Introduction

Participation

If you are interested in contributing to this book, ch

Design patterns

In software development, we often come across pro
regardless of the environment they appear in. Altho
crucial to solve the task at hand, we may abstract fr
common practices that are generically applicable.

Design patterns are a collection of reusable and test
engineering. They make our software more modula
Moreover, these patterns provide a common langua
excellent tool for effective communication when pro

Design patterns in Rust

Rust is not object-oriented, and the combination of
functional elements, a strong type system, and the b
Because of this, Rust design patterns vary with resp
oriented programming languages. That's why we de
enjoy reading it! The book is divided in three main c

- Idioms: guidelines to follow when coding. They
  community. You should break them only if you
- Design patterns: methods to solve common pr
- Anti-patterns: methods to solve common prob
  design patterns give us benefits, anti-patterns

Translations

We are utilizing mdbook-i18n-helper. Please read up translations in their repository

External translations

- 简体中文

If you want to add a translation, please open an issu

Idioms

Idioms are commonly used styles, guidelines and pa
community. Writing idiomatic code allows other dev
happening.

After all, the computer only cares about the machin
compiler. Instead, the source code is mainly benefic
this abstraction layer, why not make it more readab

Remember the KISS principle: “Keep It Simple, Stupi
best if they are kept simple rather than made comp
a key goal in design, and unnecessary complexity sh

> Code is there for humans, not computers, to und

Use borrowed types for a

Description

Using a target of a deref coercion can increase the f deciding which argument type to use for a function will accept more input types.

This is not limited to slice-able or fat pointer types. I the **borrowed type** over **borrowing the owned typ** over `&Vec<T>`, or `&T` over `&Box<T>`.

Using borrowed types you can avoid layers of indire owned type already provides a layer of indirection. I indirection, so a `&String` will have two layers of ind `&str` instead, and letting `&String` coerce to a `&str`

Example

For this example, we will illustrate some differences argument versus using a `&str`, but the ideas apply a `&[T]` or using a `&Box<T>` versus a `&T`.

Consider an example where we wish to determine i vowels. We don't need to own the string to determin

The code might look something like this:

```rust
fn three_vowels(word: &String) -> bool {
    let mut vowel_count = 0;
    for c in word.chars() {
        match c {
            'a' | 'e' | 'i' | 'o' | 'u' => {
                vowel_count += 1;
                if vowel_count >= 3 {
                    return true
                }
            }
            _ => vowel_count = 0
        }
    }
    false
}

fn main() {
    let ferris = "Ferris".to_string();
    let curious = "Curious".to_string();
    println!("{}: {}", ferris, three_vowels(&
    println!("{}: {}", curious, three_vowels(

    // This works fine, but the following two
    // println!("Ferris: {}", three_vowels("F
    // println!("Curious: {}", three_vowels("

}
```

This works fine because we are passing a `&String` t comments on the last two lines, the example will fai coerce to a `&String` type. We can fix this by simply

For instance, if we change our function declaration t

```rust
fn three_vowels(word: &str) -> bool {
```

then both versions will compile and print the same (

```
Ferris: false
Curious: true
```

But wait, that's not all! There is more to this story. It' that doesn't matter, I will never be using a `&'static` when we used `"Ferris"`). Even ignoring this specia using `&str` will give you more flexibility than using &

Let's now take an example where someone gives us determine if any of the words in the sentence conta probably should make use of the function we have a each word from the sentence.

An example of this could look like this:

```
fn three_vowels(word: &str) -> bool {
    let mut vowel_count = 0;
    for c in word.chars() {
        match c {
            'a' | 'e' | 'i' | 'o' | 'u' => {
                vowel_count += 1;
                if vowel_count >= 3 {
                    return true
                }
            }
            _ => vowel_count = 0
        }
    }
    false
}

fn main() {
    let sentence_string =
        "Once upon a time, there was a frienc
Ferris".to_string();
    for word in sentence_string.split(' ') {
        if three_vowels(word) {
            println!("{} has three consecutiv
        }
    }
}
```

Running this example using our function declared w

```
curious has three consecutive vowels!
```

However, this example will not run when our functic `&String`. This is because string slices are a `&str` ar require an allocation to be converted to `&String` w converting from `String` to `&str` is cheap and impl

See also

- Rust Language Reference on Type Coercions
- For more discussion on how to handle `String` by Herman J. Radtke III

Concatenating strings w

Description

It is possible to build up strings using the `push` and
`String`, or using its `+` operator. However, it is ofte
especially where there is a mix of literal and non-lite

Example

```
fn say_hello(name: &str) -> String {
    // We could construct the result string m
    // let mut result = "Hello ".to_owned();
    // result.push_str(name);
    // result.push('!');
    // result

    // But using format! is better.
    format!("Hello {}!", name)
}
```

Advantages

Using `format!` is usually the most succinct and rea

Disadvantages

It is usually not the most efficient way to combine st
a mutable string is usually the most efficient (especi
allocated to the expected size).

Constructors

Description

Rust does not have constructors as a language cons
use an associated function `new` to create an object:

```
/// Time in seconds.
///
/// # Example
///
/// ```
/// let s = Second::new(42);
/// assert_eq!(42, s.value());
/// ```
pub struct Second {
    value: u64
}

impl Second {
    // Constructs a new instance of [`Second`
    // Note this is an associated function -
    pub fn new(value: u64) -> Self {
        Self { value }
    }

    /// Returns the value in seconds.
    pub fn value(&self) -> u64 {
        self.value
    }
}
```

Default Constructors

Rust supports default constructors with the `Defaul`

```rust
/// Time in seconds.
///
/// # Example
///
/// ```
/// let s = Second::default();
/// assert_eq!(0, s.value());
/// ```
pub struct Second {
    value: u64
}

impl Second {
    /// Returns the value in seconds.
    pub fn value(&self) -> u64 {
        self.value
    }
}

impl Default for Second {
    fn default() -> Self {
        Self { value: 0 }
    }
}
```

`Default` can also be derived if all types of all fields
`Second`:

```rust
/// Time in seconds.
///
/// # Example
///
/// ```
/// let s = Second::default();
/// assert_eq!(0, s.value());
/// ```
#[derive(Default)]
pub struct Second {
    value: u64
}

impl Second {
    /// Returns the value in seconds.
    pub fn value(&self) -> u64 {
        self.value
    }
}
```

Note: It is common and expected for types to imple
`new` constructor. `new` is the constructor conventior
so if it is reasonable for the basic constructor to take
it is functionally identical to default.

Hint: The advantage of implementing or deriving `D`
used where a `Default` implementation is required,
`*or_default` functions in the standard library.

See also

- The default idiom for a more in-depth descript
- The builder pattern for constructing objects wl
configurations.
- API Guidelines/C-COMMON-TRAITS for implem

The `Default` Trait

Description

Many types in Rust have a constructor. However, thi
abstract over "everything that has a `new()` method'
conceived, which can be used with containers and o
`Option::unwrap_or_default()`). Notably, some cor
applicable.

Not only do one-element containers like `Cow` , `Box`
contained `Default` types, one can automatically `#[`
fields all implement it, so the more types implement
becomes.

On the other hand, constructors can take multiple a
method does not. There can even be multiple const
there can only be one `Default` implementation per

Example

```rust
use std::{path::PathBuf, time::Duration};

// note that we can simply auto-derive Defaul
#[derive(Default, Debug, PartialEq)]
struct MyConfiguration {
    // Option defaults to None
    output: Option<PathBuf>,
    // Vecs default to empty vector
    search_path: Vec<PathBuf>,
    // Duration defaults to zero time
    timeout: Duration,
    // bool defaults to false
    check: bool,
}

impl MyConfiguration {
    // add setters here
}

fn main() {
    // construct a new instance with default
    let mut conf = MyConfiguration::default()
    // do something with conf here
    conf.check = true;
    println!("conf = {:#?}", conf);

    // partial initialization with default va
    let conf1 = MyConfiguration {
        check: true,
        ..Default::default()
    };
    assert_eq!(conf, conf1);
}
```

See also

- The constructor idiom is another way to gener "default"
- The `Default` documentation (scroll down for t
- `Option::unwrap_or_default()`
- `derive(new)`

Collections are smart po

Description

Use the `Deref` trait to treat collections like smart pc
views of data.

Example

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

struct Vec<T> {
    data: RawVec<T>,
    //..
}

impl<T> Deref for Vec<T> {
    type Target = [T];

    fn deref(&self) -> &[T] {
        //..
    }
}
```

A `Vec<T>` is an owning collection of `T` s, while a slic
`T` s. Implementing `Deref` for `Vec` allows implicit de
and includes the relationship in auto-derefencing se
expect to be implemented for `Vec` s are instead imp

Also `String` and `&str` have a similar relation.

Motivation

Ownership and borrowing are key aspects of the Ru
account for these semantics properly to give a good
a data structure that owns its data, offering a borrov
flexible APIs.

Advantages

Most methods can be implemented only for the bor
available for the owning view.

Gives clients a choice between borrowing or taking

Disadvantages

Methods and traits only available via dereferencing
bounds checking, so generic programming with dat
complex (see the `Borrow` and `AsRef` traits, etc.).

Discussion

Smart pointers and collections are analogous: a sm
whereas a collection points to many objects. From t
there is little difference between the two. A collectio
access each datum is via the collection and the colle
data (even in cases of shared ownership, some kind
appropriate). If a collection owns its data, it is usuall
as borrowed so that it can be referenced multiple ti

Most smart pointers (e.g., `Foo<T>`) implement `Dere`
will usually dereference to a custom type. `[T]` and
but in the general case, this is not necessary. `Foo<T`
`Deref<Target=Bar<T>>` where `Bar` is a dynamically
borrowed view of the data in `Foo<T>` .

Commonly, ordered collections will implement `Ind`
syntax. The target will be the borrowed view.

See also

- Deref polymorphism anti-pattern.
- Documentation for `Deref` trait.

Finalisation in destructo

Description

Rust does not provide the equivalent to `finally` bl
matter how a function is exited. Instead, an object's
that must be run before exit.

Example

```
fn bar() -> Result<(), ()> {
    // These don't need to be defined inside
    struct Foo;

    // Implement a destructor for Foo.
    impl Drop for Foo {
        fn drop(&mut self) {
            println!("exit");
        }
    }

    // The dtor of _exit will run however the
    let _exit = Foo;
    // Implicit return with `?` operator.
    baz()?;
    // Normal return.
    Ok(())
}
```

Motivation

If a function has multiple return points, then execut
repetitive (and thus bug-prone). This is especially th
a macro. A common case is the `?` operator which re
continues if it is `Ok`. `?` is used as an exception hand
(which has `finally`), there is no way to schedule co
exceptional cases. Panicking will also exit a function

Advantages

Code in destructors will (nearly) always be run - cop

Disadvantages

It is not guaranteed that destructors will run. For ex
function or if running a function crashes before exit
case of a panic in an already panicking thread. There
as finalizers where it is absolutely essential that fina

This pattern introduces some hard to notice, implici
clear indication of destructors to be run on exit. This

Requiring an object and `Drop` impl just for finalisati

Discussion

There is some subtlety about how exactly to store th
be kept alive until the end of the function and must
always be a value or uniquely owned pointer (e.g., `B`
`Rc`) is used, then the finalizer can be kept alive beyo
similar reasons, the finalizer should not be moved o

The finalizer must be assigned into a variable, other
rather than when it goes out of scope. The variable
variable is only used as a finalizer, otherwise the cor
never used. However, do not call the variable `_` wit
destroyed immediately.

In Rust, destructors are run when an object goes ou
reach the end of block, there is an early return, or th
Rust unwinds the stack running destructors for each
destructors get called even if the panic happens in a

If a destructor panics while unwinding, there is no g
thread immediately, without running further destru
not absolutely guaranteed to run. It also means that
destructors not to panic, since it could leave resourc

See also

RAII guards.

mem::{take(_), repl

owned values in change

Description

Say we have a `&mut MyEnum` which has (at least) twc `}` and `B { name: String }`. Now we want to chang while keeping `MyEnum::B` intact.

We can do this without cloning the `name`.

Example

```
use std::mem;

enum MyEnum {
    A { name: String, x: u8 },
    B { name: String }
}

fn a_to_b(e: &mut MyEnum) {
    if let MyEnum::A { name, x: 0 } = e {
        // This takes out our `name` and puts
        // (note that empty strings don't all
        // Then, construct the new enum varia
        // be assigned to `*e`).
        *e = MyEnum::B { name: mem::take(name
    }
}
```

This also works with more variants:

```
use std::mem;

enum MultiVariateEnum {
    A { name: String },
    B { name: String },
    C,
    D
}

fn swizzle(e: &mut MultiVariateEnum) {
    use MultiVariateEnum::*;
    *e = match e {
        // Ownership rules do not allow takir
        // take the value out of a mutable re
        A { name } => B { name: mem::take(nam
        B { name } => A { name: mem::take(nam
        C => D,
        D => C
    }
}
```

Motivation

When working with enums, we may want to change another variant. This is usually done in two phases t the first phase, we observe the existing value and lo next. In the second phase we may conditionally cha above).

The borrow checker won't allow us to take out `name` must be there.) We could of course `.clone()` name `MyEnum::B`, but that would be an instance of the Clo pattern. Anyway, we can avoid the extra allocation b borrow.

`mem::take` lets us swap out the value, replacing it w the previous value. For `String`, the default value is need to allocate. As a result, we get the original `nam` wrap this in another enum.

NOTE: `mem::replace` is very similar, but allows us t with. An equivalent to our `mem::take` line would be `String::new())`.

Note, however, that if we are using an `Option` and `Option`'s `take()` method provides a shorter and m

Advantages

Look ma, no allocation! Also you may feel like Indiar

Disadvantages

This gets a bit wordy. Getting it wrong repeatedly wi
The compiler may fail to optimize away the double s
performance as opposed to what you'd do in unsafe

Furthermore, the type you are taking needs to imple
the type you're working with doesn't implement this

Discussion

This pattern is only of interest in Rust. In GC'd langu
value by default (and the GC would keep track of ref
like C you'd simply alias the pointer and fix things la

However, in Rust, we have to do a little more work t
have one owner, so to take it out, we need to put so
replacing the artifact with a bag of sand.

See also

This gets rid of the Clone to satisfy the borrow checl

On-Stack Dynamic Dispa

Description

We can dynamically dispatch over multiple values, h
multiple variables to bind differently-typed objects.
we can use deferred conditional initialization, as see

Example

```
use std::io;
use std::fs;

// These must live longer than `readable`, ar
let (mut stdin_read, mut file_read);

// We need to ascribe the type to get dynamic
let readable: &mut dyn io::Read = if arg == '
    stdin_read = io::stdin();
    &mut stdin_read
} else {
    file_read = fs::File::open(arg)?;
    &mut file_read
};

// Read from `readable` here.
```

Motivation

Rust monomorphises code by default. This means a
for each type it is used with and optimized indepen
code on the hot path, it also bloats the code in place
essence, thus costing compile time and cache usage

Luckily, Rust allows us to use dynamic dispatch, but

Advantages

We do not need to allocate anything on the heap. N
something we won't use later, nor do we need to m
follows to work with both `File` or `Stdin`.

Disadvantages

The code needs more moving parts than the `Box`-b

```
// We still need to ascribe the type for dyna
let readable: Box<dyn io::Read> = if arg == '
    Box::new(io::stdin())
} else {
    Box::new(fs::File::open(arg)?)
};
// Read from `readable` here.
```

Discussion

Rust newcomers will usually learn that Rust require
use, so it's easy to overlook the fact that *unused* vari
works quite hard to ensure that this works out fine
dropped at the end of their scope.

The example meets all the constraints Rust places o

- All variables are initialized before using (in this
- Each variable only holds values of a single type
  `Stdin`, `file` is of type `File` and `readable` is
- Each borrowed value outlives all the reference

See also

- Finalisation in destructors and RAII guards can
  lifetimes.
- For conditionally filled `Option<&T>`s of (mutab
  `Option<T>` directly and use its `.as_ref()` me

FFI Idioms

Writing FFI code is an entire course in itself. Howeve
can act as pointers, and avoid traps for inexperience

This section contains idioms that may be useful whe

1. Idiomatic Errors - Error handling with integer c
   as `NULL` pointers)

2. Accepting Strings with minimal unsafe code

3. Passing Strings to FFI functions

Error Handling in FFI

Description

In foreign languages like C, errors are represented b
system allows much more rich error information to
a full type.

This best practice shows different kinds of error coc
usable way:

1. Flat Enums should be converted to integers an
2. Structured Enums should be converted to an ii
   message for detail.
3. Custom Error Types should become "transpare

Code Example

Flat Enums

```
enum DatabaseError {
    IsReadOnly = 1, // user attempted a write
    IOError = 2, // user should read the C er
    FileCorrupted = 3, // user should run a r
}

impl From<DatabaseError> for libc::c_int {
    fn from(e: DatabaseError) -> libc::c_int
        (e as i8).into()
    }
}
```

Structured Enums

```
pub mod errors {
    enum DatabaseError {
        IsReadOnly,
        IOError(std::io::Error),
        FileCorrupted(String), // message des
    }

    impl From<DatabaseError> for libc::c_int
        fn from(e: DatabaseError) -> libc::c_
            match e {
                DatabaseError::IsReadOnly =>
                DatabaseError::IOError(_) =>
                DatabaseError::FileCorrupted(
            }
        }
    }
}

pub mod c_api {
    use super::errors::DatabaseError;

    #[no_mangle]
    pub extern "C" fn db_error_description(
        e: *const DatabaseError
        ) -> *mut libc::c_char {

        let error: &DatabaseError = unsafe {
            // SAFETY: pointer lifetime is gr
frame
            &*e
        };

        let error_str: String = match error {
            DatabaseError::IsReadOnly => {
                format!("cannot write to reac
            }
            DatabaseError::IOError(e) => {
                format!("I/O Error: {}", e);
            }
            DatabaseError::FileCorrupted(s) =
                format!("File corrupted, run
            }
        };

        let c_error = unsafe {
            // SAFETY: copying error_str to a
            // character at the end
            let mut malloc: *mut u8 = libc::m
*mut _;

            if malloc.is_null() {
                return std::ptr::null_mut();
            }

            let src = error_str.as_bytes().as
```

```
                std::ptr::copy_nonoverlapping(src

                std::ptr::write(malloc.add(error_

                malloc as *mut libc::c_char
            };

            c_error
        }
    }
```

Custom Error Types

```
struct ParseError {
    expected: char,
    line: u32,
    ch: u16
}

impl ParseError { /* ... */ }

/* Create a second version which is exposed a
#[repr(C)]
pub struct parse_error {
    pub expected: libc::c_char,
    pub line: u32,
    pub ch: u16
}

impl From<ParseError> for parse_error {
    fn from(e: ParseError) -> parse_error {
        let ParseError { expected, line, ch }
        parse_error { expected, line, ch }
    }
}
```

Advantages

This ensures that the foreign language has clear acc compromising the Rust code's API at all.

Disadvantages

It's a lot of typing, and some types may not be able t

Accepting Strings

Description

When accepting strings via FFI through pointers, the
followed:

1. Keep foreign strings “borrowed”, rather than c
2. Minimize the amount of complexity and `unsaf`
C-style string to native Rust strings.

Motivation

The strings used in C have different behaviours to th

- C strings are null-terminated while Rust strings
- C strings can contain any arbitrary non-zero by
- C strings are accessed and manipulated using
interactions with Rust strings go through safe r

The Rust standard library comes with C equivalents `CString` and `&CStr`, that allow us to avoid a lot of t
involved in converting between C strings and Rust st

The `&CStr` type also allows us to work with borrowe
between Rust and C is a zero-cost operation.

Code Example

```
pub mod unsafe_module {

    // other module content

    /// Log a message at the specified level.
    ///
    /// # Safety
    ///
    /// It is the caller's guarantee to ensu
    ///
    /// - is not a null pointer
    /// - points to valid, initialized data
    /// - points to memory ending in a null 
    /// - won't be mutated for the duration 
    #[no_mangle]
    pub unsafe extern "C" fn mylib_log(
        msg: *const libc::c_char,
        level: libc::c_int
    ) {
        let level: crate::LogLevel = match l

        // SAFETY: The caller has already gu
        // `# Safety` section of the doc-com
        let msg_str: &str = match std::ffi::
            Ok(s) => s,
            Err(e) => {
                crate::log_error("FFI string
                return;
            }
        };

        crate::log(msg_str, level);
    }
}
```

Advantages

The example is is written to ensure that:

1. The `unsafe` block is as small as possible.
2. The pointer with an "untracked" lifetime becon

Consider an alternative, where the string is actually

```
pub mod unsafe_module {

    // other module content

    pub extern "C" fn mylib_log(msg: *const l
{
        // DO NOT USE THIS CODE.
        // IT IS UGLY, VERBOSE, AND CONTAINS

        let level: crate::LogLevel = match le

        let msg_len = unsafe { /* SAFETY: str
            libc::strlen(msg)
        };

        let mut msg_data = Vec::with_capacity

        let msg_cstr: std::ffi::CString = uns
            // SAFETY: copying from a foreign
            // for the entire stack frame int
            std::ptr::copy_nonoverlapping(msg

            msg_data.set_len(msg_len + 1);

            std::ffi::CString::from_vec_with_
        }

        let msg_str: String = unsafe {
            match msg_cstr.into_string() {
                Ok(s) => s,
                Err(e) => {
                    crate::log_error("FFI str
                    return;
                }
            }
        };

        crate::log(&msg_str, level);
    }
}
```

This code in inferior to the original in two respects:

1. There is much more `unsafe` code, and more i uphold.
2. Due to the extensive arithmetic required, there Rust `undefined behaviour`.

The bug here is a simple mistake in pointer arithme bytes of it. However, the `NUL` terminator at the end

The Vector then had its size *set* to the length of the z to it, which could have added a zero at the end. As a

uninitialized memory. When the `CString` is created
the Vector will cause `undefined behaviour` !

Like many such issues, this would be difficult issue t
panic because the string was not `UTF-8` , sometimes
end of the string, sometimes it would just completel

Disadvantages

None?

Passing Strings

Description

When passing strings to FFI functions, there are fou

1. Make the lifetime of owned strings as long as p
2. Minimize `unsafe` code during the conversion.
3. If the C code can modify the string data, use `V
4. Unless the Foreign Function API requires it, the transfer to the callee.

Motivation

Rust has built-in support for C-style strings with its there are different approaches one can take with st function call from a Rust function.

The best practice is simple: use `CString` in such a w However, a secondary caveat is that *the object must* should be maximized. In addition, the documentatio `CString` after modification is UB, so additional wor

Code Example

```
pub mod unsafe_module {

    // other module content

    extern "C" {
        fn seterr(message: *const libc::c_cha
        fn geterr(buffer: *mut libc::c_char,
libc::c_int;
    }

    fn report_error_to_ffi<S: Into<String>>(
        err: S
    ) -> Result<(), std::ffi::NulError>{
        let c_err = std::ffi::CString::new(er

        unsafe {
            // SAFETY: calling an FFI whose c
            // const, so no modification shou
            seterr(c_err.as_ptr());
        }

        Ok(())
        // The lifetime of c_err continues un
    }

    fn get_error_from_ffi() -> Result<String,
        let mut buffer = vec![0u8; 1024];
        unsafe {
            // SAFETY: calling an FFI whose c
            // that the input need only live
            let written: usize = geterr(buffe

            buffer.truncate(written + 1);
        }

        std::ffi::CString::new(buffer).unwrap
    }
}
```

Advantages

The example is written in a way to ensure that:

1. The `unsafe` block is as small as possible.
2. The `CString` lives long enough.
3. Errors with typecasts are always propagated w

A common mistake (so common it's in the documen
first block:

```
pub mod unsafe_module {

    // other module content

    fn report_error<S: Into<String>>(err: S)
std::ffi::NulError> {
        unsafe {
            // SAFETY: whoops, this contains
            seterr(std::ffi::CString::new(err
        }
        Ok(())
    }
}
```

This code will result in a dangling pointer, because t
extended by the pointer creation, unlike if a referen

Another issue frequently raised is that the initializat
However, recent versions of Rust actually optimize t
`zmalloc`, meaning it is as fast as the operating syste
(which is quite fast).

Disadvantages

None?

Iterating over an `Optio`

Description

`Option` can be viewed as a container that contains
particular, it implements the `IntoIterator` trait, an
code that needs such a type.

Examples

Since `Option` implements `IntoIterator`, it can be

```
let turing = Some("Turing");
let mut logicians = vec!["Curry", "Kleene", '

logicians.extend(turing);

// equivalent to
if let Some(turing_inner) = turing {
    logicians.push(turing_inner);
}
```

If you need to tack an `Option` to the end of an exist
`.chain()`:

```
let turing = Some("Turing");
let logicians = vec!["Curry", "Kleene", "Mark

for logician in logicians.iter().chain(turing
    println!("{} is a logician", logician);
}
```

Note that if the `Option` is always `Some`, then it is m
`std::iter::once` on the element instead.

Also, since `Option` implements `IntoIterator`, it's p
loop. This is equivalent to matching it with `if let S`
should prefer the latter.

See also

- `std::iter::once` is an iterator which yields ex
readable alternative to `Some(foo).into_iter(`
- `Iterator::filter_map` is a version of `Iterat`
functions which return `Option`.
- The `ref_slice` crate provides functions for co
element slice.
- Documentation for `Option<T>`

Pass variables to closure

Description

By default, closures capture their environment by b
closure to move whole environment. However, ofte
variables to closure, give it copy of some data, pass
other transformation.

Use variable rebinding in separate scope for that.

Example

Use

```
use std::rc::Rc;

let num1 = Rc::new(1);
let num2 = Rc::new(2);
let num3 = Rc::new(3);
let closure = {
    // `num1` is moved
    let num2 = num2.clone();  // `num2` is cl
    let num3 = num3.as_ref();  // `num3` is b
    move || {
        *num1 + *num2 + *num3;
    }
};
```

instead of

```
use std::rc::Rc;

let num1 = Rc::new(1);
let num2 = Rc::new(2);
let num3 = Rc::new(3);

let num2_cloned = num2.clone();
let num3_borrowed = num3.as_ref();
let closure = move || {
    *num1 + *num2_cloned + *num3_borrowed;
};
```

Advantages

Copied data are grouped together with closure defi
and they will be dropped immediately even if they a

Closure uses same variable names as surrounding c
moved.

Disadvantages

Additional indentation of closure body.

`#[non_exhaustive]` an
for extensibility

Description

A small set of scenarios exist where a library author
public struct or new variants to an enum without br

Rust offers two solutions to this problem:

- Use `#[non_exhaustive]` on `struct` s, `enum` s,
  documentation on all the places where `#[non_`
  docs.
- You may add a private field to a struct to preve
  or matched against (see Alternative)

Example

```rust
mod a {
    // Public struct.
    #[non_exhaustive]
    pub struct S {
        pub foo: i32,
    }

    #[non_exhaustive]
    pub enum AdmitMoreVariants {
        VariantA,
        VariantB,
        #[non_exhaustive]
        VariantC { a: String }
    }
}

fn print_matched_variants(s: a::S) {
    // Because S is `#[non_exhaustive]`, it c
    // we must use `..` in the pattern.
    let a::S { foo: _, ..} = s;

    let some_enum = a::AdmitMoreVariants::Var
    match some_enum {
        a::AdmitMoreVariants::VariantA => pr
        a::AdmitMoreVariants::VariantB => pr

        // .. required because this variant
        a::AdmitMoreVariants::VariantC { a, .

        // The wildcard match is required bec
        // added in the future
        _ => println!("it's a new variant")
    }
}
```

Alternative: `Private fields` fo

`#[non_exhaustive]` only works across crate bound method may be used.

Adding a field to a struct is a mostly backwards com uses a pattern to deconstruct a struct instance, they and adding a new one would break that pattern. Th use `..` in the pattern, in which case adding another Making at least one of the struct's fields private forc patterns, ensuring that the struct is future-proof.

The downside of this approach is that you might ne field to the struct. You can use the `()` type so that t prepend _ to the field name to avoid the unused fi

```
pub struct S {
    pub a: i32,
    // Because `b` is private, you cannot mat
`S`
    //  cannot be directly instantiated or ma
    _b: ()
}
```

Discussion

On `struct`s, `#[non_exhaustive]` allows adding add compatible way. It will also prevent clients from usir the fields are public. This may be helpful, but it's wo additional field to be found by clients as a compiler be silently undiscovered.

`#[non_exhaustive]` can be applied to enum variant variant behaves in the same way as a `#[non_exhaus`

Use this deliberately and with caution: incrementing or variants is often a better option. `#[non_exhausti` where you're modeling an external resource that ma library, but is not a general purpose tool.

Disadvantages

`#[non_exhaustive]` can make your code much less forced to handle unknown enum variants. It should evolutions are required **without** incrementing the r

When `#[non_exhaustive]` is applied to `enum`s, it fo variant. If there is no sensible action to take in this c and code paths that are only executed in extremely to `panic!()` in this scenario, it may have been bett time. In fact, `#[non_exhaustive]` forces clients to ha is rarely a sensible action to take in this scenario.

See also

- RFC introducing #[non_exhaustive] attribute fo

Easy doc initialization

Description

If a struct takes significant effort to initialize when w
your example with a helper function which takes th

Motivation

Sometimes there is a struct with multiple or complic
methods. Each of these methods should have exam

For example:

```
struct Connection {
    name: String,
    stream: TcpStream,
}

impl Connection {
    /// Sends a request over the connection.
    ///
    /// # Example
    /// ```no_run
    /// # // Boilerplate are required to get
    /// # let stream = TcpStream::connect("12
    /// # let connection = Connection { name:
    /// # let request = Request::new("Request
"payload");
    /// let response = connection.send_reques
    /// assert!(response.is_ok());
    /// ```
    fn send_request(&self, request: Request)
        // ...
    }

    /// Oh no, all that boilerplate needs to
    fn check_status(&self) -> Status {
        // ...
    }
}
```

Example

Instead of typing all of this boilerplate to create a `Co`
just create a wrapping helper function which takes t

```
struct Connection {
    name: String,
    stream: TcpStream,
}

impl Connection {
    /// Sends a request over the connection.
    ///
    /// # Example
    /// ```
    /// # fn call_send(connection: Connection
    /// let response = connection.send_reques
    /// assert!(response.is_ok());
    /// # }
    /// ```
    fn send_request(&self, request: Request)
        // ...
    }
}
```

Note in the above example the line `assert!(respon`
while testing because it is inside a function which is

Advantages

This is much more concise and avoids repetitive coc

Disadvantages

As example is in a function, the code will not be test
make sure it compiles when running a `cargo test`.
you need `no_run`. With this, you do not need to adc

Discussion

If assertions are not required this pattern works wel

If they are, an alternative can be to create a public n
which is annotated with `#[doc(hidden)]` (so that us

can be called inside of rustdoc because it is part of t

Temporary mutability

Description

Often it is necessary to prepare and process some c
inspected and never modified. The intention can be
mutable variable as immutable.

It can be done either by processing data within a ne
variable.

Example

Say, vector must be sorted before usage.

Using nested block:

```
let data = {
    let mut data = get_vec();
    data.sort();
    data
};

// Here `data` is immutable.
```

Using variable rebinding:

```
let mut data = get_vec();
data.sort();
let data = data;

// Here `data` is immutable.
```

Advantages

Compiler ensures that you don't accidentally mutate

Disadvantages

Nested block requires additional indentation of bloc
from block or redefine variable.

Return consumed argun

Description

If a fallible function consumes (moves) an argument
error.

Example

```
pub fn send(value: String) -> Result<(), Senc
    println!("using {value} in a meaningful w
    // Simulate non-deterministic fallible ac
    use std::time::SystemTime;
    let period =
SystemTime::now().duration_since(SystemTime::
    if period.subsec_nanos() % 2 == 1 {
        Ok(())
    } else {
        Err(SendError(value))
    }
}

pub struct SendError(String);

fn main() {
    let mut value = "imagine this is very lor

    let success = 's: {
        // Try to send value two times.
        for _ in 0..2 {
            value = match send(value) {
                Ok(()) => break 's true,
                Err(SendError(value)) => valu
            }
        }
        false
    };

    println!("success: {}", success);
}
```

Motivation

In case of error you may want to try some alternativ non-deterministic function. But if the argument is al clone it on every call, which is not very efficient.

The standard library uses this approach in e.g. `Stri` a vector that doesn't contain valid UTF-8, a `FromUtf` original vector back using `FromUtf8Error::into_by`

Advantages

Better performance because of moving arguments

Disadvantages

Slightly more complex error types.

Design Patterns

Design patterns are "general reusable solutions to a
a given context in software design". Design patterns
culture of a programming language. Design pattern:
pattern in one language may be unnecessary in ano
impossible to express due to a missing feature.

If overused, design patterns can add unnecessary co
are a great way to share intermediate and advancec
programming language.

Design patterns in Rust

Rust has many unique features. These features give
classes of problems. Some of them are also pattern

YAGNI

YAGNI is an acronym that stands for `You Aren't Gc`
design principle to apply as you write code.

> The best code I ever wrote was code I never wrot

If we apply YAGNI to design patterns, we see that th
out many patterns. For instance, there is no need fo
we can just use traits.

Behavioural Patterns

From Wikipedia:

> Design patterns that identify common communic
> doing so, these patterns increase flexibility in car

Command

Description

The basic idea of the Command pattern is to separa
pass them as parameters.

Motivation

Suppose we have a sequence of actions or transacti
these actions or commands to be executed or invok
time. These commands may also be triggered as a r
when a user pushes a button, or on arrival of a data
might be undoable. This may come in useful for ope
to store logs of executed commands so that we cou
system crashes.

Example

Define two database operations `create table` and
is a command which knows how to undo the comm
`field` . When a user invokes a database migration o
executed in the defined order, and when the user in
whole set of commands is invoked in reverse order.

Approach: Using trait objects

We define a common trait which encapsulates our c
`execute` and `rollback` . All command `structs` mu

```rust
pub trait Migration {
    fn execute(&self) -> &str;
    fn rollback(&self) -> &str;
}

pub struct CreateTable;
impl Migration for CreateTable {
    fn execute(&self) -> &str {
        "create table"
    }
    fn rollback(&self) -> &str {
        "drop table"
    }
}

pub struct AddField;
impl Migration for AddField {
    fn execute(&self) -> &str {
        "add field"
    }
    fn rollback(&self) -> &str {
        "remove field"
    }
}

struct Schema {
    commands: Vec<Box<dyn Migration>>,
}

impl Schema {
    fn new() -> Self {
        Self { commands: vec![] }
    }

    fn add_migration(&mut self, cmd: Box<dyn
        self.commands.push(cmd);
    }

    fn execute(&self) -> Vec<&str> {
        self.commands.iter().map(|cmd| cmd.e
    }
    fn rollback(&self) -> Vec<&str> {
        self.commands
            .iter()
            .rev() // reverse iterator's dire
            .map(|cmd| cmd.rollback())
            .collect()
    }
}

fn main() {
    let mut schema = Schema::new();

    let cmd = Box::new(CreateTable);
    schema.add_migration(cmd);
    let cmd = Box::new(AddField);
```

```
    schema.add_migration(cmd);

    assert_eq!(vec!["create table", "add fiel
    assert_eq!(vec!["remove field", "drop tab
}
```

Approach: Using function pointe

We could follow another approach by creating each function and store function pointers to invoke these Since function pointers implement all three traits `F` well pass and store closures instead of function poi

```rust
type FnPtr = fn() -> String;
struct Command {
    execute: FnPtr,
    rollback: FnPtr,
}

struct Schema {
    commands: Vec<Command>,
}

impl Schema {
    fn new() -> Self {
        Self { commands: vec![] }
    }
    fn add_migration(&mut self, execute: FnPt
        self.commands.push(Command { execute,
    }
    fn execute(&self) -> Vec<String> {
        self.commands.iter().map(|cmd| (cmd.e
    }
    fn rollback(&self) -> Vec<String> {
        self.commands
            .iter()
            .rev()
            .map(|cmd| (cmd.rollback)())
            .collect()
    }
}

fn add_field() -> String {
    "add field".to_string()
}

fn remove_field() -> String {
    "remove field".to_string()
}

fn main() {
    let mut schema = Schema::new();
    schema.add_migration(|| "create table".to
table".to_string());
    schema.add_migration(add_field, remove_f
    assert_eq!(vec!["create table", "add fiel
    assert_eq!(vec!["remove field", "drop tab
}
```

Approach: Using `Fn` trait objects

Finally, instead of defining a common command tra
implementing the `Fn` trait separately in vectors.

```rust
type Migration<'a> = Box<dyn Fn() -> &'a str>

struct Schema<'a> {
    executes: Vec<Migration<'a>>,
    rollbacks: Vec<Migration<'a>>,
}

impl<'a> Schema<'a> {
    fn new() -> Self {
        Self {
            executes: vec![],
            rollbacks: vec![],
        }
    }
    fn add_migration<E, R>(&mut self, execute
    where
        E: Fn() -> &'a str + 'static,
        R: Fn() -> &'a str + 'static,
    {
        self.executes.push(Box::new(execute))
        self.rollbacks.push(Box::new(rollback
    }
    fn execute(&self) -> Vec<&str> {
        self.executes.iter().map(|cmd| cmd())
    }
    fn rollback(&self) -> Vec<&str> {
        self.rollbacks.iter().rev().map(|cmd|
    }
}

fn add_field() -> &'static str {
    "add field"
}

fn remove_field() -> &'static str {
    "remove field"
}

fn main() {
    let mut schema = Schema::new();
    schema.add_migration(|| "create table", |
    schema.add_migration(add_field, remove_f
    assert_eq!(vec!["create table", "add fiel
    assert_eq!(vec!["remove field", "drop tab
}
```

Discussion

If our commands are small and may be defined as f
using function pointers might be preferable since it
But if our command is a whole struct with a bunch c

seperated module then using trait objects would be
can be found in `actix`, which uses trait objects whe
routes. In case of using `Fn` trait objects we can crea
way as we used in case of function pointers.

As performance, there is always a trade-off betweer
and organisation. Static dispatch gives faster perfor
provides flexibility when we structure our applicatio

See also

- Command pattern
- Another example for the `command` pattern

Interpreter

Description

If a problem occurs very often and requires long an
problem instances might be expressed in a simple l
could solve it by interpreting the sentences written i

Basically, for any kind of problems we define:

- A domain specific language,
- A grammar for this language,
- An interpreter that solves the problem instanc

Motivation

Our goal is to translate simple mathematical expres
Reverse Polish notation) For simplicity, our expressi
two operations `+`, `-`. For example, the expression

Context Free Grammar for our p

Our task is translating infix expressions into postfix
grammar for a set of infix expressions over `0`, ..., `9`

- Terminal symbols: `0`, `...`, `9`, `+`, `-`
- Non-terminal symbols: `exp`, `term`
- Start symbol is `exp`
- And the following are production rules

```
exp -> exp + term
exp -> exp - term
exp -> term
term -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

NOTE: This grammar should be further transformed
do with it. For example, we might need to remove le
see Compilers: Principles,Techniques, and Tools (ak

Solution

We simply implement a recursive descent parser. Fo
when an expression is syntactically wrong (for exam
according to the grammar definition).

```
pub struct Interpreter<'a> {
    it: std::str::Chars<'a>,
}

impl<'a> Interpreter<'a> {

    pub fn new(infix: &'a str) -> Self {
        Self { it: infix.chars() }
    }

    fn next_char(&mut self) -> Option<char> {
        self.it.next()
    }

    pub fn interpret(&mut self, out: &mut Str
        self.term(out);

        while let Some(op) = self.next_char()
            if op == '+' || op == '-' {
                self.term(out);
                out.push(op);
            } else {
                panic!("Unexpected symbol '{}
            }
        }
    }

    fn term(&mut self, out: &mut String) {
        match self.next_char() {
            Some(ch) if ch.is_digit(10) => ou
            Some(ch) => panic!("Unexpected sy
            None => panic!("Unexpected end of
        }
    }
}

pub fn main() {
    let mut intr = Interpreter::new("2+3");
    let mut postfix = String::new();
    intr.interpret(&mut postfix);
    assert_eq!(postfix, "23+");

    intr = Interpreter::new("1-2+3-4");
    postfix.clear();
    intr.interpret(&mut postfix);
    assert_eq!(postfix, "12-3+4-");
}
```

Discussion

There may be a wrong perception that the Interpret grammars for formal languages and implementatio fact, this pattern is about expressing problem instar implementing functions/classes/structs that solve th language has `macro_rules!` that allow us to define expand this syntax into source code.

In the following example we create a simple `macro_` length of `n` dimensional vectors. Writing `norm!(x,1` more efficient than packing `x,1,2` into a `Vec` and c length.

```
macro_rules! norm {
    ($($element:expr),*) => {
        {
            let mut n = 0.0;
            $(
                n += ($element as f64)*($elem
            )*
            n.sqrt()
        }
    };
}

fn main() {
    let x = -3f64;
    let y = 4f64;

    assert_eq!(3f64, norm!(x));
    assert_eq!(5f64, norm!(x, y));
    assert_eq!(0f64, norm!(0, 0, 0));
    assert_eq!(1f64, norm!(0.5, -0.5, 0.5, -0
}
```

See also

- Interpreter pattern
- Context free grammar
- macro_rules!

Newtype

What if in some cases we want a type to behave sim
behaviour at compile time when using only type alia

For example, if we want to create a custom `Display`
security considerations (e.g. passwords).

For such cases we could use the `Newtype` pattern to
encapsulation.

Description

Use a tuple struct with a single field to make an opa
new type, rather than an alias to a type (`type` items

Example

```
use std::fmt::Display;

// Create Newtype Password to override the D
struct Password(String);

impl Display for Password {
    fn fmt(&self, f: &mut std::fmt::Formatter
        write!(f, "****************")
    }
}

fn main() {
    let unsecured_password: String = "ThisIsM
    let secured_password: Password = Passworc
    println!("unsecured_password: {unsecured_
    println!("secured_password: {secured_pass
}
```

```
unsecured_password: ThisIsMyPassword
secured_password: ****************
```

Motivation

The primary motivation for newtypes is abstraction. implementation details between types while precise newtype rather than exposing the implementation t change implementation backwards compatibly.

Newtypes can be used for distinguishing units, e.g., `Miles` and `Kilometres`.

Advantages

The wrapped and wrapper types are not type comp users of the newtype will never 'confuse' the wrappe

Newtypes are a zero-cost abstraction - there is no ru

The privacy system ensures that users cannot acces private, which it is by default).

Disadvantages

The downside of newtypes (especially compared wit special language support. This means there can be through' method for every method you want to exp for every trait you want to also be implemented for

Discussion

Newtypes are very common in Rust code. Abstractic common uses, but they can be used for other reaso

- restricting functionality (reduce the functions e
- making a type with copy semantics have move
- abstraction by providing a more concrete type

```
pub struct Foo(Bar<T1, T2>);
```

Here, `Bar` might be some public, generic type and Users of our module shouldn't know that we implen we're really hiding here is the types `T1` and `T2`, anc

See also

- Advanced Types in the book
- Newtypes in Haskell
- Type aliases
- derive_more, a crate for deriving many builtin
- The Newtype Pattern In Rust

RAII with guards

Description

RAII stands for “Resource Acquisition is Initialisation
essence of the pattern is that resource initialisation
and finalisation in the destructor. This pattern is ext
as a guard of some resource and relying on the type
always mediated by the guard object.

Example

Mutex guards are the classic example of this patterr
simplified version of the real implementation):

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

struct Foo {}

struct Mutex<T> {
    // We keep a reference to our data: T her
    //..
}

struct MutexGuard<'a, T: 'a> {
    data: &'a T,
    //..
}

// Locking the mutex is explicit.
impl<T> Mutex<T> {
    fn lock(&self) -> MutexGuard<T> {
        // Lock the underlying OS mutex.
        //..

        // MutexGuard keeps a reference to se
        MutexGuard {
            data: self,
            //..
        }
    }
}

// Destructor for unlocking the mutex.
impl<'a, T> Drop for MutexGuard<'a, T> {
    fn drop(&mut self) {
        // Unlock the underlying OS mutex.
        //..
    }
}

// Implementing Deref means we can treat Mute
impl<'a, T> Deref for MutexGuard<'a, T> {
    type Target = T;

    fn deref(&self) -> &T {
        self.data
    }
}

fn baz(x: Mutex<Foo>) {
    let xx = x.lock();
    xx.foo(); // foo is a method on Foo.
    // The borrow checker ensures we can't st
underlying
    // Foo which will outlive the guard xx.

    // x is unlocked when we exit this funct
executed.
}
```

Motivation

Where a resource must be finalised after use, RAII c
is an error to access that resource after finalisation,
prevent such errors.

Advantages

Prevents errors where a resource is not finalised an
finalisation.

Discussion

RAII is a useful pattern for ensuring resources are p
can make use of the borrow checker in Rust to stati
using resources after finalisation takes place.

The core aim of the borrow checker is to ensure tha
that data. The RAII guard pattern works because the
the underlying resource and only exposes such refe
cannot outlive the underlying resource and that refe
the guard cannot outlive the guard. To see how this
signature of `deref` without lifetime elision:

```
fn deref<'a>(&'a self) -> &'a T {
    //..
}
```

The returned reference to the resource has the sam
checker therefore ensures that the lifetime of the re
lifetime of `self`.

Note that implementing `Deref` is not a core part of
guard object more ergonomic. Implementing a `get`
well.

See also

Finalisation in destructors idiom

RAII is a common pattern in C++: cppreference.com,

Style guide entry (currently just a placeholder).

Strategy (aka Policy)

Description

The Strategy design pattern is a technique that enal
allows to decouple software modules through Depe

The basic idea behind the Strategy pattern is that, gi
problem, we define only the skeleton of the algorith
separate the specific algorithm's implementation int

In this way, a client using the algorithm may choose
general algorithm workflow remains the same. In ot
of the class does not depend on the specific implem
specific implementation must adhere to the abstrac
"Dependency Inversion".

Motivation

Imagine we are working on a project that generates
reports to be generated in different formats (strateg
formats. But things vary over time, and we don't kno
get in the future. For example, we may need to gene
format, or just modify one of the existing formats.

Example

In this example our invariants (or abstractions) are
and `Json` are our strategy structs. These strategies
trait.

```
use std::collections::HashMap;

type Data = HashMap<String, u32>;

trait Formatter {
    fn format(&self, data: &Data, buf: &mut S
}

struct Report;

impl Report {
    // Write should be used but we kept it as
    fn generate<T: Formatter>(g: T, s: &mut S
        // backend operations...
        let mut data = HashMap::new();
        data.insert("one".to_string(), 1);
        data.insert("two".to_string(), 2);
        // generate report
        g.format(&data, s);
    }
}

struct Text;
impl Formatter for Text {
    fn format(&self, data: &Data, buf: &mut S
        for (k, v) in data {
            let entry = format!("{} {}\n", k,
            buf.push_str(&entry);
        }
    }
}

struct Json;
impl Formatter for Json {
    fn format(&self, data: &Data, buf: &mut S
        buf.push('[');
        for (k, v) in data.into_iter() {
            let entry = format!(r#"{{"{}":"{}
            buf.push_str(&entry);
            buf.push(',');
        }
        if !data.is_empty() {
            buf.pop(); // remove extra , at t
        }
        buf.push(']');
    }
}

fn main() {
    let mut s = String::from("");
    Report::generate(Text, &mut s);
    assert!(s.contains("one 1"));
    assert!(s.contains("two 2"));

    s.clear(); // reuse the same buffer
    Report::generate(Json, &mut s);
```

```
    assert!(s.contains(r#"{"one":"1"}"#));
    assert!(s.contains(r#"{"two":"2"}"#));
}
```

Advantages

The main advantage is separation of concerns. For e
know anything about specific implementations of `J`
implementations does not care about how data is p
only thing they have to know is a specific trait to imp
concrete algorithm implementation processing the
`format(...)`.

Disadvantages

For each strategy there must be implemented at lea
modules increases with number of strategies. If the
then users have to know how strategies differ from

Discussion

In the previous example all strategies are implemen
different strategies includes:

- All in one file (as shown in this example, simila
- Separated as modules, E.g. `formatter::json`
- Use compiler feature flags, E.g. `json` feature,
- Separated as crates, E.g. `json` crate, `text` cra

Serde crate is a good example of the `Strategy` patt
customization of the serialization behavior by manu
`Deserialize` traits for our type. For example, we co
`serde_cbor` since they expose similar methods. Ha
`serde_transcode` much more useful and ergonomi

However, we don't need to use traits in order to des

The following toy example demonstrates the idea o
`closures`:

```
struct Adder;
impl Adder {
    pub fn add<F>(x: u8, y: u8, f: F) -> u8
    where
        F: Fn(u8, u8) -> u8,
    {
        f(x, y)
    }
}

fn main() {
    let arith_adder = |x, y| x + y;
    let bool_adder = |x, y| {
        if x == 1 || y == 1 {
            1
        } else {
            0
        }
    };
    let custom_adder = |x, y| 2 * x + y;

    assert_eq!(9, Adder::add(4, 5, arith_adde
    assert_eq!(0, Adder::add(0, 0, bool_adder
    assert_eq!(5, Adder::add(1, 3, custom_adc
}
```

In fact, Rust already uses this idea for `Options`'s `ma`

```
fn main() {
    let val = Some("Rust");

    let len_strategy = |s: &str| s.len();
    assert_eq!(4, val.map(len_strategy).unwra

    let first_byte_strategy = |s: &str| s.byt
    assert_eq!(82, val.map(first_byte_strateg
}
```

See also

- Strategy Pattern
- Dependency Injection
- Policy Based Design
- Implementing a TCP server for Space Applicati

Visitor

Description

A visitor encapsulates an algorithm that operates ov objects. It allows multiple different algorithms to be having to modify the data (or their primary behaviou

Furthermore, the visitor pattern allows separating th from the operations performed on each object.

Example

```
// The data we will visit
mod ast {
    pub enum Stmt {
        Expr(Expr),
        Let(Name, Expr),
    }

    pub struct Name {
        value: String,
    }

    pub enum Expr {
        IntLit(i64),
        Add(Box<Expr>, Box<Expr>),
        Sub(Box<Expr>, Box<Expr>),
    }
}

// The abstract visitor
mod visit {
    use ast::*;

    pub trait Visitor<T> {
        fn visit_name(&mut self, n: &Name) ->
        fn visit_stmt(&mut self, s: &Stmt) ->
        fn visit_expr(&mut self, e: &Expr) ->
    }
}

use visit::*;
use ast::*;

// An example concrete implementation - walks
code.
struct Interpreter;
impl Visitor<i64> for Interpreter {
    fn visit_name(&mut self, n: &Name) -> i64
    fn visit_stmt(&mut self, s: &Stmt) -> i64
        match *s {
            Stmt::Expr(ref e) => self.visit_e
            Stmt::Let(..) => unimplemented!()
        }
    }

    fn visit_expr(&mut self, e: &Expr) -> i64
        match *e {
            Expr::IntLit(n) => n,
            Expr::Add(ref lhs, ref rhs) => se
self.visit_expr(rhs),
            Expr::Sub(ref lhs, ref rhs) => se
self.visit_expr(rhs),
        }
    }
}
```

One could implement further visitors, for example a
modify the AST data.

Motivation

The visitor pattern is useful anywhere that you want
heterogeneous data. If data is homogeneous, you ca
visitor object (rather than a functional approach) all
communicate information between nodes.

Discussion

It is common for the `visit_*` methods to return vo
that case it is possible to factor out the traversal coc
(and also to provide noop default methods). In Rust,
provide `walk_*` functions for each datum. For exan

```
pub fn walk_expr(visitor: &mut Visitor, e: &E
    match *e {
        Expr::IntLit(_) => {},
        Expr::Add(ref lhs, ref rhs) => {
            visitor.visit_expr(lhs);
            visitor.visit_expr(rhs);
        }
        Expr::Sub(ref lhs, ref rhs) => {
            visitor.visit_expr(lhs);
            visitor.visit_expr(rhs);
        }
    }
}
```

In other languages (e.g., Java) it is common for data
performs the same duty.

See also

The visitor pattern is a common pattern in most OO

[Wikipedia article](https://en.wikipedia.org/wiki/Visitor_pattern)

The [fold](#) pattern is similar to visitor but produces a r structure.

Creational Patterns

From Wikipedia:

> Design patterns that deal with object creation me
> in a manner suitable to the situation. The basic fo
> in design problems or in added complexity to the
> solve this problem by somehow controlling this o

Builder

Description

Construct an object with calls to a builder helper.

Example

```
#[derive(Debug, PartialEq)]
pub struct Foo {
    // Lots of complicated fields.
    bar: String,
}

impl Foo {
    // This method will help users to discove
    pub fn builder() -> FooBuilder {
        FooBuilder::default()
    }
}

#[derive(Default)]
pub struct FooBuilder {
    // Probably lots of optional fields.
    bar: String,
}

impl FooBuilder {
    pub fn new(/* ... */) -> FooBuilder {
        // Set the minimally required fields
        FooBuilder {
            bar: String::from("X"),
        }
    }

    pub fn name(mut self, bar: String) -> Foo
        // Set the name on the builder itsel
value.
        self.bar = bar;
        self
    }

    // If we can get away with not consuming
    // advantage. It means we can use the Foo
constructing
    // many Foos.
    pub fn build(self) -> Foo {
        // Create a Foo from the FooBuilder,
FooBuilder
        // to Foo.
        Foo { bar: self.bar }
    }
}

#[test]
fn builder_test() {
    let foo = Foo {
        bar: String::from("Y"),
    };
    let foo_from_builder: Foo =
FooBuilder::new().name(String::from("Y")).bu
    assert_eq!(foo, foo_from_builder);
}
```

Motivation

Useful when you would otherwise require many cor
side effects.

Advantages

Separates methods for building from other method

Prevents proliferation of constructors.

Can be used for one-liner initialisation as well as mc

Disadvantages

More complex than creating a struct object directly,

Discussion

This pattern is seen more frequently in Rust (and fo
languages because Rust lacks overloading. Since yo
a given name, having multiple constructors is less n

This pattern is often used where the builder object i
being just a builder. For example, see `std::process`
process). In these cases, the `T` and `TBuilder` nami

The example takes and returns the builder by value
more efficient) to take and return the builder as a m
makes this work naturally. This approach has the ac

```
let mut fb = FooBuilder::new();
fb.a();
fb.b();
let f = fb.build();
```

as well as the `FooBuilder::new().a().b().build()`

See also

- Description in the style guide
- derive_builder, a crate for automatically implei the boilerplate.
- Constructor pattern for when construction is s
- Builder pattern (wikipedia)
- Construction of complex values

Fold

Description

Run an algorithm over each item in a collection of d
a whole new collection.

The etymology here is unclear to me. The terms 'fol
compiler, although it appears to me to be more like
See the discussion below for more details.

Example

```rust
// The data we will fold, a simple AST.
mod ast {
    pub enum Stmt {
        Expr(Box<Expr>),
        Let(Box<Name>, Box<Expr>),
    }

    pub struct Name {
        value: String,
    }

    pub enum Expr {
        IntLit(i64),
        Add(Box<Expr>, Box<Expr>),
        Sub(Box<Expr>, Box<Expr>),
    }
}

// The abstract folder
mod fold {
    use ast::*;

    pub trait Folder {
        // A leaf node just returns the node
this
        // to inner nodes too.
        fn fold_name(&mut self, n: Box<Name>)
        // Create a new inner node by folding
        fn fold_stmt(&mut self, s: Box<Stmt>)
            match *s {
                Stmt::Expr(e) => Box::new(Stm
                Stmt::Let(n, e) => Box::new(S
self.fold_expr(e))),
            }
        }
        fn fold_expr(&mut self, e: Box<Expr>)
    }
}

use fold::*;
use ast::*;

// An example concrete implementation - renam
struct Renamer;
impl Folder for Renamer {
    fn fold_name(&mut self, n: Box<Name>) ->
        Box::new(Name { value: "foo".to_owned
    }
    // Use the default methods for the other
}
```

The result of running the `Renamer` on an AST is a ne with every name changed to `foo`. A real life folder r between nodes in the struct itself.

A folder can also be defined to map one data structu
data structure. For example, we could fold an AST ir
level intermediate representation).

Motivation

It is common to want to map a data structure by per
node in the structure. For simple operations on sim
using `Iterator::map` . For more complex operation
affect the operation on later nodes, or where iterati
trivial, using the fold pattern is more appropriate.

Like the visitor pattern, the fold pattern allows us to
from the operations performed to each node.

Discussion

Mapping data structures in this fashion is common i
languages, it would be more common to mutate the
'functional' approach is common in Rust, mostly due
Using fresh data structures, rather than mutating ol
code easier in most circumstances.

The trade-off between efficiency and reusability can
are accepted by the `fold_*` methods.

In the above example we operate on `Box` pointers.
exclusively, the original copy of the data structure ca
if a node is not changed, reusing it is very efficient.

If we were to operate on borrowed references, the o
however, a node must be cloned even if unchanged

Using a reference counted pointer gives the best of
original data structure, and we don't need to clone u
less ergonomic to use and mean that the data struc

See also

Iterators have a `fold` method, however this folds a
than into a new data structure. An iterator's `map` is

In other languages, fold is usually used in the sense
pattern. Some functional languages have powerful c
maps over data structures.

The visitor pattern is closely related to fold. They sh
structure performing an operation on each node. H
new data structure nor consume the old one.

Structural Patterns

From Wikipedia:

> Design patterns that ease the design by identifyir
> relationships among entities.

Struct decomposition fo borrowing

Description

Sometimes a large struct will cause issues with the b be borrowed independently, sometimes the whole s preventing other uses. A solution might be to decon structs. Then compose these together into the origi borrowed separately and have more flexible behavi

This will often lead to a better design in other ways: reveals smaller units of functionality.

Example

Here is a contrived example of where the borrow ch struct:

```
struct Database {
    connection_string: String,
    timeout: u32,
    pool_size: u32,
}

fn print_database(database: &Database) {
    println!("Connection string: {}", databas
    println!("Timeout: {}", database.timeout)
    println!("Pool size: {}", database.pool_s
}

fn main() {
    let mut db = Database {
        connection_string: "initial string".t
        timeout: 30,
        pool_size: 100,
    };

    let connection_string = &mut db.connectio
    print_database(&db);  // Immutable borrow
    // *connection_string = "new string".to_s
used

}
```

We can apply this design pattern and refactor `Datal` solving the borrow checking issue:

```
// Database is now composed of three structs
PoolSize.
// Let's decompose it into smaller structs
#[derive(Debug, Clone)]
struct ConnectionString(String);

#[derive(Debug, Clone, Copy)]
struct Timeout(u32);

#[derive(Debug, Clone, Copy)]
struct PoolSize(u32);

// We then compose these smaller structs back
struct Database {
    connection_string: ConnectionString,
    timeout: Timeout,
    pool_size: PoolSize,
}

// print_database can then take ConnectionStr
instead
fn print_database(connection_str: Connection
                  timeout: Timeout,
                  pool_size: PoolSize) {
    println!("Connection string: {:?}", conne
    println!("Timeout: {:?}", timeout);
    println!("Pool size: {:?}", pool_size);
}

fn main() {
    // Initialize the Database with the three
    let mut db = Database {
        connection_string: ConnectionString("
        timeout: Timeout(30),
        pool_size: PoolSize(100),
    };

    let connection_string = &mut db.connectio
    print_database(connection_string.clone(),
    *connection_string = ConnectionString("ne
}
```

Motivation

This pattern is most useful, when you have a struct t you want to borrow independently. Thus having a m

Advantages

Decomposition of structs lets you work around limit
often produces a better design.

Disadvantages

It can lead to more verbose code. And sometimes, t
abstractions, and so we end up with a worse design
indicating that the program should be refactored in

Discussion

This pattern is not required in languages that don't l
sense is unique to Rust. However, making smaller u
cleaner code: a widely acknowledged principle of so
the language.

This pattern relies on Rust's borrow checker to be al
each other. In the example, the borrow checker kno
can be borrowed independently, it does not try to b
pattern useless.

Prefer small crates

Description

Prefer small crates that do one thing well.

Cargo and crates.io make it easy to add third-party l
or C++. Moreover, since packages on crates.io cann
publication, any build that works now should contin
take advantage of this tooling, and use smaller, mor

Advantages

- Small crates are easier to understand, and enc
- Crates allow for re-using code between project developed as part of the Servo browser engine outside the project.
- Since the compilation unit of Rust is the crate, can allow more of the code to be built in parall

Disadvantages

- This can lead to "dependency hell", when a pro versions of a crate at the same time. For exam 1.0 and 0.5. Since the `Url` from `url:1.0` and types, an HTTP client that uses `url:0.5` woulc scraper that uses `url:1.0`.
- Packages on crates.io are not curated. A crate documentation, or be outright malicious.
- Two small crates may be less optimized than c not perform link-time optimization (LTO) by de

Examples

The `url` crate provides tools for working with URLs

The `num_cpus` crate provides a function to query th

The `ref_slice` crate provides functions for convert

See also

- crates.io: The Rust community crate host

Contain unsafety in sma

Description

If you have `unsafe` code, create the smallest possib
needed invariants to build a minimal safe interface
larger module that contains only safe code and pres
that the outer module can contain unsafe functions
the unsafe code. Users may use this to gain speed b

Advantages

- This restricts the unsafe code that must be au
- Writing the outer module is much easier, since the inner module

Disadvantages

- Sometimes, it may be hard to find a suitable in
- The abstraction may introduce inefficiencies.

Examples

- The `toolshed` crate contains its unsafe operat interface to users.
- `std`'s `String` class is a wrapper over `Vec<u8>` contents must be valid UTF-8. The operations However, users have the option of using an `ur` which case the onus is on them to guarantee t

See also

- Ralf Jung's Blog about invariants in unsafe cod

FFI Patterns

Writing FFI code is an entire course in itself. Howeve
can act as pointers, and avoid traps for inexperience

This section contains design patterns that may be us

1. Object-Based API design that has good memor
   boundary of what is safe and what is unsafe

2. Type Consolidation into Wrappers - group mul
   opaque "object"

Object-Based APIs

Description

When designing APIs in Rust which are exposed to c
important design principles which are contrary to n

1. All Encapsulated types should be *owned* by Rus
2. All Transactional data types should be *owned* b
3. All library behavior should be functions acting
4. All library behavior should be encapsulated int *provenance/lifetime*.

Motivation

Rust has built-in FFI support to other languages. It d
authors to provide C-compatible APIs through differ
to this practice).

Well-designed Rust FFI follows C API design principl
Rust as little as possible. There are three goals with

1. Make it easy to use in the target language.
2. Avoid the API dictating internal unsafety on th
3. Keep the potential for memory unsafety and R possible.

Rust code must trust the memory safety of the forei
However, every bit of `unsafe` code on the Rust side
exacerbate `undefined behaviour`.

For example, if a pointer provenance is wrong, that
memory access. But if it is manipulated by unsafe c
corruption.

The Object-Based API design allows for writing shim
characteristics, and a clean boundary of what is saf

Code Example

The POSIX standard defines the API to access an on-
excellent example of an “object-based” API.

Here is the definition in C, which hopefully should b
FFI. The commentary below should help explain it fo

```
struct DBM;
typedef struct { void *dptr, size_t dsize } d

int     dbm_clearerr(DBM *);
void    dbm_close(DBM *);
int     dbm_delete(DBM *, datum);
int     dbm_error(DBM *);
datum   dbm_fetch(DBM *, datum);
datum   dbm_firstkey(DBM *);
datum   dbm_nextkey(DBM *);
DBM    *dbm_open(const char *, int, mode_t);
int     dbm_store(DBM *, datum, datum, int);
```

This API defines two types: `DBM` and `datum`.

The `DBM` type was called an “encapsulated” type abo
state, and acts as an entry point for the library’s beh

It is completely opaque to the user, who cannot crea
know its size or layout. Instead, they must call `dbm_o`
pointer to one.

This means all `DBM`s are “owned” by the library in a
unknown size is kept in memory controlled by the li
manage its life cycle with `open` and `close`, and per
functions.

The `datum` type was called a “transactional” type ab
exchange of information between the library and its

The database is designed to store “unstructured dat
meaning. As a result, the `datum` is the C equivalent
count of how many there are. The main difference is
which is what `void` indicates.

Keep in mind that this header is written from the lib
has some type they are using, which has a known si
by the rules of C casting, any type behind a pointer c

As noted earlier, this type is *transparent* to the user.
user. This has subtle ramifications, due to that point
owns the memory that pointer points to?

The answer for best memory safety is, "the user". Bu
the user does not know how to allocate it correctly (
value is). In this case, the library code is expected to
to – such as the C library `malloc` and `free` – and th
sense.

This may all seem speculative, but this is what a poi
thing as Rust: "user defined lifetime." The user of th
documentation in order to use it correctly. That saic
fewer or greater consequences if users do it wrong.
practice is about, and the key is to *transfer ownershi*

Advantages

This minimizes the number of memory safety guara
relatively small number:

1. Do not call any function with a pointer not retu
   corruption).
2. Do not call any function on a pointer after clos
3. The `dptr` on any `datum` must be `NULL`, or poi
   advertised length.

In addition, it avoids a lot of pointer provenance issu
consider an alternative in some depth: key iteration

Rust is well known for its iterators. When implemen
separate type with a bounded lifetime to its owner,

Here is how iteration would be done in Rust for `DBM`

```
struct Dbm { ... }

impl Dbm {
    /* ... */
    pub fn keys<'it>(&'it self) -> DbmKeysIte
    /* ... */
}

struct DbmKeysIter<'it> {
    owner: &'it Dbm,
}

impl<'it> Iterator for DbmKeysIter<'it> { ...
```

This is clean, idiomatic, and safe. thanks to Rust's gu straightforward API translation would look like:

```
#[no_mangle]
pub extern "C" fn dbm_iter_new(owner: *const
    // THIS API IS A BAD IDEA! For real appl
instead.
}
#[no_mangle]
pub extern "C" fn dbm_iter_next(
    iter: *mut DbmKeysIter,
    key_out: *const datum
) -> libc::c_int {
    // THIS API IS A BAD IDEA! For real appl
instead.
}
#[no_mangle]
pub extern "C" fn dbm_iter_del(*mut DbmKeysIt
    // THIS API IS A BAD IDEA! For real appl
instead.
}
```

This API loses a key piece of information: the lifetim
lifetime of the `Dbm` object that owns it. A user of the
causes the iterator to outlive the data it is iterating
memory.

This example written in C contains a bug that will be

```
int count_key_sizes(DBM *db) {
    // DO NOT USE THIS FUNCTION. IT HAS A SUB
    datum key;
    int len = 0;

    if (!dbm_iter_new(db)) {
        dbm_close(db);
        return -1;
    }

    int l;
    while ((l = dbm_iter_next(owner, &key)) >
by -1
        free(key.dptr);
        len += key.dsize;
        if (l == 0) { // end of the iterator
            dbm_close(owner);
        }
    }
    if l >= 0 {
        return -1;
    } else {
        return len;
    }
}
```

This bug is a classic. Here's what happens when the
marker:

1. The loop condition sets `l` to zero, and enters
2. The length is incremented, in this case by zero
3. The if statement is true, so the database is clos
   statement here.
4. The loop condition executes again, causing a

The worst part about this bug? If the Rust implemen
most of the time! If the memory for the `Dbm` object
check will almost certainly fail, resulting in the iterat
But occasionally, it will cause a segmentation fault, c
corruption!

None of this can be avoided by Rust. From its persp
returned pointers to them, and gave up control of tl
"play nice".

The programmer must read and understand the AP
consider that par for the course in C, a good API des
API for `DBM` did this by *consolidating the ownership* o

```
datum   dbm_firstkey(DBM *);
datum   dbm_nextkey(DBM *);
```

Thus, all the lifetimes were bound together, and suc

Disadvantages

However, this design choice also has a number of dı
considered as well.

First, the API itself becomes less expressive. With PC
per object, and every call changes its state. This is m
almost any language, even though it is safe. Perhaps
lifetimes are less hierarchical, this limitation is more

Second, depending on the relationships of the API's
involved. Many of the easier design points have oth

- Wrapper Type Consolidation groups multiple F
  "object"

- FFI Error Passing explains error handling with i
values (such as `NULL` pointers)
- Accepting Foreign Strings allows accepting stri
easier to get right than Passing Strings to FFI

However, not every API can be done this way. It is u
programmer as to who their audience is.

Type Consolidation into

Description

This pattern is designed to allow gracefully handling
minimizing the surface area for memory unsafety.

One of the cornerstones of Rust's aliasing rules is lif
patterns of access between types can be memory s

However, when Rust types are exported to other lar
into pointers. In Rust, a pointer means "the user ma
their responsibility to avoid memory unsafety.

Some level of trust in the user code is thus required
Rust can do nothing about. However, some API desi
on the code written in the other language.

The lowest risk API is the "consolidated wrapper", w
object are folded into a "wrapper type", while keepi

Code Example

To understand this, let us look at a classic example
collection.

That API looks like this:

1. The iterator is initialized with `first_key`.
2. Each call to `next_key` will advance the iterato
3. Calls to `next_key` if the iterator is at the end w
4. As noted above, the iterator is "wrapped into"
   API).

If the iterator implements `nth()` efficiently, then it
each function call:

```
struct MySetWrapper {
    myset: MySet,
    iter_next: usize,
}

impl MySetWrapper {
    pub fn first_key(&mut self) -> Option<&Ke
        self.iter_next = 0;
        self.next_key()
    }
    pub fn next_key(&mut self) -> Option<&Key
        if let Some(next) = self.myset.keys()
            self.iter_next += 1;
            Some(next)
        } else {
            None
        }
    }
}
```

As a result, the wrapper is simple and contains no u

Advantages

This makes APIs safer to use, avoiding issues with lif Based APIs for more on the advantages and pitfalls

Disadvantages

Often, wrapping types is quite difficult, and sometin make things easier.

As an example, consider an iterator which does not definitely be worth putting in special logic to make t or to support a different access pattern efficiently tl use.

Trying to Wrap Iterators (and Failing)

To wrap any type of iterator into the API correctly, tl C version of the code would do: erase the lifetime o

Suffice it to say, this is *incredibly* difficult.

Here is an illustration of just *one* pitfall.

A first version of `MySetWrapper` would look like this:

```
struct MySetWrapper {
    myset: MySet,
    iter_next: usize,
    // created from a transmuted Box<KeysIter
    iterator: Option<NonNull<KeysIter<'static
}
```

With `transmute` being used to extend a lifetime, an
But it gets even worse: *any other operation can cause*

Consider that the `MySet` in the wrapper could be m
iteration, such as storing a new value to the key it w
discourage this, and in fact some similar C libraries

A simple implementation of `myset_store` would be

```
pub mod unsafe_module {

    // other module content

    pub fn myset_store(
        myset: *mut MySetWrapper,
        key: datum,
        value: datum) -> libc::c_int {

        // DO NOT USE THIS CODE. IT IS UNSAFE

        let myset: &mut MySet = unsafe { // S
here!
            &mut (*myset).myset
        };

        /* ...check and cast key and value da

        match myset.store(casted_key, casted_
            Ok(_) => 0,
            Err(e) => e.into()
        }
    }
}
```

If the iterator exists when this function is called, we
rules. According to Rust, the mutable reference in th
the object. If the iterator simply exists, it's not exclus
`behaviour`! [1]

To avoid this, we must have a way of ensuring that r

That basically means clearing out the iterator's shar
reconstructing it. In most cases, that will still be less

Some may ask: how can C do this more efficiently? T
rules are the problem, and C simply ignores them fo
common to see code that is declared in the manual
circumstances. In fact, the GNU C library has an enti
behavior!

Rust would rather make everything memory safe all
optimizations that C code cannot attain. Being denie
price Rust programmers need to pay.

[1] For the C programmers out there scratching their heads
code cause the UB. The exclusivity rule also enables comp
inconsistent observations by the iterator's shared referen
instructions for efficiency). These observations may happe
created.

Anti-patterns

An anti-pattern is a solution to a "recurring problem being highly counterproductive". Just as valuable as knowing how *not* to solve it. Anti-patterns give us gr relative to design patterns. Anti-patterns are not coı can be an anti-pattern, too.

Clone to satisfy the borr

Description

The borrow checker prevents Rust users from devel ensuring that either: only one mutable reference ex immutable references exist. If the code written does this anti-pattern arises when the developer resolves variable.

Example

```
// define any variable
let mut x = 5;

// Borrow `x` -- but clone it first
let y = &mut (x.clone());

// without the x.clone() two lines prior, th
// x has been borrowed
// thanks to x.clone(), x was never borrowed
println!("{}", x);

// perform some action on the borrow to preve
//out of existence
*y += 1;
```

Motivation

It is tempting, particularly for beginners, to use this with the borrow checker. However, there are seriou causes a copy of the data to be made. Any changes synchronized – as if two completely separate variab

There are special cases – `Rc<T>` is designed to hanc manages exactly one copy of the data, and cloning i

There is also `Arc<T>` which provides shared owners allocated in the heap. Invoking `.clone()` on `Arc` p points to the same allocation on the heap as the sou count.

In general, clones should be deliberate, with full unc
clone is used to make a borrow checker error disap|
pattern may be in use.

Even though `.clone()` is an indication of a bad pat
inefficient code, in cases such as when:

- the developer is still new to ownership
- the code doesn't have great speed or memory
  or prototypes)
- satisfying the borrow checker is really complic
  readability over performance

If an unnecessary clone is suspected, The Rust Book
understood fully before assessing whether the clon

Also be sure to always run `cargo clippy` in your pr
which `.clone()` is not necessary, like 1, 2, 3 or 4.

See also

- `mem::{take(_), replace(_)}` to keep owned
- `Rc<T>` documentation, which handles .clone()
- `Arc<T>` documentation, a thread-safe referen
- Tricks with ownership in Rust

#![deny(warnings)]

Description

A well-intentioned crate author wants to ensure the
they annotate their crate root with the following:

Example

```
#![deny(warnings)]

// All is well.
```

Advantages

It is short and will stop the build if anything is amiss

Drawbacks

By disallowing the compiler to build with warnings, a
famed stability. Sometimes new features or old mist
are done, thus lints are written that `warn` for a certa
to `deny`.

For example, it was discovered that a type could hav
This was deemed a bad idea, but in order to make tl
`overlapping-inherent-impls` lint was introduced t
on this fact, before it becomes a hard error in a futu

Also sometimes APIs get deprecated, so their use wi
was none.

All this conspires to potentially break the build wher

Furthermore, crates that supply additional lints (e.g.
unless the annotation is removed. This is mitigated

`lints=warn` command line argument, turns all `deny

Alternatives

There are two ways of tackling this problem: First, w
the code, and second, we can name the lints we wa

The following command line will build with all warni

```
RUSTFLAGS="-D warnings" cargo build
```

This can be done by any individual developer (or be
remember that this may break the build when some
change to the code.

Alternatively, we can specify the lints that we want t
warning lints that is (hopefully) safe to deny (as of R

```
#![deny(bad_style,
       const_err,
       dead_code,
       improper_ctypes,
       non_shorthand_field_patterns,
       no_mangle_generic_items,
       overflowing_literals,
       path_statements,
       patterns_in_fns_without_body,
       private_in_public,
       unconditional_recursion,
       unused,
       unused_allocation,
       unused_comparisons,
       unused_parens,
       while_true)]
```

In addition, the following `allow` ed lints may be a go

```
#![deny(missing_debug_implementations,
       missing_docs,
       trivial_casts,
       trivial_numeric_casts,
       unused_extern_crates,
       unused_import_braces,
       unused_qualifications,
       unused_results)]
```

Some may also want to add `missing-copy-impleme

Note that we explicitly did not add the `deprecated`
be more deprecated APIs in the future.

See also

- A collection of all clippy lints
- deprecate attribute documentation
- Type `rustc -W help` for a list of lints on your
  general list of options
- rust-clippy is a collection of lints for better Rus

`Deref` polymorphism

Description

Misuse the `Deref` trait to emulate inheritance betw

Example

Sometimes we want to emulate the following comm
as Java:

```
class Foo {
    void m() { ... }
}

class Bar extends Foo {}

public static void main(String[] args) {
    Bar b = new Bar();
    b.m();
}
```

We can use the deref polymorphism anti-pattern to

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

struct Foo {}

impl Foo {
    fn m(&self) {
        //..
    }
}

struct Bar {
    f: Foo,
}

impl Deref for Bar {
    type Target = Foo;
    fn deref(&self) -> &Foo {
        &self.f
    }
}

fn main() {
    let b = Bar { f: Foo {} };
    b.m();
}
```

There is no struct inheritance in Rust. Instead we us
instance of `Foo` in `Bar` (since the field is a value, it i
they would have the same layout in memory as the
use `#[repr(C)]` if you want to be sure)).

In order to make the method call work we implemen
target (returning the embedded `Foo` field). That me
(for example, using `*`) then we will get a `Foo`. That
gives a `T` from a reference to `T`, here we have two
dot operator does implicit dereferencing, it means t
methods on `Foo` as well as `Bar`.

Advantages

You save a little boilerplate, e.g.,

```rust
impl Bar {
    fn m(&self) {
        self.f.m()
    }
}
```

Disadvantages

Most importantly this is a surprising idiom - future p
not expect this to happen. That's because we are mi
using it as intended (and documented, etc.). It's also
completely implicit.

This pattern does not introduce subtyping between
or C++ does. Furthermore, traits implemented by `F`
implemented for `Bar`, so this pattern interacts badl
generic programming.

Using this pattern gives subtly different semantics fr
to `self`. Usually it remains a reference to the sub-c
'class' where the method is defined.

Finally, this pattern only supports single inheritance
class-based privacy, or other inheritance-related fea
will be subtly surprising to programmers used to Jav

Discussion

There is no one good alternative. Depending on the
better to re-implement using traits or to write out th
manually. We do intend to add a mechanism for inh
is likely to be some time before it reaches stable Rus
issue for more details.

The `Deref` trait is designed for the implementation
intention is that it will take a pointer-to-`T` to a `T`, n
is a shame that this isn't (probably cannot be) enforc

Rust tries to strike a careful balance between explici
explicit conversions between types. Automatic deref
where the ergonomics strongly favour an implicit m
this is limited to degrees of indirection, not conversi

See also

- Collections are smart pointers idiom.
- Delegation crates for less boilerplate like deleg

- Documentation for `Deref` trait.

Functional Usage of Rus

Rust is an imperative language, but it follows many

> In computer science, *functional programming* is a
> programs are constructed by applying and comp
> programming paradigm in which function definiti
> each return a value, rather than a sequence of im
> the state of the program.

Programming paradigms

One of the biggest hurdles to understanding functic imperative background is the shift in thinking. Impe something, whereas declarative programs describe from 1 to 10 to show this.

Imperative

```
let mut sum = 0;
for i in 1..11 {
    sum += i;
}
println!("{}", sum);
```

With imperative programs, we have to play compile start with a `sum` of `0`. Next, we iterate through the through the loop, we add the corresponding value i

i	sum
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

This is how most of us start out programming. We le

Declarative

```
println!("{}", (1..11).fold(0, |a, b| a + b));
```

Whoa! This is really different! What's going on here? programs we are describing **what** to do, rather thar composes functions. The name is a convention from

Here, we are composing functions of addition (this c from 1 to 10. The `0` is the starting point, so `a` is `0` range, `1`. `0 + 1 = 1` is the result. So now we `fold` `+ 2 = 3` is the next result. This process continues u range, `10`.

a	b	re
0	1	
1	2	
3	3	
6	4	
10	5	
15	6	
21	7	
28	8	
36	9	
45	10	

Generics as Type Classes

Description

Rust's type system is designed more like functional l imperative languages (like Java and C++). As a result programming problems into "static typing" problem choosing a functional language, and is critical to ma

A key part of this idea is the way generic types work types are a meta-programming construct for the co `vector<char>` in C++ are just two different copies o `vector` type (known as a `template`) with two differ

In Rust, a generic type parameter creates what is kn "type class constraint", and each different paramete *changes the type*. In other words, `Vec<isize>` and `V` which are recognized as distinct by all parts of the ty

This is called **monomorphization**, where different t code. This special behavior requires `impl` blocks to values for the generic type cause different types, an `impl` blocks.

In object-oriented languages, classes can inherit beh this allows the attachment of not only additional be type class, but extra behavior as well.

The nearest equivalent is the runtime polymorphism members can be added to objects willy-nilly by any languages, all of Rust's additional methods can be ty because their generics are statically defined. That m remaining safe.

Example

Suppose you are designing a storage server for a se software involved, there are two different protocols network boot), and NFS (for remote mount storage)

Your goal is to have one program, written in Rust, w

have protocol handlers and listen for both kinds of r
will then allow a lab administrator to configure stora
actual files.

The requests from machines in the lab for files cont
matter what protocol they came from: an authentica
retrieve. A straightforward implementation would lo

```
enum AuthInfo {
    Nfs(crate::nfs::AuthInfo),
    Bootp(crate::bootp::AuthInfo),
}

struct FileDownloadRequest {
    file_name: PathBuf,
    authentication: AuthInfo,
}
```

This design might work well enough. But now suppo
metadata that was *protocol specific*. For example, wi
what their mount point was in order to enforce add

The way the current struct is designed leaves the pr
means any method that applies to one protocol anc
programmer to do a runtime check.

Here is how getting an NFS mount point would look

```
struct FileDownloadRequest {
    file_name: PathBuf,
    authentication: AuthInfo,
    mount_point: Option<PathBuf>,
}

impl FileDownloadRequest {
    // ... other methods ...

    /// Gets an NFS mount point if this is ar
    /// return None.
    pub fn mount_point(&self) -> Option<&Path
        self.mount_point.as_ref()
    }
}
```

Every caller of `mount_point()` must check for `None`
true even if they know only NFS requests are ever u

It would be far more optimal to cause a compile-tim
were confused. After all, the entire path of the user's
the library they use, will know whether a request is a

In Rust, this is actually possible! The solution is to *ac*
API.

Here is what that looks like:

```rust
use std::path::{Path, PathBuf};

mod nfs {
    #[derive(Clone)]
    pub(crate) struct AuthInfo(String); // NF
}

mod bootp {
    pub(crate) struct AuthInfo(); // no authe
}

// private module, lest outside users invent
mod proto_trait {
    use std::path::{Path, PathBuf};
    use super::{bootp, nfs};

    pub(crate) trait ProtoKind {
        type AuthInfo;
        fn auth_info(&self) -> Self::AuthInfo
    }

    pub struct Nfs {
        auth: nfs::AuthInfo,
        mount_point: PathBuf,
    }

    impl Nfs {
        pub(crate) fn mount_point(&self) -> &
            &self.mount_point
        }
    }

    impl ProtoKind for Nfs {
        type AuthInfo = nfs::AuthInfo;
        fn auth_info(&self) -> Self::AuthInfo
            self.auth.clone()
        }
    }

    pub struct Bootp(); // no additional meta

    impl ProtoKind for Bootp {
        type AuthInfo = bootp::AuthInfo;
        fn auth_info(&self) -> Self::AuthInfo
            bootp::AuthInfo()
        }
    }
}

use proto_trait::ProtoKind; // keep internal
pub use proto_trait::{Nfs, Bootp}; // re-expo

struct FileDownloadRequest<P: ProtoKind> {
    file_name: PathBuf,
    protocol: P,
}
```

```
// all common API parts go into a generic imp
impl<P: ProtoKind> FileDownloadRequest<P> {
    fn file_path(&self) -> &Path {
        &self.file_name
    }

    fn auth_info(&self) -> P::AuthInfo {
        self.protocol.auth_info()
    }
}

// all protocol-specific impls go into their
impl FileDownloadRequest<Nfs> {
    fn mount_point(&self) -> &Path {
        self.protocol.mount_point()
    }
}

fn main() {
    // your code here
}
```

With this approach, if the user were to make a mista

```
fn main() {
    let mut socket = crate::bootp::listen()?;
    while let Some(request) = socket.next_rec
        match request.mount_point().as_ref()
            "/secure" => socket.send("Access
            _ => {} // continue on...
        }
        // Rest of the code here
    }
}
```

They would get a syntax error. The type `FileDownlo` implement `mount_point()`, only the type `FileDown` created by the NFS module, not the BOOTP module

Advantages

First, it allows fields that are common to multiple sta the non-shared fields generic, they are implemented

Second, it makes the `impl` blocks easier to read, be Methods common to all states are typed once in on state are in a separate block.

Both of these mean there are fewer lines of code, ar

Disadvantages

This currently increases the size of the binary, due t
implemented in the compiler. Hopefully the implem
future.

Alternatives

- If a type seems to need a "split API" due to con
consider the Builder Pattern instead.
- If the API between types does not change – on
Strategy Pattern is better used instead.

See also

This pattern is used throughout the standard library

- `Vec<u8>` can be cast from a String, unlike ever
- They can also be cast into a binary heap, but o
implements the `Ord` trait.[2]
- The `to_string` method was specialized for `Co`

It is also used by several popular crates to allow API

- The `embedded-hal` ecosystem used for embec
this pattern. For example, it allows statically ve
registers used to control embedded pins. Whe
`Pin<MODE>` struct, whose generic determines t
which are not on the `Pin` itself. [4]
- The `hyper` HTTP client library uses this to exp
requests. Clients with different connectors hav
as different trait implementations, while a core
connector. [5]
- The "type state" pattern – where an object gair

state or invariant – is implemented in Rust usir
slightly different technique. [6]

[1] See: impl From<CString> for Vec<u8>

[2] See: impl<T: Ord> FromIterator<T> for BinaryHeap<T>

[3] See: impl<'_> ToString for Cow<'_, str>

[4] Example: https://docs.rs/stm32f30x-hal/0.1.0/stm32f30x

[5] See: https://docs.rs/hyper/0.14.5/hyper/client/struct.Clie

[6] See: The Case for the Type State Pattern and Rusty Type

Functional Language Op

Optics is a type of API design that is common to fun
functional concept that is not frequently used in Rus

Nevertheless, exploring the concept may be helpful
APIs, such as visitors. They also have niche use case

This is quite a large topic, and would require actual
into its abilities. However their applicability in Rust i

To explain the relevant parts of the concept, the `Se`
as it is one that is difficult for many to to understand

In the process, different specific patterns, called Opt
The Poly Iso, and *The Prism*.

An API Example: Serde

Trying to understand the way *Serde* works by only re
especially the first time. Consider the `Deserializer`
which parses a new data format:

```
pub trait Deserializer<'de>: Sized {
    type Error: Error;

    fn deserialize_any<V>(self, visitor: V) -
    where
        V: Visitor<'de>;

    fn deserialize_bool<V>(self, visitor: V)
    where
        V: Visitor<'de>;

    // remainder omitted
}
```

And here's the definition of the `Visitor` trait passe

```
pub trait Visitor<'de>: Sized {
    type Value;

    fn visit_bool<E>(self, v: bool) -> Result
    where
        E: Error;

    fn visit_u64<E>(self, v: u64) -> Result<S
    where
        E: Error;

    fn visit_str<E>(self, v: &str) -> Result<
    where
        E: Error;

    // remainder omitted
}
```

There is a lot of type erasure going on here, with mu
passed back and forth.

But what is the big picture? Why not just have the v
needs in a streaming API, and call it a day? Why all tl

One way to understand it is to look at a functional la

This is a way to do composition of behavior and pro
patterns common to Rust: failure, type transformati

The Rust language does not have very good support
appear in the design of the language itself, and their
some of Rust's APIs. As a result, this attempts to exp
does it.

This will perhaps shed light on what those APIs are a
composability.

Basic Optics

The Iso

The Iso is a value transformer between two types. It
conceptually important building block.

As an example, suppose that we have a custom Has

concordance for a document.[2] It uses strings for key
values (file offsets, for instance).

A key feature is the ability to serialize this format to
would be to implement a conversion to and from a s
ignored for the time being, they will be handled late

To write it in a normal form expected by functional l

```
case class ConcordanceSerDe {
  serialize: Concordance -> String
  deserialize: String -> Concordance
}
```

The Iso is thus a pair of functions which convert valu
`deserialize`.

A straightforward implementation:

```
use std::collections::HashMap;

struct Concordance {
    keys: HashMap<String, usize>,
    value_table: Vec<(usize, usize)>,
}

struct ConcordanceSerde {}

impl ConcordanceSerde {
    fn serialize(value: Concordance) -> Strin
        todo!()
    }
    // invalid concordances are empty
    fn deserialize(value: String) -> Concorda
        todo!()
    }
}
```

This may seem rather silly. In Rust, this type of beha
all, the standard library has `FromStr` and `ToString`

But that is where our next subject comes in: Poly Iso

Poly Isos

The previous example was simply converting betwe
block builds upon it with generics, and is more inter

Poly Isos allow an operation to be generic over any t

This brings us closer to parsing. Consider what a ba: cases. Again, this is its normal form:

```
case class Serde[T] {
    deserialize(String) -> T
    serialize(T) -> String
}
```

Here we have our first generic, the type `T` being co

In Rust, this could be implemented with a pair of tra and `ToString`. The Rust version even handles error

```
pub trait FromStr: Sized {
    type Err;

    fn from_str(s: &str) -> Result<Self, Sel
}

pub trait ToString {
    fn to_string(&self) -> String;
}
```

Unlike the Iso, the Poly Iso allows application of mul generically. This is what you would want for a basic

At first glance, this seems like a good option for writ

```
use anyhow;

use std::str::FromStr;

struct TestStruct {
  a: usize,
  b: String,
}

impl FromStr for TestStruct {
    type Err = anyhow::Error;
    fn from_str(s: &str) -> Result<TestStruct
        todo!()
    }
}

impl ToString for TestStruct {
    fn to_string(&self) -> String {
        todo!()
    }
}

fn main() {
    let a = TestStruct { a: 5, b: "hello".to_
    println!("Our Test Struct as JSON: {}", a
}
```

That seems quite logical. However, there are two pr

First, `to_string` does not indicate to API users, "thi
agree on a JSON representation, and many of the ty
already don't. Using this is a poor fit. This can easily

But there is a second, subtler problem: scaling.

When every type writes `to_string` by hand, this wo
wants their type to be serializable has to write a bur
JSON libraries – to do it themselves, it will turn into a

The answer is one of Serde's two key innovations: an
represent Rust data in structures common to data s
that it can use Rust's code generation abilities to cre
it calls a `Visitor`.

This means, in normal form (again, skipping error ha

```
case class Serde[T] {
    deserialize: Visitor[T] -> T
    serialize: T -> Visitor[T]
}

case class Visitor[T] {
    toJson: Visitor[T] -> String
    fromJson: String -> Visitor[T]
}
```

The result is one Poly Iso and one Iso (respectively).
with traits:

```
trait Serde {
    type V;
    fn deserialize(visitor: Self::V) -> Self;
    fn serialize(self) -> Self::V;
}

trait Visitor {
    fn to_json(self) -> String;
    fn from_json(json: String) -> Self;
}
```

Because there is a uniform set of rules to transform
form, it is even possible to have code generation cre
type `T`:

```
#[derive(Default, Serde)] // the "Serde" der
struct TestStruct {
  a: usize,
  b: String,
}

// user writes this macro to generate an asso
generate_visitor!(TestStruct);
```

Or do they?

```
fn main() {
    let a = TestStruct { a: 5, b: "hello".to_
    let a_data = a.serialize().to_json();
    println!("Our Test Struct as JSON: {}", a
    let b = TestStruct::deserialize(
        generated_visitor_for!(TestStruct)::f
}
```

It turns out that the conversion isn't symmetric after
generated code the name of the actual type necessa
`String` is hidden. We'd need some kind of `generat

type name.

It's wonky, but it works… until we get to the elephan

The only format currently supported is JSON. How w

The current design requires completely re-writing al
a new Serde trait. That is quite terrible and not exte

In order to solve that, we need something more pov

Prism

To take format into account, we need something in

```
case class Serde[T, F] {
    serialize: T, F -> String
    deserialize: String, F -> Result[T, Error
}
```

This construct is called a Prism. It is "one level highe
case, the "intersecting" type F is the key).

Unfortunately because `Visitor` is a trait (since eacl
code), this would require a kind of generic type bou

Fortunately, we still have that `Visitor` type from be
attempting to allow each data structure to define th

Well what if we could add one more interface for th
is just an implementation detail, and it would "bridg

In normal form:

```
case class Serde[T] {
    serialize: F -> String
    deserialize F, String -> Result[T, Error]
}

case class VisitorForT {
    build: F, String -> Result[T, Error]
    decompose: F, T -> String
}

case class SerdeFormat[T, V] {
    toString: T, V -> String
    fromString: V, String -> Result[T, Error]
}
```

And what do you know, a pair of Poly Isos at the bot
traits!

Thus we have the Serde API:

1. Each type to be serialized implements `Deseri`
   the `Serde` class
2. They get a type (well two, one for each directio
   which is usually (but not always) done through
   This contains the logic to construct or destruct
   format of the Serde data model.
3. The type implementing the `Deserializer` trai
   format, being "driven by" the `Visitor`.

This splitting and Rust type erasure is really to achie

You can see it on the `Deserializer` trait

```
pub trait Deserializer<'de>: Sized {
    type Error: Error;

    fn deserialize_any<V>(self, visitor: V) -
    where
        V: Visitor<'de>;

    fn deserialize_bool<V>(self, visitor: V)
    where
        V: Visitor<'de>;

    // remainder omitted
}
```

And the visitor:

```
pub trait Visitor<'de>: Sized {
    type Value;

    fn visit_bool<E>(self, v: bool) -> Result
    where
        E: Error;

    fn visit_u64<E>(self, v: u64) -> Result<S
    where
        E: Error;

    fn visit_str<E>(self, v: &str) -> Result<
    where
        E: Error;

    // remainder omitted
}
```

And the trait `Deserialize` implemented by the mac

```
pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Res
    where
        D: Deserializer<'de>;
}
```

This has been abstract, so let's look at a concrete ex

How does actual Serde deserialize a bit of JSON into

1. The user would call a library function to deseri `Deserializer` based on the JSON format.
2. Based on the fields in the struct, a `Visitor` wo moment) which knows how to create each typ needed to represent it: `Vec` (list), `u64` and `St`
3. The deserializer would make calls to the `Visit`
4. The `Visitor` would indicate if the items found error to indicate deserialization has failed.

For our very simple structure above, the expected p

1. Begin visiting a map (*Serde*'s equivalent to `Has`
2. Visit a string key called "keys".
3. Begin visiting a map value.
4. For each item, visit a string key then an integer
5. Visit the end of the map.
6. Store the map into the `keys` field of the data s
7. Visit a string key called "value_table".
8. Begin visiting a list value.

9. For each item, visit an integer.
10. Visit the end of the list
11. Store the list into the `value_table` field.
12. Visit the end of the map.

But what determines which "observation" pattern is

A functional programming language would be able t
each type based on the type itself. Rust does not su
need to have its own code written based on its field

Serde solves this usability challenge with a derive ma

```
use serde::Deserialize;

#[derive(Deserialize)]
struct IdRecord {
    name: String,
    customer_id: String,
}
```

That macro simply generates an impl block causing
`Deserialize`.

This is the function that determines how to create th
based on the struct's fields. When the parsing library
parsing library - it creates a `Deserializer` and calls
parameter.

The `deserialize` code will then create a `Visitor` v
the `Deserializer`. If everything goes well, eventual
corresponding to the type being parsed and return

For a complete example, see the *Serde* documentati

The result is that types to be deserialized only imple
file formats only need to implement the "bottom lay
with the rest of the ecosystem, since generic types v

In conclusion, Rust's generic-inspired type system ca
use their power, as shown in this API design. But it r
create bridges for its generics.

If you are interested in learning more about this top

See Also

- lens-rs crate for a pre-built lenses implementa
these examples
- Serde itself, which makes these concepts intuit
structs) without needing to understand the de
- luminance is a crate for drawing computer gra
including procedural macros to create full pris
that remain generic
- An Article about Lenses in Scala that is very rea
- Paper: Profunctor Optics: Modular Data Acces
- Musli is a library which attempts to use a simil
e.g. doing away with the visitor

[1] School of Haskell: A Little Lens Starter Tutorial

[2] Concordance on Wikipedia

Additional resources

A collection of complementary helpful content

Talks

- Design Patterns in Rust by Nicholas Cameron a
- Writing Idiomatic Libraries in Rust by Pascal He
- Rust Programming Techniques by Nicholas Ca

Books (Online)

- The Rust API Guidelines

Design principles

A brief overview over common d

SOLID

- Single Responsibility Principle (SRP): A class sh
that is, only changes to one part of the softwar
affect the specification of the class.
- Open/Closed Principle (OCP): "Software entitie
closed for modification."
- Liskov Substitution Principle (LSP): "Objects in
instances of their subtypes without altering th
- Interface Segregation Principle (ISP): "Many clie
one general-purpose interface."
- Dependency Inversion Principle (DIP): One sho
concretions."

DRY (Don't Repeat Yourself)

"Every piece of knowledge must have a single, unam
within a system"

KISS principle

most systems work best if they are kept simple rath
simplicity should be a key goal in design, and unnec

Law of Demeter (LoD)

a given object should assume as little as possible ab
anything else (including its subcomponents), in acco

"information hiding"

Design by contract (DbC)

software designers should define formal, precise an
for software components, which extend the ordinar
with preconditions, postconditions and invariants

Encapsulation

bundling of data with the methods that operate on t
access to some of an object's components. Encapsu
state of a structured data object inside a class, prev
access to them.

Command-Query-Separation(CQ

"Functions should not produce abstract side effects.
permitted to produce side effects." - Bertrand Meye
Construction

Principle of least astonishment (

a component of a system should behave in a way th
The behavior should not astonish or surprise users

Linguistic-Modular-Units

"Modules must correspond to syntactic units in the
Object-Oriented Software Construction

Self-Documentation

“The designer of a module should strive to make all
the module itself.” - Bertrand Meyer: Object-Oriente

Uniform-Access

“All services offered by a module should be availabl
does not betray whether they are implemented thro
computation.” - Bertrand Meyer: Object-Oriented Sc

Single-Choice

“Whenever a software system must support a set of
module in the system should know their exhaustive
Oriented Software Construction

Persistence-Closure

“Whenever a storage mechanism stores an object, it
that object. Whenever a retrieval mechanism retriev
also retrieve any dependent of that object that has r
Meyer: Object-Oriented Software Construction