More utility traits

Chapter 12 & 13

Daniël de Kok

Today

Today, we will look at more important utility traits:

- `Sized` for distinguishing sized and unsized types.
- `Copy` and `Clone` for copying data structures.
- `Deref` and `DerefMut` for dereferencing.
- `From` and `Into` for type conversions.

Sized

Introduction

A sized type is a type that will always have the same size in memory. For example:

- `usize`: 4/8 bytes
- `String`: pointer + capacity + length
- `&[u8]`: pointer + length
- `&str`: pointer + length

All sized types automatically implement the `Sized` trait as a marker.

Unsized types

Unsized types do not always have the same size:

- `str`: number of bytes necessary to store the string.
- `[u8]`: number of bytes in the u8 slice.

Unsized types

Unsized types do not always have the same size:

- `str`: number of bytes necessary to store the string.
- `[u8]`: number of bytes in the u8 slice.

This is why we can't have a `str` or `[u8]` as on the stack or as a struct member: amount of memory necessary varies.

Unsized types

Unsized types do not always have the same size:

- `str`: number of bytes necessary to store the string.
- `[u8]`: number of bytes in the u8 slice.

This is why we can't have a `str` or `[u8]` as on the stack or as a struct member: amount of memory necessary varies.

The `?Sized` trait contains the set of sized and unsized types.

Example

```
pub struct Box<T: ?Sized>(Unique<T>);
```

Since the data that `Box` points to is on the heap, it does not need to be sized.

Why is `?Sized` necessary?

- All type parameters have an implicit bound of `Sized`.

Why is `?Sized` necessary?

- All type parameters have an implicit bound of `Sized`.
- Exception: implicit `Self` type of a trait.

Why is `?Sized` necessary?

- All type parameters have an implicit bound of `Sized`.
- Exception: implicit `Self` type of a trait.
- If we want to use an sized and unsized types as a type parameter: require the `?Sized` trait for the type parameter.

Example

```
trait Concat<T> {
    type Output;
    fn concat(&self, v: &T) -> Self::Output;
}

// Error: `str` does not have a constant size known at compile-time
impl Concat<str> for str {
    type Output = String;

    fn concat(&self, v: &str) -> Self::Output {
        let mut s = String::from(self);
        s.push_str(v);
        s
    }
}
```

Fix

```
trait Concat<T> where T: ?Sized {
    type Output;
    fn concat(&self, v: &T) -> Self::Output;
}

impl Concat<str> for str {
    type Output = String;

    fn concat(&self, v: &str) -> Self::Output {
        let mut s = String::from(self);
        s.push_str(v);
        s
    }
}
```

Copy/Clone

Introduction

- Types that implement `Copy` are copy types.

Introduction

- Types that implement `Copy` are copy types.
- Types that do not implement `Copy` are move types.

Introduction

- Types that implement `Copy` are copy types.
- Types that do not implement `Copy` are move types.
- Often useful to make a copy of a move type: `Clone`

Introduction

- Types that implement `Copy` are copy types.
- Types that do not implement `Copy` are move types.
- Often useful to make a copy of a move type: `Clone`

Both traits can be derived automatically, but requires fields to be `Copy` or `Clone` as well.

Clone

```
trait Clone: Sized {
  fn clone(&self) -> Self;

  fn clone_from(&mut self, source: &Self) {
      *self = source.clone()
  }
}
```

Implementation

- Implementing `Clone` manually is often necessary with fields that are not `Clone` types.

Implementation

- Implementing `Clone` manually is often necessary with fields that are not `Clone` types.
- Real-world example:

```
pub struct ZipfRangeGenerator<R> {
    upper_bound: usize,
    rng: R,
    dist: ZipfDistribution,
}
```

Implementation

- Implementing `Clone` manually is often necessary with fields that are not `Clone` types.
- Real-world example:

```
pub struct ZipfRangeGenerator<R> {
    upper_bound: usize,
    rng: R,
    dist: ZipfDistribution,
}
```

- `ZipfDistribution` from the `zipf` crate does not implement `Clone`.

Implementation

```
impl<R> Clone for ZipfRangeGenerator<R>
where
    R: Clone,
{
    fn clone(&self) -> Self {
        ZipfRangeGenerator {
            upper_bound: self.upper_bound,
            rng: self.rng.clone(),
            dist: ZipfDistribution::new(self.upper_bound,
                ZIPF_RANGE_GENERATOR_EXPONENT).unwrap(),
        }
    }
}
```

Clone and generic types

```
impl<R> Clone for ZipfRangeGenerator<R>
where
    R: Clone,
{
    // ...
}
```

Clone is only defined for ZipfRangeGenerator if R implements Clone.

Copy

- Reminder: copy types are always bitwise copied, they do not move.

Copy

- Reminder: copy types are always bitwise copied, they do not move.
- Only for types that are safe to bit-copy.

Copy

- Reminder: copy types are always bitwise copied, they do not move.
- Only for types that are safe to bit-copy.
- Can only be implemented/derived on types that have `Copy` members.

Copy

- Reminder: copy types are always bitwise copied, they do not move.
- Only for types that are safe to bit-copy.
- Can only be implemented/derived on types that have `Copy` members.
- `Copy` definition:

```
pub trait Copy: Clone { }
```

Copy

- Reminder: copy types are always bitwise copied, they do not move.
- Only for types that are safe to bit-copy.
- Can only be implemented/derived on types that have `Copy` members.
- `Copy` definition:

```
pub trait Copy: Clone { }
```

- `Clone` is a marker trait.

Example (`Vec<T>`)

- `Vec` is **not** a copy type.
- `Vec` consists of:
  - A pointer to a heap array.
  - Length
  - Capacity

Example (`Vec<T>`)

- `Vec` is **not** a copy type.
- `Vec` consists of:
  - A pointer to a heap array.
  - Length
  - Capacity
- A `Vec` instance owns its heap array.

Example (`Vec<T>`)

- `Vec` is **not** a copy type.
- `Vec` consists of:
  - A pointer to a heap array.
  - Length
  - Capacity
- A `Vec` instance owns its heap array.
- If `Vec` implemented copy: bitwise copy of the pointer.

Example (`Vec<T>`)

- `Vec` is **not** a copy type.
- `Vec` consists of:
  - A pointer to a heap array.
  - Length
  - Capacity
- A `Vec` instance owns its heap array.
- If `Vec` implemented copy: bitwise copy of the pointer.
- Heap array owned by two `Vec`s: unsound, double-free on drop.

Example (`&[T]`)

- The slice reference `&[T]` is a copy type.
- `&[T]`
  - A pointer to an array.
  - Length
- A slice reference does not own the array

Example (`&[T]`)

- The slice reference `&[T]` is a copy type.
- `&[T]`
  - A pointer to an array.
  - Length
- A slice reference does not own the array
- Bitwise copy is fine: does not create two owners of the same data.

Deriving Copy

```rust
#[derive(Clone, Copy)]
struct Rectangle {
    x: usize,
    y: usize,
    width: usize,
    height: usize,
}
```

In-class assignment

- What is the cost of cloning a `Vec` using `Clone`?
- What is the benefit of using `clone_from` on a `Vec`?
- Investigate whether `File` implements clone. How? Why not?

Deref/DerefMut

Introduction

The dereference operators `*` and `.` are syntactic sugar for `Deref` and `DerefMut`:

```
let mut b = Box::new("hello".to_string());

b.len(); // Desugars to:
b.deref().len()
```

Introduction

The dereference operators `*` and `.` are syntactic sugar for `Deref` and `DerefMut`:

```
let mut b = Box::new("hello".to_string());

b.len(); // Desugars to:
b.deref().len()

*b; // Desugars to:
*b.deref();
```

Introduction

The dereference operators `*` and `.` are syntactic sugar for `Deref` and `DerefMut`:

```
let mut b = Box::new("hello".to_string());

b.len(); // Desugars to:
b.deref().len()

*b; // Desugars to:
*b.deref();

b.push_str(" world"); // Desugars to:
b.deref_mut().push_str(" world");
```

Introduction

The dereference operators `*` and `.` are syntactic sugar for `Deref` and `DerefMut`:

```
let mut b = Box::new("hello".to_string());

b.len(); // Desugars to:
b.deref().len()

*b; // Desugars to:
*b.deref();

b.push_str(" world"); // Desugars to:
b.deref_mut().push_str(" world");

*b = "something else".to_string(); // Desugars to:
*b.deref_mut();
```

Deref

```
pub trait Deref {
    type Target: ?Sized;

    fn deref(&self) -> &Self::Target;
}
```

DerefMut

```
pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Nullable pointer

Let's make a nullable pointer type!

Nullable pointer

Let's make a nullable pointer type!

Don't do this in real projects, unless you have an extremely good reason!

Nullable pointer

Let's make a nullable pointer type!

Don't do this in real projects, unless you have an extremely good reason!

```
struct NullablePointer<T>(Option<Box<T>>);
```

Nullable pointer: `Default`

```
impl<T> Default for NullablePointer<T> {
    fn default() -> Self {
        NullablePointer(None)
    }
}
```

Nullable pointer: constructor

```
impl<T> NullablePointer<T> {
    fn new(t: T) -> Self {
        NullablePointer(Some(Box::new(t)))
    }
}
```

Nullable pointer: Deref

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

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

    fn deref(&self) -> &Self::Target {
        // What's a nullable pointer without a null
        // pointer exception? Erm. Panic.
        self.0.as_ref()
              .expect("Dereferenced a null pointer")
    }
}
```

Nullable pointer: DerefMut

```
use std::ops::DerefMut;

impl<T> DerefMut for NullablePointer<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        // What's a nullable pointer without a null
        // pointer exception? Erm. Panic.
        self.0.as_mut()
              .expect("Dereferenced a null pointer")
    }
}
```

Pitfall

```
let p = NullablePointer::new(10usize);
```

What is the type of `p.clone()`?

Pitfall

```
let p = NullablePointer::new(10usize);
```

What is the type of `p.clone()`?

`usize`

Fix

```
#[derive(Clone)]
struct NullablePointer<T>(Option<Box<T>>);
```

Fun fact: `NullablePointer` memory layout

```
struct NullablePointer<T>(Option<Box<T>>);
```

Fun fact: `NullablePointer` memory layout

```
struct NullablePointer<T>(Option<Box<T>>);
```

Expectation:

- `NullablePointer` gets the size of `Option`
- `Option`:
  - Tag: 1 byte
  - Pointer: 4/8 bytes
  - Optional: padding

Fun fact: `NullablePointer` memory layout

```
pub fn null_usize() -> NullablePointer<usize> {
    Default::default()
}
```

Fun fact: `NullablePointer` memory layout

```
pub fn null_usize() -> NullablePointer<usize> {
    Default::default()
}
```

```
example::null_usize:
  push rbp
  mov rbp, rsp
  xor eax, eax
  pop rbp
  ret
```

Fun fact: `NullablePointer` memory layout

```
struct NullablePointer<T>(Option<Box<T>>);
```

- Actual memory layout: pointer

Fun fact: `NullablePointer` memory layout

```
struct NullablePointer<T>(Option<Box<T>>);
```

- Actual memory layout: pointer
- This optimization is made `Options` of non-nullable pointers.

Deref coercion

If you have a type `T` that implements `Deref<Target = U>`, values of `&T` will be coerced automatically to `&U`. For example:

Deref coercion

If you have a type `T` that implements `Deref<Target = U>`, values of `&T` will be coerced automatically to `&U`. For example:

```
fn greet_name(name: &str) {
    println!("Hello {}!", name);
}

let rust = "Rust".to_string();

// Works, because of deref coercion:
greet_name(&rust);
```

Deref and Vec

```
let mut v = vec![5, 4, 3, 2, 1];
v.sort();
```

- `sort` is not a method of `Vec<T>`, but of `&[T]`.

Deref and Vec

```
let mut v = vec![5, 4, 3, 2, 1];
v.sort();
```

- `sort` is not a method of `Vec<T>`, but of `&[T]`.
- Why does this work?

Deref and Vec

```
let mut v = vec![5, 4, 3, 2, 1];
v.sort();
```

- `sort` is not a method of `Vec<T>`, but of `&[T]`.
- Why does this work?
- `Vec<T>` derefs to `&[T]` and `&mut [T]`.

Deref and Vec

```
let mut v = vec![5, 4, 3, 2, 1];
v.sort();
```

- `sort` is not a method of `Vec<T>`, but of `&[T]`.
- Why does this work?
- `Vec<T>` derefs to `&[T]` and `&mut [T]`.
- The same applies for `String` and `&str`.

When to use

Follow Rust standard library documentation:

`Deref` **should only be implemented for smart pointers** to avoid confusion.

In-class assignment

Look again at the `peek_mut` method of `BinaryHeap`. How is `Deref`/`DerefMut` used here?

AsRef/AsMut

Introduction

If a type implements `AsRef<T>` → you can borrow `&T` from it efficiently:

```
pub trait AsRef<T> where T: ?Sized,
{
    fn as_ref(&self) -> &T;
}

pub trait AsMut<T> where T: ?Sized,
{
    fn as_mut(&mut self) -> &mut T;
}
```

Example

```
impl File {
    pub fn open<P: AsRef<Path>>(path: P) -> Result<File> {
        // ...
    }
}
```

- Can take many different string types, such as `&str`, `String`, `OsStr`, and `OsString`.

Example

```
impl File {
    pub fn open<P: AsRef<Path>>(path: P) -> Result<File> {
        // ...
    }
}
```

- Can take many different string types, such as `&str`, `String`, `OsStr`, and `OsString`.
- Can all be borrowed efficiently as `Path`.

Example

```
impl File {
    pub fn open<P: AsRef<Path>>(path: P) -> Result<File> {
        // ...
    }
}
```

- Can take many different string types, such as `&str`, `String`, `OsStr`, and `OsString`.
- Can all be borrowed efficiently as `Path`.
- Why not just take `&str`?

Example

```
impl File {
    pub fn open<P: AsRef<Path>>(path: P) -> Result<File> {
        // ...
    }
}
```

- Can take many different string types, such as `&str`, `String`, `OsStr`, and `OsString`.
- Can all be borrowed efficiently as `Path`.
- Why not just take `&str`? Not all file names are valid UTF-8!

From/Into

Introduction

`From` and `Into` are traits that are used for conversions between types that cannot fail:

Introduction

`From` and `Into` are traits that are used for conversions between types that cannot fail:

```
pub trait From<T>: Sized {
    fn from(_: T) -> Self;
}

pub trait Into<T>: Sized {
    fn into(self) -> T;
}
```

Introduction

`From` and `Into` are traits that are used for conversions between types that cannot fail:

```
pub trait From<T>: Sized {
    fn from(_: T) -> Self;
}

pub trait Into<T>: Sized {
    fn into(self) -> T;
}
```

- `From` specifies how a type can create itself from another type.

Introduction

`From` and `Into` are traits that are used for conversions between types that cannot fail:

```
pub trait From<T>: Sized {
    fn from(_: T) -> Self;
}

pub trait Into<T>: Sized {
    fn into(self) -> T;
}
```

- `From` specifies how a type can create itself from another type.
- `Into` specifies how a type can convert itself to another type.

Example

```
let owned_string = String::from("hello");
let owned_string2: String = "hello".into();
assert_eq!("hello", owned_string);
assert_eq!("hello", owned_string2);
```

Relation between `From`/`Into`

`Into` is implemented for all `From` types:

```
impl<T, U> Into<U> for T where U: From<T>
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

Which one to implement?

Always implement `From`:

- `Into` is implemented for all `From` types;
- `From` is more general.

A Vec wrapper

```rust
#[derive(Debug)]
pub struct MyVec<T>(Vec<T>);

impl Into<MyVec<usize>> for Vec<usize> {
    fn into(self) -> MyVec<usize> {
        MyVec(self)
    }
}
```

Trait `impl` coherency rule

```
impl Into<MyVec<usize>> for Vec<usize> { ... }
```

Course grained rule: impl blocks only possible for local types or traits.

Trait `impl` coherency rule

```
impl Into<MyVec<usize>> for Vec<usize> { ... }
```

Course grained rule: impl blocks only possible for local types or traits.

Seems to reject this `impl` block.

Trait `impl` coherency rule

Fine-grained `impl` rule for foreign traits[1]:

[1] https://doc.rust-lang.org/stable/error-index.html#E0210

Trait `impl` coherency rule

Fine-grained `impl` rule for foreign traits[1]:

Consider an `impl`:

```
impl<P1, ..., Pm> ForeignTrait<T1, ..., Tn> for T0 { ... }
```

where `P1, ..., Pm` are the type parameters of the impl and `T0, ..., Tn` are types. One of the types `T0, ..., Tn` must be a local type. Let `i` be the smallest integer such that `Ti` is a local type. Then no type parameter can appear in any of the `Tj` for `j < i`.

[1] https://doc.rust-lang.org/stable/error-index.html#E0210

Into for MyVec

```
impl Into<MyVec<usize>> for Vec<usize> { ... }
```

Fine: T1 is the local type, but no type parameter appears in T0.

Limitation of `Into` with type parameters

What if we want to make the `Into` implementation generic?

Limitation of `Into` with type parameters

What if we want to make the `Into` implementation generic?

```
pub struct MyVec<T>(Vec<T>);

// Compiler error: type parameter `T` must be used
// as the type parameter for some local type
impl<T> Into<MyVec<T>> for Vec<T> {
    fn into(self) -> MyVec<T> {
        MyVec(self)
    }
}
```

Limitation of `Into` with type parameters

What if we want to make the `Into` implementation generic?

```
pub struct MyVec<T>(Vec<T>);

// Compiler error: type parameter `T` must be used
// as the type parameter for some local type
impl<T> Into<MyVec<T>> for Vec<T> {
    fn into(self) -> MyVec<T> {
        MyVec(self)
    }
}
```

First local type is in `T1`, first type parameter appears in `T0`.

Implementation with `From`

```
impl<T> From<Vec<T>> for MyVec<T> {
    fn from(v: Vec<T>) -> MyVec<T> {
        MyVec(v)
    }
}
```

Implementation with `From`

```
impl<T> From<Vec<T>> for MyVec<T> {
    fn from(v: Vec<T>) -> MyVec<T> {
        MyVec(v)
    }
}
```

Compiles fine: `T0` is the local type.

In-class assignment

Implement `From` such that you can convert `NotNaN` values back to `f32`.