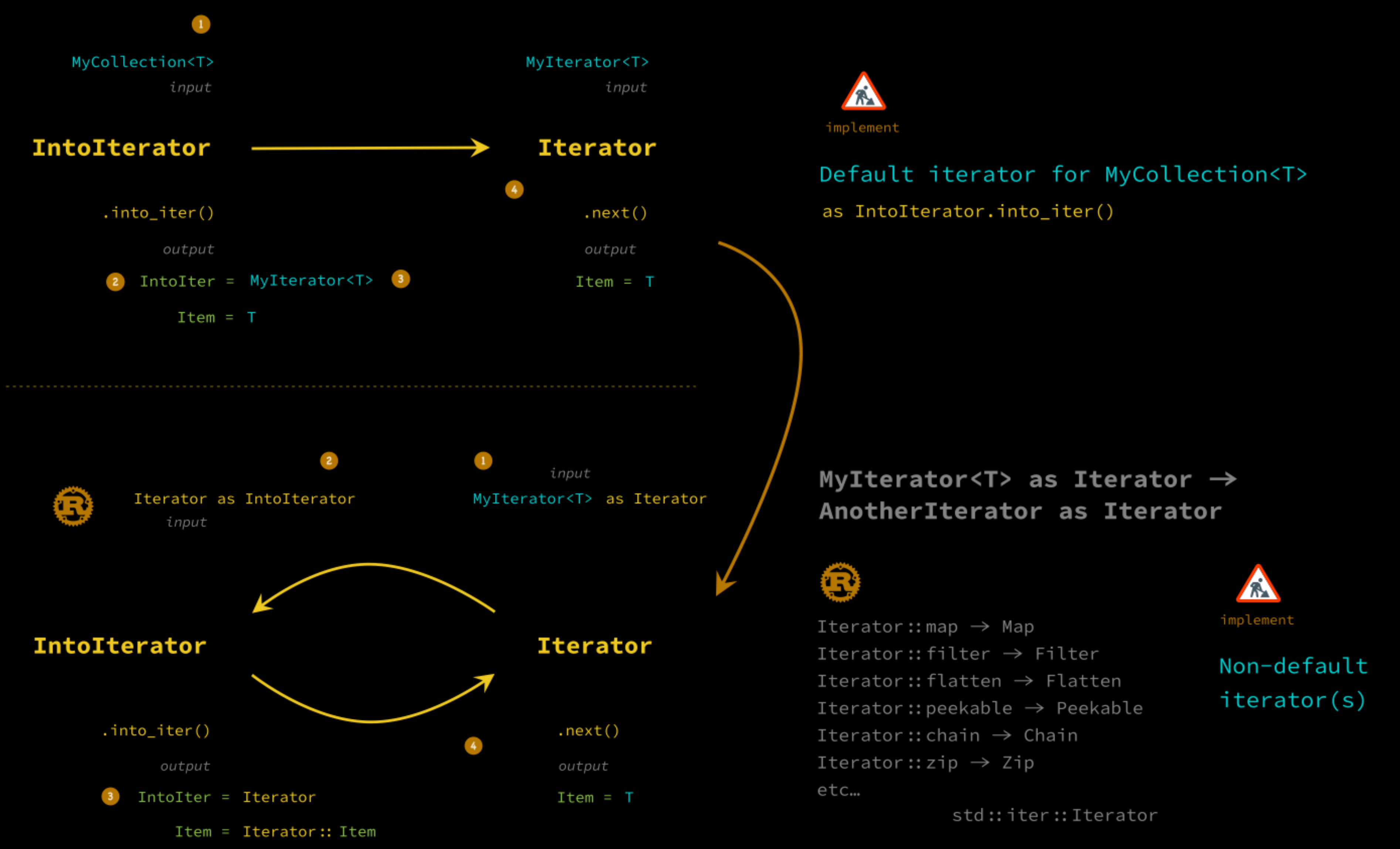
iter, Iterator
and MyIterator



```
impl<I: Iterator> IntoIterator for I {
    type Item = I::Item;
    type IntoIter = I;
    fn into_iter(self) → I {
        self
    }
}
```

v6.2 Last updated Feb 18, 2021

The interaction of the combined approaches



1

MyCollection<T>

input

IntoIterator

.into_iter()

output

2 IntoIter = MyIterator<T> 3

Item = T

MyIterator<T>

input

Iterator

4

.next()

output

Item = T

implement

Default iterator for MyCollection<T>

as IntoIterator.into_iter()

2

Iterator as IntoIterator

input

IntoIterator

.into_iter()

output

3 IntoIter = Iterator

Item = Iterator::Item

1

input

MyIterator<T> as Iterator

Iterator

4

.next()

output

Item = T

MyIterator<T> as Iterator →
AnotherIterator as Iterator

Iterator::map → Map
Iterator::filter → Filter
Iterator::flatten → Flatten
Iterator::peekable → Peekable
Iterator::chain → Chain
Iterator::zip → Zip
etc…

std::iter::Iterator

implement

Non-default iterator(s)

v6.2 Last updated Feb 18, 2021

IntoIterator is implemented for all versions of Vec<T>; array does not have a capacity to allow values to be "moved out".

`_ Vec<T> → (fn next(&mut self) → Option<T>)`

Item output	**T**	**&'_ T**	**&'_ mut T**
Move		... move a borrow	
Method	into_iter()	into_iter()	into_iter() ← trait method
Caller	Vec<T>	↓ &Vec<T>	↓ &mut Vec<T>
Receiver	Vec<T>	&[T]	&mut &[T]
Borrow			
Method	NA	iter()	iter_mut() ← inherent method
Caller		↓ &Vec<T>	↓ &mut Vec<T>
Receiver		&[T]	&mut [T]

`_ [T; N] → (fn next(&mut self) → Option<T>)`

Item output	**T**	**&'_ T**	**&'_ mut T**
Move		... move a borrow	
Method	into_iter()	into_iter()	into_iter() ⚠
Caller	↓ [T; N]	↓ &[T; N]	↓ &mut [T; N]
Receiver	&[T] ✗	&[T]	&mut &[T]
Borrow			
Method	NA	iter()	iter_mut()
Caller		↓ &[T; N]	↓ &mut [T; N]
Receiver		&[T]	&mut [T]

Other observations

→ the naming convention of the inherent methods

→ the "into_iter" namesake implies a move; the intent can get ambiguous when "moving a borrow"; for arrays, the compiler complains and recommends using the inherent methods instead

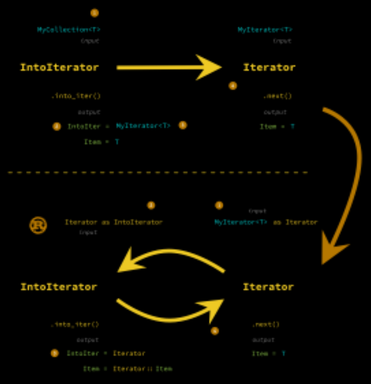
→ Both the Vec and array iterators with a borrow semantic, converge onto the slice implementations of the iterators `&[T] or &mut [T]`

✗ There is no way to move values out of an array using iterators. However, the compiler permits iterating over values of an array by "graciously" casting the array to a borrow. This policy may change, in which case code that dereferences these items will "break".

⚠ Calling into_iter() on an array generates a warning because it is considered "ambiguous".

v6.2 Last updated Feb 18, 2021

Vec uses a combination of the two approaches; the full range of iterators can be instantiated from a Vec<T> with method names that describe the return type



Each of the Vec<T> types has a "default" iterator returned by the into_iter() trait method; only Vec<T> has a unique iterator that manages the move operation; the borrow-related Vec types point to "custom", inherent methods belonging to the slice type.

`_ Vec<T> → VecIterator<_, T>`

Vec<T>	as IntoIterator.into_iter()		→ IntoIter<T>
&'a Vec<T>	as IntoIterator.into_iter()	~ &'a [T].iter()	→ Iter<'_, T>
&'a mut Vec<T>	as IntoIterator.into_iter()	~ &'a mut [T].iter_mut()	→ IterMut<'_, T>

Note: The dot-operator calls deref on the borrow-related Vec types in order to find an implementation for the iter() and inter_mut() inherent methods.

```
&'a Vec<T>.iter()   … deref to                  &'a [T].iter()         → Iter<'_, T>
&'a mut Vec<T>.iter_mut()  … deref to            &'a mut [T].iter_mut() → IterMut<'_, T>
```

Also note: Covariance rules permit other expressions; however, they ultimately point to one of the above calls.
For instance: `Vec<T>.iter() ⇒ &'a Vec<T>.iter()`

Flexible, robust API

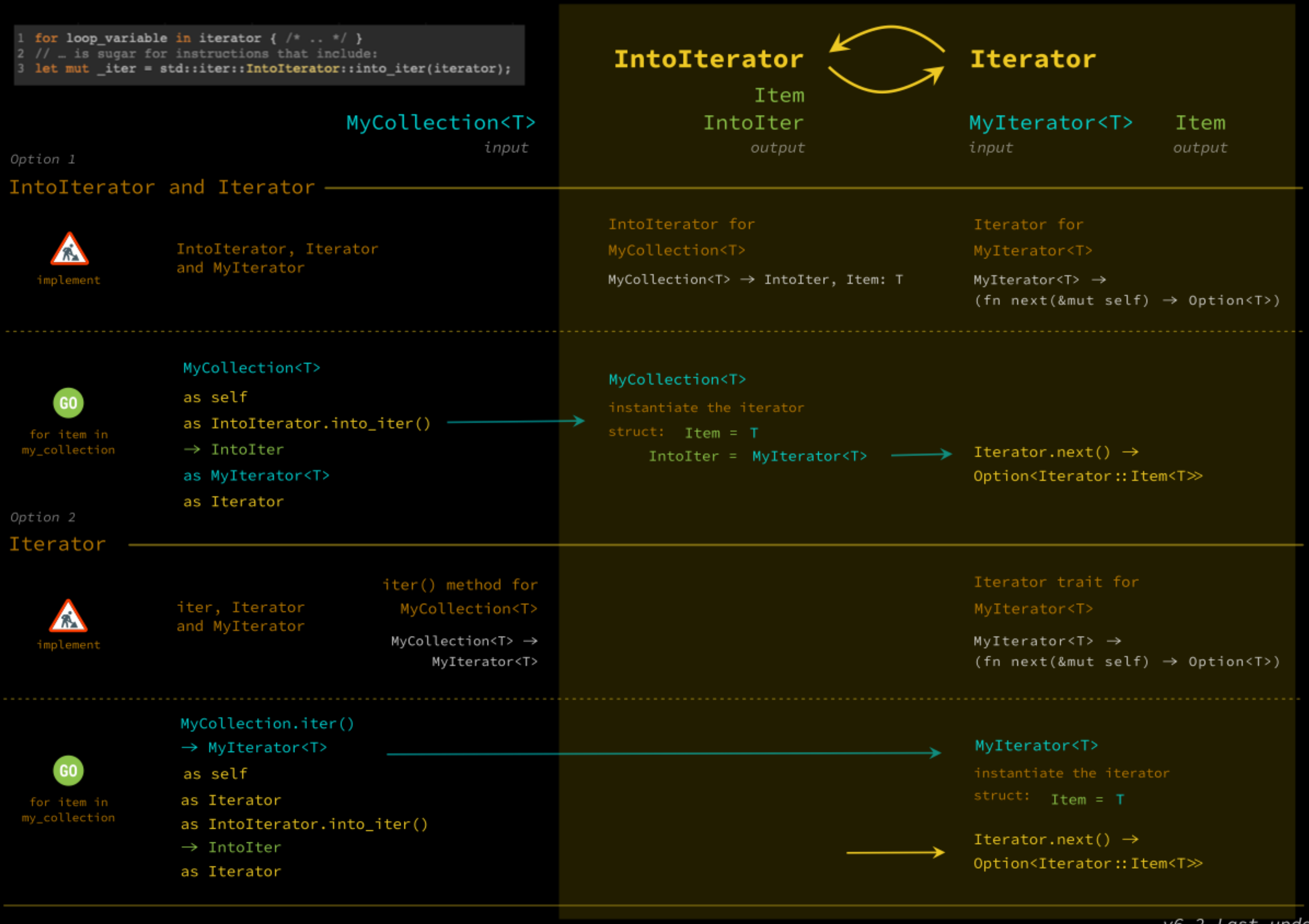
```
Vec<T>.iter(): Iter<'_, T>
Vec<T>.iter_mut(): IterMut<'_, T>
Vec<T>.into_iter(): IntoIter<T>

&'a mut Vec<T>.iter(): Iter<'_, T>
&'a mut Ver<T>.iter_mut(): IterMut<'_, T>

&'a Vec<T>.iter(): Iter<'_, T>
```

v6.2 Last updated Feb 18, 2021

There are two ways to integrate MyCollection<T> into the Rust `for loop` infrastructure

In the core::iter::IntoIterator (v1.50), 11 types implement IntoIterator: Array(borrows), Slice(borrows), Option(move and borrows), Result(move and borrows) and the Iterator trait. Many, many more than 11 types implement Iterator.

All of the data structures in std::collections each implement 3 versions of the IntoIteratotar trait except HashSet and BinaryHeap that do not implement the trait for the &mut type.



```
for loop_variable in iterator { /* .. */ }
// … is sugar for instructions that include:
let mut _iter = std::iter::IntoIterator::into_iter(iterator);
```

IntoIterator

Iterator

MyCollection<T>
input

Item
IntoIter
output

MyIterator<T>
input

Item
output

Option 1

IntoIterator and Iterator

implement

IntoIterator, Iterator
and MyIterator

IntoIterator for
MyCollection<T>
MyCollection<T> → IntoIter, Item: T

Iterator for
MyIterator<T>
MyIterator<T> →
(fn next(&mut self) → Option<T>)

for item in
my_collection

MyCollection<T>
as self
as IntoIterator.into_iter()
→ IntoIter
as MyIterator<T>
as Iterator

MyCollection<T>
instantiate the iterator
struct: Item = T
IntoIter = MyIterator<T>

Iterator.next() →
Option<Iterator::Item<T>>

Option 2

Iterator

implement

iter, Iterator
and MyIterator

iter() method for
MyCollection<T>
MyCollection<T> →
MyIterator<T>

Iterator trait for
MyIterator<T>
MyIterator<T> →
(fn next(&mut self) → Option<T>)

for item in
my_collection

MyCollection.iter()
→ MyIterator<T>
as self
as Iterator
as IntoIterator.into_iter()
→ IntoIter
as Iterator

MyIterator<T>
instantiate the iterator
struct: Item = T

Iterator.next() →
Option<Iterator::Item<T>>

v6.2 Last updated Feb 18, 2021