Module option

Module std::option

Optional values.

Type `Option` represents an optional value: every `Option` is either `Some` and contains a value, or `None`, and does not. `Option` types are very common in Rust code, as they have a number of uses:

- Initial values
- Return values for functions that are not defined over their entire input range (partial functions)
- Return value for otherwise reporting simple errors, where `None` is returned on error
- Optional struct fields
- Struct fields that can be loaned or "taken"
- Optional function arguments
- Nullable pointers
- Swapping things out of difficult situations

`Option`s are commonly paired with pattern matching to query the presence of a value and take action, always accounting for the `None` case.

Run

```
fn divide(numerator: f64, denominator: f64) -> Option<f64> {
    if denominator == 0.0 {
        None
    } else {
        Some(numerator / denominator)
    }
}

// The return value of the function is an option
let result = divide(2.0, 3.0);

// Pattern match to retrieve the value
match result {
    // The division was valid
    Some(x) => println!("Result: {}", x),
    // The division was invalid
    None    => println!("Cannot divide by 0"),
}
```

Options and pointers ("nullable" pointers)

Rust's pointer types must always point to a valid location; there are no "null" references. Instead, Rust has *optional* pointers, like the optional owned box, `Option<Box<T>>`.

The following example uses `Option` to create an optional box of `i32`. Notice that in order to use the inner `i32` value, the `check_optional` function first needs to use pattern matching to determine whether the box has a value (i.e., it is `Some(...)`) or not (`None`).

Run

```
let optional = None;
check_optional(optional);

let optional = Some(Box::new(9000));
check_optional(optional);

fn check_optional(optional: Option<Box<i32>>) {
    match optional {
        Some(p) => println!("has value {}", p),
        None => println!("has no value"),
    }
}
```

Representation

Rust guarantees to optimize the following types `T` such that `Option<T>` has the same size as `T`:

- `Box<U>`
- `&U`
- `&mut U`
- `fn`, `extern "C" fn`
- `num::NonZero*`
- `ptr::NonNull<U>`
- `#[repr(transparent)]` struct around one of the types in this list.

This is called the "null pointer optimization" or NPO.

It is further guaranteed that, for the cases above, one can `mem::transmute` from all valid values of `T` to `Option<T>` and from `Some::<T>(_)` to `T` (but transmuting `None::<T>` to `T` is undefined behaviour).

Method overview

In addition to working with pattern matching, `Option` provides a wide variety of different methods.

Querying the variant

The `is_some` and `is_none` methods return `true` if the `Option` is `Some` or `None`, respectively.

Adapters for working with references

- `as_ref` converts from `&Option<T>` to `Option<&T>`
- `as_mut` converts from `&mut Option<T>` to `Option<&mut T>`
- `as_deref` converts from `&Option<T>` to `Option<&T::Target>`
- `as_deref_mut` converts from `&mut Option<T>` to `Option<&mut T::Target>`
- `as_pin_ref` converts from `Pin<&Option<T>>` to `Option<Pin<&T>>`
- `as_pin_mut` converts from `Pin<&mut Option<T>>` to `Option<Pin<&mut T>>`

Extracting the contained value

These methods extract the contained value in an `Option<T>` when it is the `Some` variant. If the `Option` is `None`:

- `expect` panics with a provided custom message
- `unwrap` panics with a generic message
- `unwrap_or` returns the provided default value
- `unwrap_or_default` returns the default value of the type `T` (which must implement the `Default` trait)
- `unwrap_or_else` returns the result of evaluating the provided function

Transforming contained values

These methods transform `Option` to `Result`:

- `ok_or` transforms `Some(v)` to `Ok(v)`, and `None` to `Err(err)` using the provided default `err` value
- `ok_or_else` transforms `Some(v)` to `Ok(v)`, and `None` to a value of `Err` using the provided function
- `transpose` transposes an `Option` of a `Result` into a `Result` of an `Option`

These methods transform the `Some` variant:

- `filter` calls the provided predicate function on the contained value `t` if the `Option` is `Some(t)`, and returns `Some(t)` if the function returns `true`; otherwise, returns `None`
- `flatten` removes one level of nesting from an `Option<Option<T>>`
- `map` transforms `Option<T>` to `Option<U>` by applying the provided function to the contained value of `Some` and leaving `None` values unchanged

These methods transform `Option<T>` to a value of a possibly different type `U`:

- `map_or` applies the provided function to the contained value of `Some`, or returns the provided default value if the `Option` is `None`
- `map_or_else` applies the provided function to the contained value of `Some`, or returns the result of evaluating the provided fallback function if the `Option` is `None`

These methods combine the `Some` variants of two `Option` values:

- `zip` returns `Some((s, o))` if `self` is `Some(s)` and the provided `Option` value is `Some(o)`; otherwise, returns `None`
- `zip_with` calls the provided function `f` and returns `Some(f(s, o))` if `self` is `Some(s)` and the provided `Option` value is `Some(o)`; otherwise, returns `None`

Boolean operators

These methods treat the `Option` as a boolean value, where `Some` acts like `true` and `None` acts like `false`. There are two categories of these methods: ones that take an `Option` as input, and ones that take a function as input (to be lazily evaluated).

The `and`, `or`, and `xor` methods take another `Option` as input, and produce an `Option` as output. Only the `and` method can produce an `Option<U>` value having a different inner type `U` than `Option<T>`.

method	self	input	output
and	None	(ignored)	None
and	Some(x)	None	None
and	Some(x)	Some(y)	Some(y)
or	None	None	None
or	None	Some(y)	Some(y)
or	Some(x)	(ignored)	Some(x)
xor	None	None	None
xor	None	Some(y)	Some(y)
xor	Some(x)	None	Some(x)
xor	Some(x)	Some(y)	None

The `and_then` and `or_else` methods take a function as input, and only evaluate the function when they need to produce a new value. Only the `and_then` method can produce an `Option<U>` value having a different inner type `U` than `Option<T>`.

method	self	function input	function result	output
and_then	None	(not provided)	(not evaluated)	None
and_then	Some(x)	x	None	None
and_then	Some(x)	x	Some(y)	Some(y)
or_else	None	(not provided)	None	None

method	self	function input	function result	output
`or_else`	`None`	(not provided)	`Some(y)`	`Some(y)`
`or_else`	`Some(x)`	(not provided)	(not evaluated)	`Some(x)`

This is an example of using methods like `and_then` and `or` in a pipeline of method calls. Early stages of the pipeline pass failure values (`None`) through unchanged, and continue processing on success values (`Some`). Toward the end, `or` substitutes an error message if it receives `None`.

Run

```
let mut bt = BTreeMap::new();
bt.insert(20u8, "foo");
bt.insert(42u8, "bar");
let res = vec![0u8, 1, 11, 200, 22]
    .into_iter()
    .map(|x| {
        // `checked_sub()` returns `None` on error
        x.checked_sub(1)
            // same with `checked_mul()`
            .and_then(|x| x.checked_mul(2))
            // `BTreeMap::get` returns `None` on error
            .and_then(|x| bt.get(&x))
            // Substitute an error message if we have `None` so far
            .or(Some(&"error!"))
            .copied()
            // Won't panic because we unconditionally used `Some` ab
            .unwrap()
    })
    .collect::<Vec<_>>();
assert_eq!(res, ["error!", "error!", "foo", "error!", "bar"]);
```

Comparison operators

If `T` implements `PartialOrd` then `Option<T>` will derive its `PartialOrd` implementation. With this order, `None` compares as less than any `Some`, and two `Some` compare the same way as their contained values would in `T`. If `T` also implements `Ord`, then so does `Option<T>`.

Run

```
assert!(None < Some(0));
assert!(Some(0) < Some(1));
```

Iterating over `Option`

An `Option` can be iterated over. This can be helpful if you need an iterator that is conditionally empty. The iterator will either produce a single value (when the `Option` is `Some`), or produce no values (when the `Option` is `None`). For example, `into_iter` acts like `once(v)` if the `Option` is `Some(v)`, and like `empty()` if the `Option` is `None`.

Iterators over `Option<T>` come in three types:

- `into_iter` consumes the `Option` and produces the contained value
- `iter` produces an immutable reference of type `&T` to the contained value
- `iter_mut` produces a mutable reference of type `&mut T` to the contained value

An iterator over `Option` can be useful when chaining iterators, for example, to conditionally insert items. (It's not always necessary to explicitly call an iterator constructor: many `Iterator` methods that accept other iterators will also accept iterable types that implement `IntoIterator`, which includes `Option`.)

Run

```
let yep = Some(42);
let nope = None;
// chain() already calls into_iter(), so we don't have to do so
let nums: Vec<i32> = (0..4).chain(yep).chain(4..8).collect();
assert_eq!(nums, [0, 1, 2, 3, 42, 4, 5, 6, 7]);
let nums: Vec<i32> = (0..4).chain(nope).chain(4..8).collect();
assert_eq!(nums, [0, 1, 2, 3, 4, 5, 6, 7]);
```

One reason to chain iterators in this way is that a function returning `impl Iterator` must have all possible return values be of the same concrete type. Chaining an iterated `Option` can help with that.

Run

```
fn make_iter(do_insert: bool) -> impl Iterator<Item = i32> {
    // Explicit returns to illustrate return types matching
    match do_insert {
        true => return (0..4).chain(Some(42)).chain(4..8),
        false => return (0..4).chain(None).chain(4..8),
    }
}
println!("{:?}", make_iter(true).collect::<Vec<_>>());
println!("{:?}", make_iter(false).collect::<Vec<_>>());
```

If we try to do the same thing, but using `once()` and `empty()`, we can't return `impl Iterator` anymore because the concrete types of the return values differ.

```
// This won't compile because all possible returns from the fu
// must have the same concrete type.
fn make_iter(do_insert: bool) -> impl Iterator<Item = i32> {
    // Explicit returns to illustrate return types not matching
    match do_insert {
        true => return (0..4).chain(once(42)).chain(4..8),
        false => return (0..4).chain(empty()).chain(4..8),
    }
}
```

Run

Collecting into `Option`

`Option` implements the `FromIterator` trait, which allows an iterator over `Option` values to be collected into an `Option` of a collection of each contained value of the original `Option` values, or `None` if any of the elements was `None`.

```
let v = vec![Some(2), Some(4), None, Some(8)];
let res: Option<Vec<_>> = v.into_iter().collect();
assert_eq!(res, None);
let v = vec![Some(2), Some(4), Some(8)];
let res: Option<Vec<_>> = v.into_iter().collect();
assert_eq!(res, Some(vec![2, 4, 8]));
```

Run

`Option` also implements the `Product` and `Sum` traits, allowing an iterator over `Option` values to provide the `product` and `sum` methods.

```
let v = vec![None, Some(1), Some(2), Some(3)];
let res: Option<i32> = v.into_iter().sum();
assert_eq!(res, None);
let v = vec![Some(1), Some(2), Some(21)];
let res: Option<i32> = v.into_iter().product();
assert_eq!(res, Some(42));
```

Run

Modifying an `Option` in-place

These methods return a mutable reference to the contained value of an `Option<T>`:

- `insert` inserts a value, dropping any old contents
- `get_or_insert` gets the current value, inserting a provided default value if it is `None`
- `get_or_insert_default` gets the current value, inserting the default value of type `T` (which must implement `Default`) if it is `None`
- `get_or_insert_with` gets the current value, inserting a default computed by the provided function if it is `None`

These methods transfer ownership of the contained value of an `Option`:

- `take` takes ownership of the contained value of an `Option`, if any, replacing the `Option` with `None`
- `replace` takes ownership of the contained value of an `Option`, if any, replacing the `Option` with a `Some` containing the provided value

Examples

Basic pattern matching on `Option`:

```
let msg = Some("howdy");

// Take a reference to the contained string
if let Some(m) = &msg {
    println!("{}", *m);
}

// Remove the contained string, destroying the Option
let unwrapped_msg = msg.unwrap_or("default message");
```

Run

Initialize a result to `None` before a loop:

```
enum Kingdom { Plant(u32, &'static str), Animal(u32, &'static
```
Run
```
}

// A list of data to search through.
let all_the_big_things = [
    Kingdom::Plant(250, "redwood"),
    Kingdom::Plant(230, "noble fir"),
    Kingdom::Plant(229, "sugar pine"),
    Kingdom::Animal(25, "blue whale"),
    Kingdom::Animal(19, "fin whale"),
    Kingdom::Animal(15, "north pacific right whale"),
];

// We're going to search for the name of the biggest animal,
// but to start with we've just got `None`.
let mut name_of_biggest_animal = None;
let mut size_of_biggest_animal = 0;
for big_thing in &all_the_big_things {
    match *big_thing {
        Kingdom::Animal(size, name) if size > size_of_biggest_animal
            // Now we've found the name of some big animal
            size_of_biggest_animal = size;
            name_of_biggest_animal = Some(name);
        }
        Kingdom::Animal(..) | Kingdom::Plant(..) => ()
    }
}

match name_of_biggest_animal {
    Some(name) => println!("the biggest animal is {}", name),
    None => println!("there are no animals :("),
}
```

Structs

IntoIter
: An iterator over the value in `Some` variant of an `Option`.

Iter
: An iterator over a reference to the `Some` variant of an `Option`.

IterMut
: An iterator over a mutable reference to the `Some` variant of an `Option`.

Enums

on

The `Option` type. See the module level documentation for more.