# Containers and iterators

1 Containers
2 Iterators

# 1 Containers

The container traits are defined in the `std::container` module.

## 1.1 Unique and managed vectors

Vectors have `O(1)` indexing and removal from the end, along with `O(1)` amortized insertion. Vectors are the most common container in Rust, and are flexible enough to fit many use cases.

Vectors can also be sorted and used as efficient lookup tables with the `std::vec::bsearch` function, if all the elements are inserted at one time and deletions are unnecessary.

## 1.2 Maps and sets

Maps are collections of unique keys with corresponding values, and sets are just unique keys without a corresponding value. The `Map` and `Set` traits in `std::container` define the basic interface.

The standard library provides three owned map/set types:

- `std::hashmap::HashMap` and `std::hashmap::HashSet`, requiring the keys to implement `Eq` and `Hash`
- `std::trie::TrieMap` and `std::trie::TrieSet`, requiring the keys to be `uint`
- `extra::treemap::TreeMap` and `extra::treemap::TreeSet`, requiring the keys to implement `TotalOrd`

These maps do not use managed pointers so they can be sent between tasks as long as the key and value types are sendable. Neither the key or value type has to be copyable.

The `TrieMap` and `TreeMap` maps are ordered, while `HashMap` uses an arbitrary order.

Each `HashMap` instance has a random 128-bit key to use with a keyed hash, making the order of a set of keys in a given hash table randomized. Rust provides a SipHash implementation for any type implementing the `IterBytes` trait.

## 1.3 Double-ended queues

The `extra::deque` module implements a double-ended queue with `O(1)` amortized inserts and removals from both ends of the container. It also has `O(1)` indexing like a vector. The contained

elements are not required to be copyable, and the queue will be sendable if the contained type is sendable.

### 1.4 Priority queues

The `extra::priority_queue` module implements a queue ordered by a key. The contained elements are not required to be copyable, and the queue will be sendable if the contained type is sendable.

Insertions have `O(log n)` time complexity and checking or popping the largest element is `O(1)`. Converting a vector to a priority queue can be done in-place, and has `O(n)` complexity. A priority queue can also be converted to a sorted vector in-place, allowing it to be used for an `O(n log n)` in-place heapsort.

## 2 Iterators

### 2.1 Iteration protocol

The iteration protocol is defined by the `Iterator` trait in the `std::iterator` module. The minimal implementation of the trait is a `next` method, yielding the next element from an iterator object:

```
/// An infinite stream of zeroes
struct ZeroStream;

impl Iterator<int> for ZeroStream {
    fn next(&mut self) -> Option<int> {
        Some(0)
    }
}
```

Reaching the end of the iterator is signalled by returning `None` instead of `Some(item)`:

```
/// A stream of N zeroes
struct ZeroStream {
    priv remaining: uint
}

impl ZeroStream {
    fn new(n: uint) -> ZeroStream {
        ZeroStream { remaining: n }
    }
}

impl Iterator<int> for ZeroStream {
    fn next(&mut self) -> Option<int> {
```

Rust 0.8-pre

fbeeeebf

```
        if self.remaining == 0 {
            None
        } else {
            self.remaining -= 1;
            Some(0)
        }
    }
}
```

## 2.2 Container iterators

Containers implement iteration over the contained elements by returning an iterator object. For example, vector slices several iterators available:

- `iter()` and `rev_iter()`, for immutable references to the elements
- `mut_iter()` and `mut_rev_iter()`, for mutable references to the elements
- `consume_iter()` and `consume_rev_iter`, to move the elements out by-value

A typical mutable container will implement at least `iter()`, `mut_iter()` and `consume_iter()` along with the reverse variants if it maintains an order.

### 2.2.1 Freezing

Unlike most other languages with external iterators, Rust has no *iterator invalidation*. As long an iterator is still in scope, the compiler will prevent modification of the container through another handle.

```
let mut xs = [1, 2, 3];
{
    let _it = xs.iter();

    // the vector is frozen for this scope, the compiler will statically
    // prevent modification
}
// the vector becomes unfrozen again at the end of the scope
```

These semantics are due to most container iterators being implemented with `&` and `&mut`.

## 2.3 Iterator adaptors

The `IteratorUtil` trait implements common algorithms as methods extending every `Iterator` implementation. For example, the `fold` method will accumulate the items yielded by an `Iterator` into a single value:

```
let xs = [1, 9, 2, 3, 14, 12];
let result = xs.iter().fold(0, |accumulator, item| accumulator - *item);
assert_eq!(result, -41);
```

Some adaptors return an adaptor object implementing the `Iterator` trait itself:

```
let xs = [1, 9, 2, 3, 14, 12];
let ys = [5, 2, 1, 8];
let sum = xs.iter().chain_(ys.iter()).fold(0, |a, b| a + *b);
assert_eq!(sum, 57);
```

Note that some adaptors like the `chain_` method above use a trailing underscore to work around an issue with method resolve. The underscores will be dropped when they become unnecessary.

## 2.4 For loops

The `for` keyword can be used as sugar for iterating through any iterator:

```
let xs = [2, 3, 5, 7, 11, 13, 17];

// print out all the elements in the vector
for x in xs.iter() {
    println(x.to_str())
}

// print out all but the first 3 elements in the vector
for x in xs.iter().skip(3) {
    println(x.to_str())
}
```

For loops are *often* used with a temporary iterator object, as above. They can also advance the state of an iterator in a mutable location:

```
let xs = [1, 2, 3, 4, 5];
let ys = ["foo", "bar", "baz", "foobar"];

// create an iterator yielding tuples of elements from both vectors
let mut it = xs.iter().zip(ys.iter());

// print out the pairs of elements up to (&3, &"baz")
for (x, y) in it {
    printfln!("%d %s", *x, *y);

    if *x == 3 {
        break;
    }
}

// yield and print the last pair from the iterator
printfln!("last: %?", it.next());
```

Rust 0.8-pre

fbeeeebf

```
// the iterator is now fully consumed
assert!(it.next().is_none());
```

## 2.5 Conversion

Iterators offer generic conversion to containers with the `collect` adaptor:

```
let xs = [0, 1, 1, 2, 3, 5, 8];
let ys = xs.rev_iter().skip(1).transform(|&x| x * 2).collect::<~[int]>();
assert_eq!(ys, ~[10, 6, 4, 2, 2, 0]);
```

The method requires a type hint for the container type, if the surrounding code does not provide sufficient information.

Containers can provide conversion from iterators through `collect` by implementing the `FromIterator` trait. For example, the implementation for vectors is as follows:

```
impl<A, T: Iterator<A>> FromIterator<A, T> for ~[A] {
    pub fn from_iterator(iterator: &mut T) -> ~[A] {
        let (lower, _) = iterator.size_hint();
        let mut xs = with_capacity(lower);
        for x in iterator {
            xs.push(x);
        }
        xs
    }
}
```

### 2.5.1 Size hints

The `Iterator` trait provides a `size_hint` default method, returning a lower bound and optionally on upper bound on the length of the iterator:

```
fn size_hint(&self) -> (uint, Option<uint>) { (0, None) }
```

The vector implementation of `FromIterator` from above uses the lower bound to pre-allocate enough space to hold the minimum number of elements the iterator will yield.

The default implementation is always correct, but it should be overridden if the iterator can provide better information.

The `ZeroStream` from earlier can provide an exact lower and upper bound:

```
/// A stream of N zeroes
struct ZeroStream {
    priv remaining: uint
}
```

```
impl ZeroStream {
    fn new(n: uint) -> ZeroStream {
        ZeroStream { remaining: n }
    }

    fn size_hint(&self) -> (uint, Option<uint>) {
        (self.remaining, Some(self.remaining))
    }
}

impl Iterator<int> for ZeroStream {
    fn next(&mut self) -> Option<int> {
        if self.remaining == 0 {
            None
        } else {
            self.remaining -= 1;
            Some(0)
        }
    }
}
```

## 2.6 Double-ended iterators

The `DoubleEndedIterator` trait represents an iterator able to yield elements from either end of a range. It inherits from the `Iterator` trait and extends it with the `next_back` function.

A `DoubleEndedIterator` can be flipped with the `invert` adaptor, returning another `DoubleEndedIterator` with `next` and `next_back` exchanged.

```
let xs = [1, 2, 3, 4, 5, 6];
let mut it = xs.iter();
printfln!("%?", it.next()); // prints `Some(&1)`
printfln!("%?", it.next()); // prints `Some(&2)`
printfln!("%?", it.next_back()); // prints `Some(&6)`

// prints `5`, `4` and `3`
for &x in it.invert() {
    printfln!("%?", x)
}
```

The `rev_iter` and `mut_rev_iter` methods on vectors just return an inverted version of the standard immutable and mutable vector iterators.

The `chain_`, `transform`, `filter`, `filter_map` and `peek` adaptors are `DoubleEndedIterator` implementations if the underlying iterators are.

```
let xs = [1, 2, 3, 4];
```

Rust 0.8-pre
fbeeeebf

```
let ys = [5, 6, 7, 8];
let mut it = xs.iter().chain_(ys.iter()).transform(|&x| x * 2);

printfln!("%?", it.next()); // prints `Some(2)`

// prints `16`, `14`, `12`, `10`, `8`, `6`, `4`
for x in it.invert() {
    printfln!("%?", x);
}
```

## 2.7 Random-access iterators

The `RandomAccessIterator` trait represents an iterator offering random access to the whole range. The `indexable` method retrieves the number of elements accessible with the `idx` method.

The `chain_` adaptor is an implementation of `RandomAccessIterator` if the underlying iterators are.

```
let xs = [1, 2, 3, 4, 5];
let ys = ~[7, 9, 11];
let mut it = xs.iter().chain_(ys.iter());
printfln!("%?", it.idx(0)); // prints `Some(&1)`
printfln!("%?", it.idx(5)); // prints `Some(&7)`
printfln!("%?", it.idx(7)); // prints `Some(&11)`
printfln!("%?", it.idx(8)); // prints `None`

// yield two elements from the beginning, and one from the end
it.next();
it.next();
it.next_back();

printfln!("%?", it.idx(0)); // prints `Some(&3)`
printfln!("%?", it.idx(4)); // prints `Some(&9)`
printfln!("%?", it.idx(6)); // prints `None`
```