Introduction

Participation

If you are interested in contributing to this book, check out the contribution guidelines.

Design patterns

In software development, we often come across problems that share similarities regardless of the environment they appear in. Although the implementation details are crucial to solve the task at hand, we may abstract from these particularities to find the common practices that are generically applicable.

Design patterns are a collection of reusable and tested solutions to recurring problems in engineering. They make our software more modular, maintainable, and extensible. Moreover, these patterns provide a common language for developers, making them an excellent tool for effective communication when problem-solving in teams.

Design patterns in Rust

Rust is not object-oriented, and the combination of all its characteristics, such as functional elements, a strong type system, and the borrow checker, makes it unique. Because of this, Rust design patterns vary with respect to other traditional object-oriented programming languages. That's why we decided to write this book. We hope you enjoy reading it! The book is divided in three main chapters:

- Idioms: guidelines to follow when coding. They are the social norms of the community. You should break them only if you have a good reason for it.
- Design patterns: methods to solve common problems when coding.
- Anti-patterns: methods to solve common problems when coding. However, while design patterns give us benefits, anti-patterns create more problems.

Translations

We are utilizing mdbook-i18n-helper. Please read up on how to *add* and *update* translations in their repository

External translations

- 简体中文

If you want to add a translation, please open an issue in the main repository.

Idioms

Idioms are commonly used styles, guidelines and patterns largely agreed upon by a community. Writing idiomatic code allows other developers to understand better what is happening.

After all, the computer only cares about the machine code that is generated by the compiler. Instead, the source code is mainly beneficial to the developer. So, since we have this abstraction layer, why not make it more readable?

Remember the KISS principle: “Keep It Simple, Stupid”. It claims that “most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided”.

> Code is there for humans, not computers, to understand.

Use borrowed types for arguments

Description

Using a target of a deref coercion can increase the flexibility of your code when you are deciding which argument type to use for a function argument. In this way, the function will accept more input types.

This is not limited to slice-able or fat pointer types. In fact, you should always prefer using the **borrowed type** over **borrowing the owned type**. Such as `&str` over `&String`, `&[T]` over `&Vec<T>`, or `&T` over `&Box<T>`.

Using borrowed types you can avoid layers of indirection for those instances where the owned type already provides a layer of indirection. For instance, a `String` has a layer of indirection, so a `&String` will have two layers of indirection. We can avoid this by using `&str` instead, and letting `&String` coerce to a `&str` whenever the function is invoked.

Example

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

Consider an example where we wish to determine if a word contains three consecutive vowels. We don't need to own the string to determine this, so we will take a reference.

The code might look something like this:

```
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(&ferris));
    println!("{}: {}", curious, three_vowels(&curious));

    // This works fine, but the following two lines would fail:
    // println!("Ferris: {}", three_vowels("Ferris"));
    // println!("Curious: {}", three_vowels("Curious"));

}
```

This works fine because we are passing a `&String` type as a parameter. If we remove the comments on the last two lines, the example will fail. This is because a `&str` type will not coerce to a `&String` type. We can fix this by simply modifying the type for our argument.

For instance, if we change our function declaration to:

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

then both versions will compile and print the same output.

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

But wait, that's not all! There is more to this story. It's likely that you may say to yourself: that doesn't matter, I will never be using a `&'static str` as an input anyways (as we did when we used `"Ferris"`). Even ignoring this special example, you may still find that using `&str` will give you more flexibility than using a `&String`.

Let's now take an example where someone gives us a sentence, and we want to determine if any of the words in the sentence contain three consecutive vowels. We probably should make use of the function we have already defined and simply feed in 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 friendly curious crab named
Ferris".to_string();
    for word in sentence_string.split(' ') {
        if three_vowels(word) {
            println!("{} has three consecutive vowels!", word);
        }
    }
}
```

Running this example using our function declared with an argument type `&str` will yield

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

However, this example will not run when our function is declared with an argument type `&String`. This is because string slices are a `&str` and not a `&String` which would require an allocation to be converted to `&String` which is not implicit, whereas converting from `String` to `&str` is cheap and implicit.

See also

- Rust Language Reference on Type Coercions
- For more discussion on how to handle `String` and `&str` see this blog series (2015) by Herman J. Radtke III

Concatenating strings with `format!`

Description

It is possible to build up strings using the `push` and `push_str` methods on a mutable `String`, or using its `+` operator. However, it is often more convenient to use `format!`, especially where there is a mix of literal and non-literal strings.

Example

```
fn say_hello(name: &str) -> String {
    // We could construct the result string manually.
    // 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 readable way to combine strings.

Disadvantages

It is usually not the most efficient way to combine strings - a series of `push` operations on a mutable string is usually the most efficient (especially if the string has been pre-allocated to the expected size).

Constructors

Description

Rust does not have constructors as a language construct. Instead, the convention is to 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 - no self.
    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 `Default` trait:

```
/// 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 implement `Default`, like they do with `Second`:

```
/// 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 implement both `Default` and an empty `new` constructor. `new` is the constructor convention in Rust, and users expect it to exist, so if it is reasonable for the basic constructor to take no arguments, then it should, even if it is functionally identical to default.

Hint: The advantage of implementing or deriving `Default` is that your type can now be used where a `Default` implementation is required, most prominently, any of the `*or_default` functions in the standard library.

See also

- The default idiom for a more in-depth description of the `Default` trait.
- The builder pattern for constructing objects where there are multiple configurations.
- API Guidelines/C-COMMON-TRAITS for implementing both, `Default` and `new`.

The `Default` Trait

Description

Many types in Rust have a constructor. However, this is *specific* to the type; Rust cannot abstract over "everything that has a `new()` method". To allow this, the `Default` trait was conceived, which can be used with containers and other generic types (e.g. see `Option::unwrap_or_default()`). Notably, some containers already implement it where applicable.

Not only do one-element containers like `Cow`, `Box` or `Arc` implement `Default` for contained `Default` types, one can automatically `#[derive(Default)]` for structs whose fields all implement it, so the more types implement `Default`, the more useful it becomes.

On the other hand, constructors can take multiple arguments, while the `default()` method does not. There can even be multiple constructors with different names, but there can only be one `Default` implementation per type.

Example

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

// note that we can simply auto-derive Default here.
#[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 values
    let mut conf = MyConfiguration::default();
    // do something with conf here
    conf.check = true;
    println!("conf = {:#?}", conf);

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

See also

- The constructor idiom is another way to generate instances that may or may not be "default"
- The `Default` documentation (scroll down for the list of implementors)
- `Option::unwrap_or_default()`
- `derive(new)`

Collections are smart pointers

Description

Use the `Deref` trait to treat collections like smart pointers, offering owning and borrowed 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 slice (`&[T]`) is a borrowed collection of `T`s. Implementing `Deref` for `Vec` allows implicit dereferencing from `&Vec<T>` to `&[T]` and includes the relationship in auto-derefencing searches. Most methods you might expect to be implemented for `Vec`s are instead implemented for slices.

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

Motivation

Ownership and borrowing are key aspects of the Rust language. Data structures must account for these semantics properly to give a good user experience. When implementing a data structure that owns its data, offering a borrowed view of that data allows for more flexible APIs.

Advantages

Most methods can be implemented only for the borrowed view, they are then implicitly available for the owning view.

Gives clients a choice between borrowing or taking ownership of data.

Disadvantages

Methods and traits only available via dereferencing are not taken into account when bounds checking, so generic programming with data structures using this pattern can get complex (see the `Borrow` and `AsRef` traits, etc.).

Discussion

Smart pointers and collections are analogous: a smart pointer points to a single object, whereas a collection points to many objects. From the point of view of the type system, there is little difference between the two. A collection owns its data if the only way to access each datum is via the collection and the collection is responsible for deleting the data (even in cases of shared ownership, some kind of borrowed view may be appropriate). If a collection owns its data, it is usually useful to provide a view of the data as borrowed so that it can be referenced multiple times.

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

Commonly, ordered collections will implement `Index` for `Range`s to provide slicing syntax. The target will be the borrowed view.

See also

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

Finalisation in destructors

Description

Rust does not provide the equivalent to `finally` blocks - code that will be executed no matter how a function is exited. Instead, an object's destructor can be used to run code that must be run before exit.

Example

```
fn bar() -> Result<(), ()> {
    // These don't need to be defined inside the function.
    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 function `bar` is exited.
    let _exit = Foo;
    // Implicit return with `?` operator.
    baz()?;
    // Normal return.
    Ok(())
}
```

Motivation

If a function has multiple return points, then executing code on exit becomes difficult and repetitive (and thus bug-prone). This is especially the case where return is implicit due to a macro. A common case is the `?` operator which returns if the result is an `Err`, but continues if it is `Ok`. `?` is used as an exception handling mechanism, but unlike Java (which has `finally`), there is no way to schedule code to run in both the normal and exceptional cases. Panicking will also exit a function early.

Advantages

Code in destructors will (nearly) always be run - copes with panics, early returns, etc.

Disadvantages

It is not guaranteed that destructors will run. For example, if there is an infinite loop in a function or if running a function crashes before exit. Destructors are also not run in the case of a panic in an already panicking thread. Therefore, destructors cannot be relied on as finalizers where it is absolutely essential that finalisation happens.

This pattern introduces some hard to notice, implicit code. Reading a function gives no clear indication of destructors to be run on exit. This can make debugging tricky.

Requiring an object and `Drop` impl just for finalisation is heavy on boilerplate.

Discussion

There is some subtlety about how exactly to store the object used as a finalizer. It must be kept alive until the end of the function and must then be destroyed. The object must always be a value or uniquely owned pointer (e.g., `Box<Foo>`). If a shared pointer (such as `Rc`) is used, then the finalizer can be kept alive beyond the lifetime of the function. For similar reasons, the finalizer should not be moved or returned.

The finalizer must be assigned into a variable, otherwise it will be destroyed immediately, rather than when it goes out of scope. The variable name must start with `_` if the variable is only used as a finalizer, otherwise the compiler will warn that the finalizer is never used. However, do not call the variable `_` with no suffix - in that case it will be destroyed immediately.

In Rust, destructors are run when an object goes out of scope. This happens whether we reach the end of block, there is an early return, or the program panics. When panicking, Rust unwinds the stack running destructors for each object in each stack frame. So, destructors get called even if the panic happens in a function being called.

If a destructor panics while unwinding, there is no good action to take, so Rust aborts the thread immediately, without running further destructors. This means that destructors are not absolutely guaranteed to run. It also means that you must take extra care in your destructors not to panic, since it could leave resources in an unexpected state.

See also

RAII guards.

`mem::{take(_), replace(_)}` to keep owned values in changed enums

Description

Say we have a `&mut MyEnum` which has (at least) two variants, `A { name: String, x: u8 }` and `B { name: String }`. Now we want to change `MyEnum::A` to a `B` if `x` is zero, 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 put in an empty String instead
        // (note that empty strings don't allocate).
        // Then, construct the new enum variant (which will
        // 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 taking `name` by value, but we cannot
        // take the value out of a mutable reference, unless we replace it:
        A { name } => B { name: mem::take(name) },
        B { name } => A { name: mem::take(name) },
        C => D,
        D => C
    }
}
```

Motivation

When working with enums, we may want to change an enum value in place, perhaps to another variant. This is usually done in two phases to keep the borrow checker happy. In the first phase, we observe the existing value and look at its parts to decide what to do next. In the second phase we may conditionally change the value (as in the example above).

The borrow checker won't allow us to take out `name` of the enum (because *something* must be there.) We could of course `.clone()` name and put the clone into our `MyEnum::B`, but that would be an instance of the Clone to satisfy the borrow checker anti-pattern. Anyway, we can avoid the extra allocation by changing `e` with only a mutable borrow.

`mem::take` lets us swap out the value, replacing it with it's default value, and returning the previous value. For `String`, the default value is an empty `String`, which does not need to allocate. As a result, we get the original `name` *as an owned value*. We can then wrap this in another enum.

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

Note, however, that if we are using an `Option` and want to replace its value with a `None`, `Option`'s `take()` method provides a shorter and more idiomatic alternative.

Advantages

Look ma, no allocation! Also you may feel like Indiana Jones while doing it.

Disadvantages

This gets a bit wordy. Getting it wrong repeatedly will make you hate the borrow checker. The compiler may fail to optimize away the double store, resulting in reduced performance as opposed to what you'd do in unsafe languages.

Furthermore, the type you are taking needs to implement the `Default` trait. However, if the type you're working with doesn't implement this, you can instead use `mem::replace`.

Discussion

This pattern is only of interest in Rust. In GC'd languages, you'd take the reference to the value by default (and the GC would keep track of refs), and in other low-level languages like C you'd simply alias the pointer and fix things later.

However, in Rust, we have to do a little more work to do this. An owned value may only have one owner, so to take it out, we need to put something back in – like Indiana Jones, replacing the artifact with a bag of sand.

See also

This gets rid of the Clone to satisfy the borrow checker anti-pattern in a specific case.

On-Stack Dynamic Dispatch

Description

We can dynamically dispatch over multiple values, however, to do so, we need to declare multiple variables to bind differently-typed objects. To extend the lifetime as necessary, we can use deferred conditional initialization, as seen below:

Example

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


// These must live longer than `readable`, and thus are declared first:
let (mut stdin_read, mut file_read);

// We need to ascribe the type to get dynamic dispatch.
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 copy of the code will be generated for each type it is used with and optimized independently. While this allows for very fast code on the hot path, it also bloats the code in places where performance is not of the essence, thus costing compile time and cache usage.

Luckily, Rust allows us to use dynamic dispatch, but we have to explicitly ask for it.

Advantages

We do not need to allocate anything on the heap. Neither do we need to initialize something we won't use later, nor do we need to monomorphize the whole code that follows to work with both `File` or `Stdin`.

Disadvantages

The code needs more moving parts than the `Box`-based version:

```
// We still need to ascribe the type for dynamic dispatch.
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 requires all variables to be initialized *before use*, so it's easy to overlook the fact that *unused* variables may well be uninitialized. Rust works quite hard to ensure that this works out fine and only the initialized values are dropped at the end of their scope.

The example meets all the constraints Rust places on us:

- All variables are initialized before using (in this case borrowing) them
- Each variable only holds values of a single type. In our example, `stdin` is of type `Stdin`, `file` is of type `File` and `readable` is of type `&mut dyn Read`
- Each borrowed value outlives all the references borrowed from it

See also

- Finalisation in destructors and RAII guards can benefit from tight control over lifetimes.
- For conditionally filled `Option<&T>`s of (mutable) references, one can initialize an `Option<T>` directly and use its `.as_ref()` method to get an optional reference.

FFI Idioms

Writing FFI code is an entire course in itself. However, there are several idioms here that can act as pointers, and avoid traps for inexperienced users of `unsafe` Rust.

This section contains idioms that may be useful when doing FFI.

1. Idiomatic Errors - Error handling with integer codes and sentinel return values (such 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 by return codes. However, Rust's type system allows much more rich error information to be captured and propogated through a full type.

This best practice shows different kinds of error codes, and how to expose them in a usable way:

1. Flat Enums should be converted to integers and returned as codes.
2. Structured Enums should be converted to an integer code with a string error message for detail.
3. Custom Error Types should become "transparent", with a C representation.

Code Example

Flat Enums

```
enum DatabaseError {
    IsReadOnly = 1, // user attempted a write operation
    IOError = 2, // user should read the C errno() for what it was
    FileCorrupted = 3, // user should run a repair tool to recover it
}

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 describing the issue
    }

    impl From<DatabaseError> for libc::c_int {
        fn from(e: DatabaseError) -> libc::c_int {
            match e {
                DatabaseError::IsReadOnly => 1,
                DatabaseError::IOError(_) => 2,
                DatabaseError::FileCorrupted(_) => 3,
            }
        }
    }
}

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 greater than the current stack frame
            &*e
        };

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

        let c_error = unsafe {
            // SAFETY: copying error_str to an allocated buffer with a NUL
            // character at the end
            let mut malloc: *mut u8 = libc::malloc(error_str.len() + 1) as *mut
_;

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

            let src = error_str.as_bytes().as_ptr();
```

```
            std::ptr::copy_nonoverlapping(src, malloc, error_str.len());

            std::ptr::write(malloc.add(error_str.len()), 0);

            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 as a C structure */
#[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 } = e;
        parse_error { expected, line, ch }
    }
}
```

Advantages

This ensures that the foreign language has clear access to error information while not compromising the Rust code's API at all.

Disadvantages

It's a lot of typing, and some types may not be able to be converted easily to C.

Accepting Strings

Description

When accepting strings via FFI through pointers, there are two principles that should be followed:

1. Keep foreign strings “borrowed”, rather than copying them directly.
2. Minimize the amount of complexity and `unsafe` code involved in converting from a C-style string to native Rust strings.

Motivation

The strings used in C have different behaviours to those used in Rust, namely:

- C strings are null-terminated while Rust strings store their length
- C strings can contain any arbitrary non-zero byte while Rust strings must be UTF-8
- C strings are accessed and manipulated using `unsafe` pointer operations while interactions with Rust strings go through safe methods

The Rust standard library comes with C equivalents of Rust’s `String` and `&str` called `CString` and `&CStr`, that allow us to avoid a lot of the complexity and `unsafe` code involved in converting between C strings and Rust strings.

The `&CStr` type also allows us to work with borrowed data, meaning passing strings 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 ensure `msg`:
    ///
    /// - is not a null pointer
    /// - points to valid, initialized data
    /// - points to memory ending in a null byte
    /// - won't be mutated for the duration of this function call
    #[no_mangle]
    pub unsafe extern "C" fn mylib_log(
        msg: *const libc::c_char,
        level: libc::c_int
    ) {
        let level: crate::LogLevel = match level { /* ... */ };

        // SAFETY: The caller has already guaranteed this is okay (see the
        // `# Safety` section of the doc-comment).
        let msg_str: &str = match std::ffi::CStr::from_ptr(msg).to_str() {
            Ok(s) => s,
            Err(e) => {
                crate::log_error("FFI string conversion failed");
                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 becomes a “tracked” shared reference

Consider an alternative, where the string is actually copied:

```
pub mod unsafe_module {

    // other module content

    pub extern "C" fn mylib_log(msg: *const libc::c_char, level: libc::c_int) {
        // DO NOT USE THIS CODE.
        // IT IS UGLY, VERBOSE, AND CONTAINS A SUBTLE BUG.

        let level: crate::LogLevel = match level { /* ... */ };

        let msg_len = unsafe { /* SAFETY: strlen is what it is, I guess? */
            libc::strlen(msg)
        };

        let mut msg_data = Vec::with_capacity(msg_len + 1);

        let msg_cstr: std::ffi::CString = unsafe {
            // SAFETY: copying from a foreign pointer expected to live
            // for the entire stack frame into owned memory
            std::ptr::copy_nonoverlapping(msg, msg_data.as_mut(), msg_len);

            msg_data.set_len(msg_len + 1);

            std::ffi::CString::from_vec_with_nul(msg_data).unwrap()
        }

        let msg_str: String = unsafe {
            match msg_cstr.into_string() {
                Ok(s) => s,
                Err(e) => {
                    crate::log_error("FFI string conversion failed");
                    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 importantly, more invariants it must uphold.
2. Due to the extensive arithmetic required, there is a bug in this version that cases Rust `undefined behaviour`.

The bug here is a simple mistake in pointer arithmetic: the string was copied, all `msg_len` bytes of it. However, the `NUL` terminator at the end was not.

The Vector then had its size *set* to the length of the *zero padded string* – rather than *resized* to it, which could have added a zero at the end. As a result, the last byte in the Vector is uninitialized memory. When the `CString` is created at the bottom of the block, its read of the Vector will cause `undefined behaviour`!

Like many such issues, this would be difficult issue to track down. Sometimes it would panic because the string was not `UTF-8`, sometimes it would put a weird character at the end of the string, sometimes it would just completely crash.

Disadvantages

None?

Passing Strings

Description

When passing strings to FFI functions, there are four principles that should be followed:

1. Make the lifetime of owned strings as long as possible.
2. Minimize `unsafe` code during the conversion.
3. If the C code can modify the string data, use `Vec` instead of `CString`.
4. Unless the Foreign Function API requires it, the ownership of the string should not transfer to the callee.

Motivation

Rust has built-in support for C-style strings with its `CString` and `CStr` types. However, there are different approaches one can take with strings that are being sent to a foreign function call from a Rust function.

The best practice is simple: use `CString` in such a way as to minimize `unsafe` code. However, a secondary caveat is that *the object must live long enough*, meaning the lifetime should be maximized. In addition, the documentation explains that “round-tripping” a `CString` after modification is UB, so additional work is necessary in that case.

Code Example

```
pub mod unsafe_module {

    // other module content

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

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

        unsafe {
            // SAFETY: calling an FFI whose documentation says the pointer is
            // const, so no modification should occur
            seterr(c_err.as_ptr());
        }

        Ok(())
        // The lifetime of c_err continues until here
    }

    fn get_error_from_ffi() -> Result<String, std::ffi::IntoStringError> {
        let mut buffer = vec![0u8; 1024];
        unsafe {
            // SAFETY: calling an FFI whose documentation implies
            // that the input need only live as long as the call
            let written: usize = geterr(buffer.as_mut_ptr(), 1023).into();

            buffer.truncate(written + 1);
        }

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

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 when possible.

A common mistake (so common it's in the documentation) is to not use the variable in the first block:

```
pub mod unsafe_module {

    // other module content

    fn report_error<S: Into<String>>(err: S) -> Result<(), std::ffi::NulError>
{
        unsafe {
            // SAFETY: whoops, this contains a dangling pointer!
            seterr(std::ffi::CString::new(err.into())?.as_ptr());
        }
        Ok(())
    }
}
```

This code will result in a dangling pointer, because the lifetime of the `CString` is not extended by the pointer creation, unlike if a reference were created.

Another issue frequently raised is that the initialization of a 1k vector of zeroes is “slow”. However, recent versions of Rust actually optimize that particular macro to a call to `zmalloc`, meaning it is as fast as the operating system’s ability to return zeroed memory (which is quite fast).

Disadvantages

None?

Iterating over an `Option`

Description

`Option` can be viewed as a container that contains either zero or one element. In particular, it implements the `IntoIterator` trait, and as such can be used with generic code that needs such a type.

Examples

Since `Option` implements `IntoIterator`, it can be used as an argument to `.extend()`:

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

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 existing iterator, you can pass it to `.chain()`:

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

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

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

Also, since `Option` implements `IntoIterator`, it's possible to iterate over it using a `for` loop. This is equivalent to matching it with `if let Some(..)`, and in most cases you should prefer the latter.

See also

- `std::iter::once` is an iterator which yields exactly one element. It's a more readable alternative to `Some(foo).into_iter()`.
- `Iterator::filter_map` is a version of `Iterator::map`, specialized to mapping functions which return `Option`.
- The `ref_slice` crate provides functions for converting an `Option` to a zero- or one-element slice.
- Documentation for `Option<T>`

Pass variables to closure

Description

By default, closures capture their environment by borrowing. Or you can use `move`-closure to move whole environment. However, often you want to move just some variables to closure, give it copy of some data, pass it by reference, or perform some 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 cloned
    let num3 = num3.as_ref();  // `num3` is borrowed
    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 definition, so their purpose is more clear, and they will be dropped immediately even if they are not consumed by closure.

Closure uses same variable names as surrounding code whether data are copied or moved.

Disadvantages

Additional indentation of closure body.

`#[non_exhaustive]` and private fields for extensibility

Description

A small set of scenarios exist where a library author may want to add public fields to a public struct or new variants to an enum without breaking backwards compatibility.

Rust offers two solutions to this problem:

- Use `#[non_exhaustive]` on `struct`s, `enum`s, and `enum` variants. For extensive documentation on all the places where `#[non_exhaustive]` can be used, see the docs.
- You may add a private field to a struct to prevent it from being directly instantiated or matched against (see Alternative)

Example

```
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 cannot be named here and
    // we must use `..` in the pattern.
    let a::S { foo: _, ..} = s;

    let some_enum = a::AdmitMoreVariants::VariantA;
    match some_enum {
        a::AdmitMoreVariants::VariantA => println!("it's an A"),
        a::AdmitMoreVariants::VariantB => println!("it's a b"),

        // .. required because this variant is non-exhaustive as well
        a::AdmitMoreVariants::VariantC { a, .. } => println!("it's a c"),

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

Alternative: `Private fields` for structs

`#[non_exhaustive]` only works across crate boundaries. Within a crate, the private field method may be used.

Adding a field to a struct is a mostly backwards compatible change. However, if a client uses a pattern to deconstruct a struct instance, they might name all the fields in the struct and adding a new one would break that pattern. The client could name some fields and use `..` in the pattern, in which case adding another field is backwards compatible. Making at least one of the struct's fields private forces clients to use the latter form of patterns, ensuring that the struct is future-proof.

The downside of this approach is that you might need to add an otherwise unneeded field to the struct. You can use the `()` type so that there is no runtime overhead and prepend `_` to the field name to avoid the unused field warning.

```
pub struct S {
    pub a: i32,
    // Because `b` is private, you cannot match on `S` without using `..` and
`S`
    //  cannot be directly instantiated or matched against
    _b: ()
}
```

Discussion

On `struct`s, `#[non_exhaustive]` allows adding additional fields in a backwards compatible way. It will also prevent clients from using the struct constructor, even if all the fields are public. This may be helpful, but it's worth considering if you *want* an additional field to be found by clients as a compiler error rather than something that may be silently undiscovered.

`#[non_exhaustive]` can be applied to enum variants as well. A `#[non_exhaustive]` variant behaves in the same way as a `#[non_exhaustive]` struct.

Use this deliberately and with caution: incrementing the major version when adding fields or variants is often a better option. `#[non_exhaustive]` may be appropriate in scenarios where you're modeling an external resource that may change out-of-sync with your library, but is not a general purpose tool.

Disadvantages

`#[non_exhaustive]` can make your code much less ergonomic to use, especially when forced to handle unknown enum variants. It should only be used when these sorts of evolutions are required **without** incrementing the major version.

When `#[non_exhaustive]` is applied to `enum`s, it forces clients to handle a wildcard variant. If there is no sensible action to take in this case, this may lead to awkward code and code paths that are only executed in extremely rare circumstances. If a client decides to `panic!()` in this scenario, it may have been better to expose this error at compile time. In fact, `#[non_exhaustive]` forces clients to handle the "Something else" case; there is rarely a sensible action to take in this scenario.

See also

- RFC introducing #[non_exhaustive] attribute for enums and structs

Easy doc initialization

Description

If a struct takes significant effort to initialize when writing docs, it can be quicker to wrap your example with a helper function which takes the struct as an argument.

Motivation

Sometimes there is a struct with multiple or complicated parameters and several methods. Each of these methods should have examples.

For example:

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

impl Connection {
    /// Sends a request over the connection.
    ///
    /// # Example
    /// ```no_run
    /// # // Boilerplate are required to get an example working.
    /// # let stream = TcpStream::connect("127.0.0.1:34254");
    /// # let connection = Connection { name: "foo".to_owned(), stream };
    /// # let request = Request::new("RequestId", RequestType::Get, "payload");
    /// let response = connection.send_request(request);
    /// assert!(response.is_ok());
    /// ```
    fn send_request(&self, request: Request) -> Result<Status, SendErr> {
        // ...
    }

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

Example

Instead of typing all of this boilerplate to create a `Connection` and `Request`, it is easier to just create a wrapping helper function which takes them as arguments:

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

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

Note in the above example the line `assert!(response.is_ok());` will not actually run while testing because it is inside a function which is never invoked.

Advantages

This is much more concise and avoids repetitive code in examples.

Disadvantages

As example is in a function, the code will not be tested. Though it will still be checked to make sure it compiles when running a `cargo test`. So this pattern is most useful when you need `no_run`. With this, you do not need to add `no_run`.

Discussion

If assertions are not required this pattern works well.

If they are, an alternative can be to create a public method to create a helper instance which is annotated with `#[doc(hidden)]` (so that users won't see it). Then this method can be called inside of rustdoc because it is part of the crate's public API.

Temporary mutability

Description

Often it is necessary to prepare and process some data, but after that data are only inspected and never modified. The intention can be made explicit by redefining the mutable variable as immutable.

It can be done either by processing data within a nested block or by redefining the 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 data after some point.

Disadvantages

Nested block requires additional indentation of block body. One more line to return data from block or redefine variable.

Return consumed argument on error

Description

If a fallible function consumes (moves) an argument, return that argument back inside an error.

Example

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

pub struct SendError(String);

fn main() {
    let mut value = "imagine this is very long string".to_string();

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

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

Motivation

In case of error you may want to try some alternative way or to retry action in case of non-deterministic function. But if the argument is always consumed, you are forced to clone it on every call, which is not very efficient.

The standard library uses this approach in e.g. `String::from_utf8` method. When given a vector that doesn't contain valid UTF-8, a `FromUtf8Error` is returned. You can get original vector back using `FromUtf8Error::into_bytes` method.

Advantages

Better performance because of moving arguments whenever possible.

Disadvantages

Slightly more complex error types.

Design Patterns

Design patterns are “general reusable solutions to a commonly occurring problem within a given context in software design”. Design patterns are a great way to describe the culture of a programming language. Design patterns are very language-specific - what is a pattern in one language may be unnecessary in another due to a language feature, or impossible to express due to a missing feature.

If overused, design patterns can add unnecessary complexity to programs. However, they are a great way to share intermediate and advanced level knowledge about a programming language.

Design patterns in Rust

Rust has many unique features. These features give us great benefit by removing whole classes of problems. Some of them are also patterns that are *unique* to Rust.

YAGNI

YAGNI is an acronym that stands for `You Aren't Going to Need It`. It’s a vital software design principle to apply as you write code.

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

If we apply YAGNI to design patterns, we see that the features of Rust allow us to throw out many patterns. For instance, there is no need for the strategy pattern in Rust because we can just use traits.

Behavioural Patterns

From Wikipedia:

> Design patterns that identify common communication patterns among objects. By doing so, these patterns increase flexibility in carrying out communication.

Command

Description

The basic idea of the Command pattern is to separate out actions into its own objects and pass them as parameters.

Motivation

Suppose we have a sequence of actions or transactions encapsulated as objects. We want these actions or commands to be executed or invoked in some order later at different time. These commands may also be triggered as a result of some event. For example, when a user pushes a button, or on arrival of a data packet. In addition, these commands might be undoable. This may come in useful for operations of an editor. We might want to store logs of executed commands so that we could reapply the changes later if the system crashes.

Example

Define two database operations `create table` and `add field`. Each of these operations is a command which knows how to undo the command, e.g., `drop table` and `remove field`. When a user invokes a database migration operation then each command is executed in the defined order, and when the user invokes the rollback operation then the whole set of commands is invoked in reverse order.

Approach: Using trait objects

We define a common trait which encapsulates our command with two operations `execute` and `rollback`. All command `structs` must implement this trait.

```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 Migration>) {
        self.commands.push(cmd);
    }

    fn execute(&self) -> Vec<&str> {
        self.commands.iter().map(|cmd| cmd.execute()).collect()
    }
    fn rollback(&self) -> Vec<&str> {
        self.commands
            .iter()
            .rev() // reverse iterator's direction
            .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 field"], schema.execute());
    assert_eq!(vec!["remove field", "drop table"], schema.rollback());
}
```

Approach: Using function pointers

We could follow another approach by creating each individual command as a different function and store function pointers to invoke these functions later at a different time. Since function pointers implement all three traits `Fn`, `FnMut`, and `FnOnce` we could as well pass and store closures instead of function pointers.

```
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: FnPtr, rollback: FnPtr) {
        self.commands.push(Command { execute, rollback });
    }
    fn execute(&self) -> Vec<String> {
        self.commands.iter().map(|cmd| (cmd.execute)()).collect()
    }
    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_string(), || "drop
table".to_string());
    schema.add_migration(add_field, remove_field);
    assert_eq!(vec!["create table", "add field"], schema.execute());
    assert_eq!(vec!["remove field", "drop table"], schema.rollback());
}
```

Approach: Using `Fn` trait objects

Finally, instead of defining a common command trait we could store each command implementing the `Fn` trait separately in vectors.

```
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: E, rollback: R)
    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()).collect()
    }
    fn rollback(&self) -> Vec<&str> {
        self.rollbacks.iter().rev().map(|cmd| cmd()).collect()
    }
}

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", || "drop table");
    schema.add_migration(add_field, remove_field);
    assert_eq!(vec!["create table", "add field"], schema.execute());
    assert_eq!(vec!["remove field", "drop table"], schema.rollback());
}
```

Discussion

If our commands are small and may be defined as functions or passed as a closure then using function pointers might be preferable since it does not exploit dynamic dispatch. But if our command is a whole struct with a bunch of functions and variables defined as seperated module then using trait objects would be more suitable. A case of application can be found

in `actix`, which uses trait objects when it registers a handler function for routes. In case of using `Fn` trait objects we can create and use commands in the same way as we used in case of function pointers.

As performance, there is always a trade-off between performance and code simplicity and organisation. Static dispatch gives faster performance, while dynamic dispatch provides flexibility when we structure our application.

See also

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

Interpreter

Description

If a problem occurs very often and requires long and repetitive steps to solve it, then the problem instances might be expressed in a simple language and an interpreter object could solve it by interpreting the sentences written in this simple language.

Basically, for any kind of problems we define:

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

Motivation

Our goal is to translate simple mathematical expressions into postfix expressions (or Reverse Polish notation) For simplicity, our expressions consist of ten digits `0`, ..., `9` and two operations `+`, `-`. For example, the expression `2 + 4` is translated into `2 4 +`.

Context Free Grammar for our problem

Our task is translating infix expressions into postfix ones. Let's define a context free grammar for a set of infix expressions over `0`, ..., `9`, `+`, and `-`, where:

- 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 depending on what we are going to do with it. For example, we might need to remove left recursion. For more details please see Compilers: Principles,Techniques, and Tools (aka Dragon Book).

Solution

We simply implement a recursive descent parser. For simplicity's sake, the code panics when an expression is syntactically wrong (for example `2-34` or `2+5-` are wrong 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 String) {
        self.term(out);

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

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

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 Interpreter design pattern is about design grammars for formal languages and implementation of parsers for these grammars. In fact, this pattern is about expressing problem instances in a more specific way and implementing functions/classes/structs that solve these problem instances. Rust language has `macro_rules!` that allow us to define special syntax and rules on how to expand this syntax into source code.

In the following example we create a simple `macro_rules!` that computes Euclidean length of `n` dimensional vectors. Writing `norm!(x,1,2)` might be easier to express and more efficient than packing `x,1,2` into a `Vec` and calling a function computing the length.

```
macro_rules! norm {
    ($($element:expr),*) => {
        {
            let mut n = 0.0;
            $(
                n += ($element as f64)*($element as f64);
            )*
            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, -0.5));
}
```

See also

- Interpreter pattern
- Context free grammar
- macro_rules!

Newtype

What if in some cases we want a type to behave similar to another type or enforce some behaviour at compile time when using only type aliases would not be enough?

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

For such cases we could use the `Newtype` pattern to provide **type safety** and **encapsulation**.

Description

Use a tuple struct with a single field to make an opaque wrapper for a type. This creates a new type, rather than an alias to a type (`type` items).

Example

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

// Create Newtype Password to override the Display trait for String
struct Password(String);

impl Display for Password {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "****************")
    }
}

fn main() {
    let unsecured_password: String = "ThisIsMyPassword".to_string();
    let secured_password: Password = Password(unsecured_password.clone());
    println!("unsecured_password: {unsecured_password}");
    println!("secured_password: {secured_password}");
}
```

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

Motivation

The primary motivation for newtypes is abstraction. It allows you to share implementation details between types while precisely controlling the interface. By using a newtype rather than exposing the implementation type as part of an API, it allows you to change implementation backwards compatibly.

Newtypes can be used for distinguishing units, e.g., wrapping `f64` to give distinguishable `Miles` and `Kilometres`.

Advantages

The wrapped and wrapper types are not type compatible (as opposed to using `type`), so users of the newtype will never 'confuse' the wrapped and wrapper types.

Newtypes are a zero-cost abstraction - there is no runtime overhead.

The privacy system ensures that users cannot access the wrapped type (if the field is private, which it is by default).

Disadvantages

The downside of newtypes (especially compared with type aliases), is that there is no special language support. This means there can be *a lot* of boilerplate. You need a 'pass through' method for every method you want to expose on the wrapped type, and an impl for every trait you want to also be implemented for the wrapper type.

Discussion

Newtypes are very common in Rust code. Abstraction or representing units are the most common uses, but they can be used for other reasons:

- restricting functionality (reduce the functions exposed or traits implemented),
- making a type with copy semantics have move semantics,
- abstraction by providing a more concrete type and thus hiding internal types, e.g.,

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

Here, `Bar` might be some public, generic type and `T1` and `T2` are some internal types. Users of our module shouldn't know that we implement `Foo` by using a `Bar`, but what we're

really hiding here is the types `T1` and `T2`, and how they are used with `Bar`.

See also

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

RAII with guards

Description

RAII stands for “Resource Acquisition is Initialisation” which is a terrible name. The essence of the pattern is that resource initialisation is done in the constructor of an object and finalisation in the destructor. This pattern is extended in Rust by using a RAII object as a guard of some resource and relying on the type system to ensure that access is always mediated by the guard object.

Example

Mutex guards are the classic example of this pattern from the std library (this is a simplified version of the real implementation):

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

struct Foo {}

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

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 self
        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 MutexGuard like a pointer to T.
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 store a reference to the underlying
    // Foo which will outlive the guard xx.

    // x is unlocked when we exit this function and xx's destructor is
executed.
}
```

Motivation

Where a resource must be finalised after use, RAII can be used to do this finalisation. If it is an error to access that resource after finalisation, then this pattern can be used to prevent such errors.

Advantages

Prevents errors where a resource is not finalised and where a resource is used after finalisation.

Discussion

RAII is a useful pattern for ensuring resources are properly deallocated or finalised. We can make use of the borrow checker in Rust to statically prevent errors stemming from using resources after finalisation takes place.

The core aim of the borrow checker is to ensure that references to data do not outlive that data. The RAII guard pattern works because the guard object contains a reference to the underlying resource and only exposes such references. Rust ensures that the guard cannot outlive the underlying resource and that references to the resource mediated by the guard cannot outlive the guard. To see how this works it is helpful to examine the signature of `deref` without lifetime elision:

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

The returned reference to the resource has the same lifetime as `self` (`'a`). The borrow checker therefore ensures that the lifetime of the reference to `T` is shorter than the lifetime of `self`.

Note that implementing `Deref` is not a core part of this pattern, it only makes using the guard object more ergonomic. Implementing a `get` method on the guard works just as well.

See also

Finalisation in destructors idiom

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

Style guide entry (currently just a placeholder).

Strategy (aka Policy)

Description

The Strategy design pattern is a technique that enables separation of concerns. It also allows to decouple software modules through Dependency Inversion.

The basic idea behind the Strategy pattern is that, given an algorithm solving a particular problem, we define only the skeleton of the algorithm at an abstract level, and we separate the specific algorithm's implementation into different parts.

In this way, a client using the algorithm may choose a specific implementation, while the general algorithm workflow remains the same. In other words, the abstract specification of the class does not depend on the specific implementation of the derived class, but specific implementation must adhere to the abstract specification. This is why we call it "Dependency Inversion".

Motivation

Imagine we are working on a project that generates reports every month. We need the reports to be generated in different formats (strategies), e.g., in `JSON` or `Plain Text` formats. But things vary over time, and we don't know what kind of requirement we may get in the future. For example, we may need to generate our report in a completely new format, or just modify one of the existing formats.

Example

In this example our invariants (or abstractions) are `Formatter` and `Report`, while `Text` and `Json` are our strategy structs. These strategies have to implement the `Formatter` trait.

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

type Data = HashMap<String, u32>;

trait Formatter {
    fn format(&self, data: &Data, buf: &mut String);
}

struct Report;

impl Report {
    // Write should be used but we kept it as String to ignore error handling
    fn generate<T: Formatter>(g: T, s: &mut String) {
        // 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 String) {
        for (k, v) in data {
            let entry = format!("{} {}\n", k, v);
            buf.push_str(&entry);
        }
    }
}

struct Json;
impl Formatter for Json {
    fn format(&self, data: &Data, buf: &mut String) {
        buf.push('[');
        for (k, v) in data.into_iter() {
            let entry = format!(r#"{{"{}":"{}"}}"#, k, v);
            buf.push_str(&entry);
            buf.push(',');
        }
        if !data.is_empty() {
            buf.pop(); // remove extra , at the end
        }
        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 example, in this case `Report` does not know anything about specific implementations of `Json` and `Text`, whereas the output implementations does not care about how data is preprocessed, stored, and fetched. The only thing they have to know is a specific trait to implement and its method defining the concrete algorithm implementation processing the result, i.e., `Formatter` and `format(...)`.

Disadvantages

For each strategy there must be implemented at least one module, so number of modules increases with number of strategies. If there are many strategies to choose from then users have to know how strategies differ from one another.

Discussion

In the previous example all strategies are implemented in a single file. Ways of providing different strategies includes:

- All in one file (as shown in this example, similar to being separated as modules)
- Separated as modules, E.g. `formatter::json` module, `formatter::text` module
- Use compiler feature flags, E.g. `json` feature, `text` feature
- Separated as crates, E.g. `json` crate, `text` crate

Serde crate is a good example of the `Strategy` pattern in action. Serde allows full customization of the serialization behavior by manually implementing `Serialize` and `Deserialize` traits for our type. For example, we could easily swap `serde_json` with `serde_cbor` since they expose similar methods. Having this makes the helper crate `serde_transcode` much more useful and ergonomic.

However, we don't need to use traits in order to design this pattern in Rust.

The following toy example demonstrates the idea of the Strategy pattern using Rust `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_adder));
    assert_eq!(0, Adder::add(0, 0, bool_adder));
    assert_eq!(5, Adder::add(1, 3, custom_adder));
}
```

In fact, Rust already uses this idea for `Options`'s `map` method:

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

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

    let first_byte_strategy = |s: &str| s.bytes().next().unwrap();
    assert_eq!(82, val.map(first_byte_strategy).unwrap());
}
```

See also

- Strategy Pattern
- Dependency Injection
- Policy Based Design

Visitor

Description

A visitor encapsulates an algorithm that operates over a heterogeneous collection of objects. It allows multiple different algorithms to be written over the same data without having to modify the data (or their primary behaviour).

Furthermore, the visitor pattern allows separating the traversal of a collection of objects 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) -> T;
        fn visit_stmt(&mut self, s: &Stmt) -> T;
        fn visit_expr(&mut self, e: &Expr) -> T;
    }
}

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

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

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

One could implement further visitors, for example a type checker, without having to modify the AST data.

Motivation

The visitor pattern is useful anywhere that you want to apply an algorithm to heterogeneous data. If data is homogeneous, you can use an iterator-like pattern. Using a visitor object (rather than a functional approach) allows the visitor to be stateful and thus communicate information between nodes.

Discussion

It is common for the `visit_*` methods to return void (as opposed to in the example). In that case it is possible to factor out the traversal code and share it between algorithms (and also to provide noop default methods). In Rust, the common way to do this is to provide `walk_*` functions for each datum. For example,

```
pub fn walk_expr(visitor: &mut Visitor, e: &Expr) {
    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 to have an `accept` method which performs the same duty.

See also

The visitor pattern is a common pattern in most OO languages.

Wikipedia article

The fold pattern is similar to visitor but produces a new version of the visited data structure.

Creational Patterns

From Wikipedia:

> Design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or in added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

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 discover the builder
    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 of Foo.
        FooBuilder {
            bar: String::from("X"),
        }
    }

    pub fn name(mut self, bar: String) -> FooBuilder {
        // Set the name on the builder itself, and return the builder by value.
        self.bar = bar;
        self
    }

    // If we can get away with not consuming the Builder here, that is an
    // advantage. It means we can use the FooBuilder as a template for
constructing
    // many Foos.
    pub fn build(self) -> Foo {
        // Create a Foo from the FooBuilder, applying all settings in
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")).build();
    assert_eq!(foo, foo_from_builder);
}
```

Motivation

Useful when you would otherwise require many constructors or where construction has side effects.

Advantages

Separates methods for building from other methods.

Prevents proliferation of constructors.

Can be used for one-liner initialisation as well as more complex construction.

Disadvantages

More complex than creating a struct object directly, or a simple constructor function.

Discussion

This pattern is seen more frequently in Rust (and for simpler objects) than in many other languages because Rust lacks overloading. Since you can only have a single method with a given name, having multiple constructors is less nice in Rust than in C++, Java, or others.

This pattern is often used where the builder object is useful in its own right, rather than being just a builder. For example, see `std::process::Command` is a builder for `Child` (a process). In these cases, the `T` and `TBuilder` naming pattern is not used.

The example takes and returns the builder by value. It is often more ergonomic (and more efficient) to take and return the builder as a mutable reference. The borrow checker makes this work naturally. This approach has the advantage that one can write code like

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

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

See also

- Description in the style guide
- derive_builder, a crate for automatically implementing this pattern while avoiding the boilerplate.
- Constructor pattern for when construction is simpler.
- Builder pattern (wikipedia)
- Construction of complex values

Fold

Description

Run an algorithm over each item in a collection of data to create a new item, thus creating a whole new collection.

The etymology here is unclear to me. The terms ‘fold’ and ‘folder’ are used in the Rust compiler, although it appears to me to be more like a map than a fold in the usual sense. See the discussion below for more details.

Example

```
// 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 itself. In some cases, we can do
this
        // to inner nodes too.
        fn fold_name(&mut self, n: Box<Name>) -> Box<Name> { n }
        // Create a new inner node by folding its children.
        fn fold_stmt(&mut self, s: Box<Stmt>) -> Box<Stmt> {
            match *s {
                Stmt::Expr(e) => Box::new(Stmt::Expr(self.fold_expr(e))),
                Stmt::Let(n, e) => Box::new(Stmt::Let(self.fold_name(n),
self.fold_expr(e))),
            }
        }
        fn fold_expr(&mut self, e: Box<Expr>) -> Box<Expr> { ... }
    }
}

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

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

The result of running the `Renamer` on an AST is a new AST identical to the old one, but with every name changed to `foo`. A real life folder might have some state preserved between

nodes in the struct itself.

A folder can also be defined to map one data structure to a different (but usually similar) data structure. For example, we could fold an AST into a HIR tree (HIR stands for high-level intermediate representation).

Motivation

It is common to want to map a data structure by performing some operation on each node in the structure. For simple operations on simple data structures, this can be done using `Iterator::map`. For more complex operations, perhaps where earlier nodes can affect the operation on later nodes, or where iteration over the data structure is non-trivial, using the fold pattern is more appropriate.

Like the visitor pattern, the fold pattern allows us to separate traversal of a data structure from the operations performed to each node.

Discussion

Mapping data structures in this fashion is common in functional languages. In OO languages, it would be more common to mutate the data structure in place. The 'functional' approach is common in Rust, mostly due to the preference for immutability. Using fresh data structures, rather than mutating old ones, makes reasoning about the code easier in most circumstances.

The trade-off between efficiency and reusability can be tweaked by changing how nodes are accepted by the `fold_*` methods.

In the above example we operate on `Box` pointers. Since these own their data exclusively, the original copy of the data structure cannot be re-used. On the other hand if a node is not changed, reusing it is very efficient.

If we were to operate on borrowed references, the original data structure can be reused; however, a node must be cloned even if unchanged, which can be expensive.

Using a reference counted pointer gives the best of both worlds - we can reuse the original data structure, and we don't need to clone unchanged nodes. However, they are less ergonomic to use and mean that the data structures cannot be mutable.

See also

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

In other languages, fold is usually used in the sense of Rust's iterators, rather than this pattern. Some functional languages have powerful constructs for performing flexible maps over data structures.

The visitor pattern is closely related to fold. They share the concept of walking a data structure performing an operation on each node. However, the visitor does not create a new data structure nor consume the old one.

Structural Patterns

From Wikipedia:

> Design patterns that ease the design by identifying a simple way to realize relationships among entities.

Struct decomposition for independent borrowing

Description

Sometimes a large struct will cause issues with the borrow checker - although fields can be borrowed independently, sometimes the whole struct ends up being used at once, preventing other uses. A solution might be to decompose the struct into several smaller structs. Then compose these together into the original struct. Then each struct can be borrowed separately and have more flexible behaviour.

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

Example

Here is a contrived example of where the borrow checker foils us in our plan to use a struct:

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

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

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

    let connection_string = &mut db.connection_string;
    print_database(&db);  // Immutable borrow of `db` happens here
    // *connection_string = "new string".to_string();  // Mutable borrow is
used
                                                       // here
}
```

We can apply this design pattern and refactor `Database` into three smaller structs, thus solving the borrow checking issue:

```
// Database is now composed of three structs - ConnectionString, Timeout and
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 into `Database`
struct Database {
    connection_string: ConnectionString,
    timeout: Timeout,
    pool_size: PoolSize,
}

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

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

    let connection_string = &mut db.connection_string;
    print_database(connection_string.clone(), db.timeout, db.pool_size);
    *connection_string = ConnectionString("new string".to_string());
}
```

Motivation

This pattern is most useful, when you have a struct that ended up with a lot of fields that you want to borrow independently. Thus having a more flexible behaviour in the end.

Advantages

Decomposition of structs lets you work around limitations in the borrow checker. And it often produces a better design.

Disadvantages

It can lead to more verbose code. And sometimes, the smaller structs are not good abstractions, and so we end up with a worse design. That is probably a 'code smell', indicating that the program should be refactored in some way.

Discussion

This pattern is not required in languages that don't have a borrow checker, so in that sense is unique to Rust. However, making smaller units of functionality often leads to cleaner code: a widely acknowledged principle of software engineering, independent of the language.

This pattern relies on Rust's borrow checker to be able to borrow fields independently of each other. In the example, the borrow checker knows that `a.b` and `a.c` are distinct and can be borrowed independently, it does not try to borrow all of `a`, which would make this 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 libraries, much more so than in say C or C++. Moreover, since packages on crates.io cannot be edited or removed after publication, any build that works now should continue to work in the future. We should take advantage of this tooling, and use smaller, more fine-grained dependencies.

Advantages

- Small crates are easier to understand, and encourage more modular code.
- Crates allow for re-using code between projects. For example, the `url` crate was developed as part of the Servo browser engine, but has since found wide use outside the project.
- Since the compilation unit of Rust is the crate, splitting a project into multiple crates can allow more of the code to be built in parallel.

Disadvantages

- This can lead to "dependency hell", when a project depends on multiple conflicting versions of a crate at the same time. For example, the `url` crate has both versions 1.0 and 0.5. Since the `Url` from `url:1.0` and the `Url` from `url:0.5` are different types, an HTTP client that uses `url:0.5` would not accept `Url` values from a web scraper that uses `url:1.0`.
- Packages on crates.io are not curated. A crate may be poorly written, have unhelpful documentation, or be outright malicious.
- Two small crates may be less optimized than one large one, since the compiler does not perform link-time optimization (LTO) by default.

Examples

The `url` crate provides tools for working with URLs.

The `num_cpus` crate provides a function to query the number of CPUs on a machine.

The `ref_slice` crate provides functions for converting `&T` to `&[T]`. (Historical example)

See also

- crates.io: The Rust community crate host

Contain unsafety in small modules

Description

If you have `unsafe` code, create the smallest possible module that can uphold the needed invariants to build a minimal safe interface upon the unsafety. Embed this into a larger module that contains only safe code and presents an ergonomic interface. Note that the outer module can contain unsafe functions and methods that call directly into the unsafe code. Users may use this to gain speed benefits.

Advantages

- This restricts the unsafe code that must be audited
- Writing the outer module is much easier, since you can count on the guarantees of the inner module

Disadvantages

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

Examples

- The `toolshed` crate contains its unsafe operations in submodules, presenting a safe interface to users.
- `std`'s `String` class is a wrapper over `Vec<u8>` with the added invariant that the contents must be valid UTF-8. The operations on `String` ensure this behavior. However, users have the option of using an `unsafe` method to create a `String`, in which case the onus is on them to guarantee the validity of the contents.

See also

- Ralf Jung's Blog about invariants in unsafe code

FFI Patterns

Writing FFI code is an entire course in itself. However, there are several idioms here that can act as pointers, and avoid traps for inexperienced users of unsafe Rust.

This section contains design patterns that may be useful when doing FFI.

1. Object-Based API design that has good memory safety characteristics, and a clean boundary of what is safe and what is unsafe
2. Type Consolidation into Wrappers - group multiple Rust types together into an opaque “object”

Object-Based APIs

Description

When designing APIs in Rust which are exposed to other languages, there are some important design principles which are contrary to normal Rust API design:

1. All Encapsulated types should be *owned* by Rust, *managed* by the user, and *opaque*.
2. All Transactional data types should be *owned* by the user, and *transparent*.
3. All library behavior should be functions acting upon Encapsulated types.
4. All library behavior should be encapsulated into types not based on structure, but *provenance/lifetime*.

Motivation

Rust has built-in FFI support to other languages. It does this by providing a way for crate authors to provide C-compatible APIs through different ABIs (though that is unimportant to this practice).

Well-designed Rust FFI follows C API design principles, while compromising the design in Rust as little as possible. There are three goals with any foreign API:

1. Make it easy to use in the target language.
2. Avoid the API dictating internal unsafety on the Rust side as much as possible.
3. Keep the potential for memory unsafety and Rust `undefined behaviour` as small as possible.

Rust code must trust the memory safety of the foreign language beyond a certain point. However, every bit of `unsafe` code on the Rust side is an opportunity for bugs, or to exacerbate `undefined behaviour`.

For example, if a pointer provenance is wrong, that may be a segfault due to invalid memory access. But if it is manipulated by unsafe code, it could become full-blown heap corruption.

The Object-Based API design allows for writing shims that have good memory safety characteristics, and a clean boundary of what is safe and what is `unsafe`.

Code Example

The POSIX standard defines the API to access an on-file database, known as DBM. It is an excellent example of an "object-based" API.

Here is the definition in C, which hopefully should be easy to read for those involved in FFI. The commentary below should help explain it for those who miss the subtleties.

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

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 above. It is designed to contain internal state, and acts as an entry point for the library's behavior.

It is completely opaque to the user, who cannot create a `DBM` themselves since they don't know its size or layout. Instead, they must call `dbm_open`, and that only gives them *a pointer to one*.

This means all `DBM`s are "owned" by the library in a Rust sense. The internal state of unknown size is kept in memory controlled by the library, not the user. The user can only manage its life cycle with `open` and `close`, and perform operations on it with the other functions.

The `datum` type was called a "transactional" type above. It is designed to facilitate the exchange of information between the library and its user.

The database is designed to store "unstructured data", with no pre-defined length or meaning. As a result, the `datum` is the C equivalent of a Rust slice: a bunch of bytes, and a count of how many there are. The main difference is that there is no type information, which is what `void` indicates.

Keep in mind that this header is written from the library's point of view. The user likely has some type they are using, which has a known size. But the library does not care, and by the rules of C casting, any type behind a pointer can be cast to `void`.

As noted earlier, this type is *transparent* to the user. But also, this type is *owned* by the user. This has subtle ramifications, due to that pointer inside it. The question is, who owns the

memory that pointer points to?

The answer for best memory safety is, "the user". But in cases such as retrieving a value, the user does not know how to allocate it correctly (since they don't know how long the value is). In this case, the library code is expected to use the heap that the user has access to – such as the C library `malloc` and `free` – and then *transfer ownership* in the Rust sense.

This may all seem speculative, but this is what a pointer means in C. It means the same thing as Rust: "user defined lifetime." The user of the library needs to read the documentation in order to use it correctly. That said, there are some decisions that have fewer or greater consequences if users do it wrong. Minimizing those are what this best practice is about, and the key is to *transfer ownership of everything that is transparent*.

Advantages

This minimizes the number of memory safety guarantees the user must uphold to a relatively small number:

1. Do not call any function with a pointer not returned by `dbm_open` (invalid access or corruption).
2. Do not call any function on a pointer after close (use after free).
3. The `dptr` on any `datum` must be `NULL`, or point to a valid slice of memory at the advertised length.

In addition, it avoids a lot of pointer provenance issues. To understand why, let us consider an alternative in some depth: key iteration.

Rust is well known for its iterators. When implementing one, the programmer makes a separate type with a bounded lifetime to its owner, and implements the `Iterator` trait.

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

```
struct Dbm { ... }

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

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

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

This is clean, idiomatic, and safe. thanks to Rust's guarantees. However, consider what a straightforward API translation would look like:

```
#[no_mangle]
pub extern "C" fn dbm_iter_new(owner: *const Dbm) -> *mut DbmKeysIter {
    // THIS API IS A BAD IDEA! For real applications, use object-based design
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 applications, use object-based design
instead.
}
#[no_mangle]
pub extern "C" fn dbm_iter_del(*mut DbmKeysIter) {
    // THIS API IS A BAD IDEA! For real applications, use object-based design
instead.
}
```

This API loses a key piece of information: the lifetime of the iterator must not exceed the lifetime of the `Dbm` object that owns it. A user of the library could use it in a way which causes the iterator to outlive the data it is iterating on, resulting in reading uninitialized memory.

This example written in C contains a bug that will be explained afterwards:

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

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

    int l;
    while ((l = dbm_iter_next(owner, &key)) >= 0) { // an error is indicated 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 iterator returns the end-of-iteration marker:

1. The loop condition sets `l` to zero, and enters the loop because `0 >= 0`.
2. The length is incremented, in this case by zero.
3. The if statement is true, so the database is closed. There should be a break statement here.
4. The loop condition executes again, causing a `next` call on the closed object.

The worst part about this bug? If the Rust implementation was careful, this code will work most of the time! If the memory for the `Dbm` object is not immediately reused, an internal check will almost certainly fail, resulting in the iterator returning a `-1` indicating an error. But occasionally, it will cause a segmentation fault, or even worse, nonsensical memory corruption!

None of this can be avoided by Rust. From its perspective, it put those objects on its heap, returned pointers to them, and gave up control of their lifetimes. The C code simply must "play nice".

The programmer must read and understand the API documentation. While some consider that par for the course in C, a good API design can mitigate this risk. The POSIX API for `DBM` did this by *consolidating the ownership* of the iterator with its parent:

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

Thus, all the lifetimes were bound together, and such unsafety was prevented.

Disadvantages

However, this design choice also has a number of drawbacks, which should be considered as well.

First, the API itself becomes less expressive. With POSIX DBM, there is only one iterator per object, and every call changes its state. This is much more restrictive than iterators in almost any language, even though it is safe. Perhaps with other related objects, whose lifetimes are less hierarchical, this limitation is more of a cost than the safety.

Second, depending on the relationships of the API's parts, significant design effort may be involved. Many of the easier design points have other patterns associated with them:

- Wrapper Type Consolidation groups multiple Rust types together into an opaque "object"

- FFI Error Passing explains error handling with integer codes and sentinel return values (such as `NULL` pointers)
- Accepting Foreign Strings allows accepting strings with minimal unsafe code, and is easier to get right than Passing Strings to FFI

However, not every API can be done this way. It is up to the best judgement of the programmer as to who their audience is.

Type Consolidation into Wrappers

Description

This pattern is designed to allow gracefully handling multiple related types, while minimizing the surface area for memory unsafety.

One of the cornerstones of Rust's aliasing rules is lifetimes. This ensures that many patterns of access between types can be memory safe, data race safety included.

However, when Rust types are exported to other languages, they are usually transformed into pointers. In Rust, a pointer means "the user manages the lifetime of the pointee." It is their responsibility to avoid memory unsafety.

Some level of trust in the user code is thus required, notably around use-after-free which Rust can do nothing about. However, some API designs place higher burdens than others on the code written in the other language.

The lowest risk API is the "consolidated wrapper", where all possible interactions with an object are folded into a "wrapper type", while keeping the Rust API clean.

Code Example

To understand this, let us look at a classic example of an API to export: iteration through a collection.

That API looks like this:

1. The iterator is initialized with `first_key`.
2. Each call to `next_key` will advance the iterator.
3. Calls to `next_key` if the iterator is at the end will do nothing.
4. As noted above, the iterator is "wrapped into" the collection (unlike the native Rust API).

If the iterator implements `nth()` efficiently, then it is possible to make it ephemeral to each function call:

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

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

As a result, the wrapper is simple and contains no `unsafe` code.

Advantages

This makes APIs safer to use, avoiding issues with lifetimes between types. See Object-Based APIs for more on the advantages and pitfalls this avoids.

Disadvantages

Often, wrapping types is quite difficult, and sometimes a Rust API compromise would make things easier.

As an example, consider an iterator which does not efficiently implement `nth()`. It would definitely be worth putting in special logic to make the object handle iteration internally, or to support a different access pattern efficiently that only the Foreign Function API will use.

Trying to Wrap Iterators (and Failing)

To wrap any type of iterator into the API correctly, the wrapper would need to do what a C version of the code would do: erase the lifetime of the iterator, and manage it manually.

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 + 'self>
    iterator: Option<NonNull<KeysIter<'static>>>,
}
```

With `transmute` being used to extend a lifetime, and a pointer to hide it, it's ugly already. But it gets even worse: *any other operation can cause Rust `undefined behaviour`*.

Consider that the `MySet` in the wrapper could be manipulated by other functions during iteration, such as storing a new value to the key it was iterating over. The API doesn't discourage this, and in fact some similar C libraries expect it.

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 TO DEMONSTRATE A PROLBEM.

        let myset: &mut MySet = unsafe { // SAFETY: whoops, UB occurs in here!
            &mut (*myset).myset
        };

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

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

If the iterator exists when this function is called, we have violated one of Rust's aliasing rules. According to Rust, the mutable reference in this block must have *exclusive* access to the object. If the iterator simply exists, it's not exclusive, so we have `undefined behaviour`! [1]

To avoid this, we must have a way of ensuring that mutable reference really is exclusive. That basically means clearing out the iterator's shared reference while it exists, and then reconstructing it. In most cases, that will still be less efficient than the C version.

Some may ask: how can C do this more efficiently? The answer is, it cheats. Rust's aliasing rules are the problem, and C simply ignores them for its pointers. In exchange, it is common to see code that is declared in the manual as "not thread safe" under some or all circumstances. In fact, the GNU C library has an entire lexicon dedicated to concurrent behavior!

Rust would rather make everything memory safe all the time, for both safety and optimizations that C code cannot attain. Being denied access to certain shortcuts is the price Rust programmers need to pay.

[1] For the C programmers out there scratching their heads, the iterator need not be read *during* this code cause the UB. The exclusivity rule also enables compiler optimizations which may cause inconsistent observations by the iterator's shared reference (e.g. stack spills or reordering instructions for efficiency). These observations may happen *any time after* the mutable reference is created.

Anti-patterns

An anti-pattern is a solution to a “recurring problem that is usually ineffective and risks being highly counterproductive”. Just as valuable as knowing how to solve a problem, is knowing how *not* to solve it. Anti-patterns give us great counter-examples to consider relative to design patterns. Anti-patterns are not confined to code. For example, a process can be an anti-pattern, too.

Clone to satisfy the borrow checker

Description

The borrow checker prevents Rust users from developing otherwise unsafe code by ensuring that either: only one mutable reference exists, or potentially many but all immutable references exist. If the code written does not hold true to these conditions, this anti-pattern arises when the developer resolves the compiler error by cloning the 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, this line would fail on compile as
// x has been borrowed
// thanks to x.clone(), x was never borrowed, and this line will run.
println!("{}", x);

// perform some action on the borrow to prevent rust from optimizing this
//out of existence
*y += 1;
```

Motivation

It is tempting, particularly for beginners, to use this pattern to resolve confusing issues with the borrow checker. However, there are serious consequences. Using `.clone()` causes a copy of the data to be made. Any changes between the two are not synchronized – as if two completely separate variables exist.

There are special cases – `Rc<T>` is designed to handle clones intelligently. It internally manages exactly one copy of the data, and cloning it will only clone the reference.

There is also `Arc<T>` which provides shared ownership of a value of type T that is allocated in the heap. Invoking `.clone()` on `Arc` produces a new `Arc` instance, which points to the same allocation on the heap as the source `Arc`, while increasing a reference count.

In general, clones should be deliberate, with full understanding of the consequences. If a clone is used to make a borrow checker error disappear, that's a good indication this anti-pattern may be in use.

Even though `.clone()` is an indication of a bad pattern, sometimes **it is fine to write inefficient code**, in cases such as when:

- the developer is still new to ownership
- the code doesn't have great speed or memory constraints (like hackathon projects or prototypes)
- satisfying the borrow checker is really complicated, and you prefer to optimize readability over performance

If an unnecessary clone is suspected, The Rust Book's chapter on Ownership should be understood fully before assessing whether the clone is required or not.

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

See also

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

`#![deny(warnings)]`

Description

A well-intentioned crate author wants to ensure their code builds without warnings. So 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 crate author opts out of Rust's famed stability. Sometimes new features or old misfeatures need a change in how things are done, thus lints are written that `warn` for a certain grace period before being turned to `deny`.

For example, it was discovered that a type could have two `impl`s with the same method. This was deemed a bad idea, but in order to make the transition smooth, the `overlapping-inherent-impls` lint was introduced to give a warning to those stumbling on this fact, before it becomes a hard error in a future release.

Also sometimes APIs get deprecated, so their use will emit a warning where before there was none.

All this conspires to potentially break the build whenever something changes.

Furthermore, crates that supply additional lints (e.g. rust-clippy) can no longer be used unless the annotation is removed. This is mitigated with –cap-lints. The `--cap-lints=warn` command line argument, turns all `deny` lint errors into warnings.

Alternatives

There are two ways of tackling this problem: First, we can decouple the build setting from the code, and second, we can name the lints we want to deny explicitly.

The following command line will build with all warnings set to `deny`:

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

This can be done by any individual developer (or be set in a CI tool like Travis, but remember that this may break the build when something changes) without requiring a change to the code.

Alternatively, we can specify the lints that we want to `deny` in the code. Here is a list of warning lints that is (hopefully) safe to deny (as of Rustc 1.48.0):

```
#![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 good idea to `deny`:

```
#![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-implementations` to their list.

Note that we explicitly did not add the `deprecated` lint, as it is fairly certain that there will 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 system. Also type `rustc --help` for a general list of options
- rust-clippy is a collection of lints for better Rust code

`Deref` polymorphism

Description

Misuse the `Deref` trait to emulate inheritance between structs, and thus reuse methods.

Example

Sometimes we want to emulate the following common pattern from OO languages such 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 do so:

```
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 use composition and include an instance of `Foo` in `Bar` (since the field is a value, it is stored inline, so if there were fields, they would have the same layout in memory as the Java version (probably, you should use `#[repr(C)]` if you want to be sure)).

In order to make the method call work we implement `Deref` for `Bar` with `Foo` as the target (returning the embedded `Foo` field). That means that when we dereference a `Bar` (for example, using `*`) then we will get a `Foo`. That is pretty weird. Dereferencing usually gives a `T` from a reference to `T`, here we have two unrelated types. However, since the dot operator does implicit dereferencing, it means that the method call will search for methods on `Foo` as well as `Bar`.

Advantages

You save a little boilerplate, e.g.,

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

Disadvantages

Most importantly this is a surprising idiom - future programmers reading this in code will not expect this to happen. That's because we are misusing the `Deref` trait rather than using it as intended (and documented, etc.). It's also because the mechanism here is completely implicit.

This pattern does not introduce subtyping between `Foo` and `Bar` like inheritance in Java or C++ does. Furthermore, traits implemented by `Foo` are not automatically implemented for `Bar`, so this pattern interacts badly with bounds checking and thus generic programming.

Using this pattern gives subtly different semantics from most OO languages with regards to `self`. Usually it remains a reference to the sub-class, with this pattern it will be the 'class' where the method is defined.

Finally, this pattern only supports single inheritance, and has no notion of interfaces, class-based privacy, or other inheritance-related features. So, it gives an experience that will be subtly surprising to programmers used to Java inheritance, etc.

Discussion

There is no one good alternative. Depending on the exact circumstances it might be better to re-implement using traits or to write out the facade methods to dispatch to `Foo` manually. We do intend to add a mechanism for inheritance similar to this to Rust, but it is likely to be some time before it reaches stable Rust. See these blog posts and this RFC issue for more details.

The `Deref` trait is designed for the implementation of custom pointer types. The intention is that it will take a pointer-to-`T` to a `T`, not convert between different types. It is a shame that this isn't (probably cannot be) enforced by the trait definition.

Rust tries to strike a careful balance between explicit and implicit mechanisms, favouring explicit conversions between types. Automatic dereferencing in the dot operator is a case where the ergonomics strongly favour an implicit mechanism, but the intention is that this is limited to degrees of indirection, not conversion between arbitrary types.

See also

- Collections are smart pointers idiom.
- Delegation crates for less boilerplate like delegate or ambassador
- Documentation for `Deref` trait.

Functional Usage of Rust

Rust is an imperative language, but it follows many functional programming paradigms.

> In computer science, *functional programming* is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that each return a value, rather than a sequence of imperative statements which change the state of the program.

Programming paradigms

One of the biggest hurdles to understanding functional programs when coming from an imperative background is the shift in thinking. Imperative programs describe **how** to do something, whereas declarative programs describe **what** to do. Let's sum the numbers 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 compiler to see what is happening. Here, we start with a `sum` of `0`. Next, we iterate through the range from 1 to 10. Each time through the loop, we add the corresponding value in the range. Then we print it out.

`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 learn that a program is a set of steps.

Declarative

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

Whoa! This is really different! What's going on here? Remember that with declarative programs we are describing **what** to do, rather than **how** to do it. `fold` is a function that composes functions. The name is a convention from Haskell.

Here, we are composing functions of addition (this closure: `|a, b| a + b`) with a range from 1 to 10. The `0` is the starting point, so `a` is `0` at first. `b` is the first element of the range, `1`. `0 + 1 = 1` is the result. So now we `fold` again, with `a = 1`, `b = 2` and so `1 + 2 = 3` is the next result. This process continues until we get to the last element in the range, `10`.

`a`	`b`	**result**
0	1	1
1	2	3
3	3	6
6	4	10
10	5	15
15	6	21
21	7	28
28	8	36
36	9	45
45	10	55

Generics as Type Classes

Description

Rust's type system is designed more like functional languages (like Haskell) rather than imperative languages (like Java and C++). As a result, Rust can turn many kinds of programming problems into "static typing" problems. This is one of the biggest wins of choosing a functional language, and is critical to many of Rust's compile time guarantees.

A key part of this idea is the way generic types work. In C++ and Java, for example, generic types are a meta-programming construct for the compiler. `vector<int>` and `vector<char>` in C++ are just two different copies of the same boilerplate code for a `vector` type (known as a `template`) with two different types filled in.

In Rust, a generic type parameter creates what is known in functional languages as a "type class constraint", and each different parameter filled in by an end user *actually changes the type*. In other words, `Vec<isize>` and `Vec<char>` *are two different types*, which are recognized as distinct by all parts of the type system.

This is called **monomorphization**, where different types are created from **polymorphic** code. This special behavior requires `impl` blocks to specify generic parameters. Different values for the generic type cause different types, and different types can have different `impl` blocks.

In object-oriented languages, classes can inherit behavior from their parents. However, this allows the attachment of not only additional behavior to particular members of a type class, but extra behavior as well.

The nearest equivalent is the runtime polymorphism in Javascript and Python, where new members can be added to objects willy-nilly by any constructor. However, unlike those languages, all of Rust's additional methods can be type checked when they are used, because their generics are statically defined. That makes them more usable while remaining safe.

Example

Suppose you are designing a storage server for a series of lab machines. Because of the software involved, there are two different protocols you need to support: BOOTP (for PXE network boot), and NFS (for remote mount storage).

Your goal is to have one program, written in Rust, which can handle both of them. It will have protocol handlers and listen for both kinds of requests. The main application logic will then allow a lab administrator to configure storage and security controls for the actual files.

The requests from machines in the lab for files contain the same basic information, no matter what protocol they came from: an authentication method, and a file name to retrieve. A straightforward implementation would look something like this:

```
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 suppose you needed to support adding metadata that was *protocol specific*. For example, with NFS, you wanted to determine what their mount point was in order to enforce additional security rules.

The way the current struct is designed leaves the protocol decision until runtime. That means any method that applies to one protocol and not the other requires the 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 an NFS request. Otherwise,
    /// return None.
    pub fn mount_point(&self) -> Option<&Path> {
        self.mount_point.as_ref()
    }
}
```

Every caller of `mount_point()` must check for `None` and write code to handle it. This is true even if they know only NFS requests are ever used in a given code path!

It would be far more optimal to cause a compile-time error if the different request types were confused. After all, the entire path of the user's code, including what functions from the library they use, will know whether a request is an NFS request or a BOOTP request.

In Rust, this is actually possible! The solution is to *add a generic type* in order to split the API.

Here is what that looks like:

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

mod nfs {
    #[derive(Clone)]
    pub(crate) struct AuthInfo(String); // NFS session management omitted
}

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

// private module, lest outside users invent their own protocol kinds!
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) -> &Path {
            &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 metadata

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

use proto_trait::ProtoKind; // keep internal to prevent impls
pub use proto_trait::{Nfs, Bootp}; // re-export so callers can see them

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

```
// all common API parts go into a generic impl block
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 own block
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 mistake and use the wrong type;

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

They would get a syntax error. The type `FileDownloadRequest<Bootp>` does not implement `mount_point()`, only the type `FileDownloadRequest<Nfs>` does. And that is created by the NFS module, not the BOOTP module of course!

Advantages

First, it allows fields that are common to multiple states to be de-duplicated. By making the non-shared fields generic, they are implemented once.

Second, it makes the `impl` blocks easier to read, because they are broken down by state. Methods common to all states are typed once in one block, and methods unique to one state are in a separate block.

Both of these mean there are fewer lines of code, and they are better organized.

Disadvantages

This currently increases the size of the binary, due to the way monomorphization is implemented in the compiler. Hopefully the implementation will be able to improve in the future.

Alternatives

- If a type seems to need a "split API" due to construction or partial initialization, consider the Builder Pattern instead.
- If the API between types does not change – only the behavior does – then the 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 every other type of `Vec<T>`.[1]
- They can also be cast into a binary heap, but only if they contain a type that implements the `Ord` trait.[2]
- The `to_string` method was specialized for `Cow` only of type `str`.[3]

It is also used by several popular crates to allow API flexibility:

- The `embedded-hal` ecosystem used for embedded devices makes extensive use of this pattern. For example, it allows statically verifying the configuration of device registers used to control embedded pins. When a pin is put into a mode, it returns a `Pin<MODE>` struct, whose generic determines the functions usable in that mode, which are not on the `Pin` itself. [4]
- The `hyper` HTTP client library uses this to expose rich APIs for different pluggable requests. Clients with different connectors have different methods on them as well as different trait implementations, while a core set of methods apply to any connector. [5]
- The "type state" pattern – where an object gains and loses API based on an internal state or invariant – is implemented in Rust using the same basic concept, and a slightly different technique. [6]

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

[2] See: impl<T> From<Vec<T, Global>> for BinaryHeap<T>

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

[4] Example: https://docs.rs/stm32f30x-hal/0.1.0/stm32f30x_hal/gpio/gpioa/struct.PA0.html

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

[6] See: The Case for the Type State Pattern and Rusty Typestate Series (an extensive thesis)

Lenses and Prisms

This is a pure functional concept that is not frequently used in Rust. Nevertheless, exploring the concept may be helpful to understand other patterns in Rust APIs, such as visitors. They also have niche use cases.

Lenses: Uniform Access Across Types

A lens is a concept from functional programming languages that allows accessing parts of a data type in an abstract, unified way.[1] In basic concept, it is similar to the way Rust traits work with type erasure, but it has a bit more power and flexibility.

For example, suppose a bank contains several JSON formats for customer data. This is because they come from different databases or legacy systems. One database contains the data needed to perform credit checks:

```
{ "name": "Jane Doe",
  "dob": "2002-02-24",
  [...]
  "customer_id": 1048576332,
}
```

Another one contains the account information:

```
{ "customer_id": 1048576332,
  "accounts": [
      { "account_id": 2121,
        "account_type: "savings",
        "joint_customer_ids": [],
        [...]
      },
      { "account_id": 2122,
        "account_type: "checking",
        "joint_customer_ids": [1048576333],
        [...]
      },
  ]
}
```

Notice that both types have a customer ID number which corresponds to a person. How would a single function handle both records of different types?

In Rust, a `struct` could represent each of these types, and a trait would have a `get_customer_id` function they would implement:

```rust
use std::collections::HashSet;

pub struct Account {
    account_id: u32,
    account_type: String,
    // other fields omitted
}

pub trait CustomerId {
    fn get_customer_id(&self) -> u64;
}

pub struct CreditRecord {
    customer_id: u64,
    name: String,
    dob: String,
    // other fields omitted
}

impl CustomerId for CreditRecord {
    fn get_customer_id(&self) -> u64 {
        self.customer_id
    }
}

pub struct AccountRecord {
    customer_id: u64,
    accounts: Vec<Account>,
}

impl CustomerId for AccountRecord {
    fn get_customer_id(&self) -> u64 {
        self.customer_id
    }
}

// static polymorphism: only one type, but each function call can choose it
fn unique_ids_set<R: CustomerId>(records: &[R]) -> HashSet<u64> {
    records.iter().map(|r| r.get_customer_id()).collect()
}

// dynamic dispatch: iterates over any type with a customer ID, collecting all
// values together
fn unique_ids_iter<I>(iterator: I) -> HashSet<u64>
    where I: Iterator<Item=Box<dyn CustomerId>>
{
    iterator.map(|r| r.as_ref().get_customer_id()).collect()
}
```

Lenses, however, allow the code supporting customer ID to be moved from the *type* to the *accessor function*. Rather than implementing a trait on each type, all matching structures can simply be accessed the same way.

While the Rust language itself does not support this (type erasure is the preferred solution to this problem), the lens-rs crate allows code that feels like this to be written with macros:

```rust
use std::collections::HashSet;

use lens_rs::{optics, Lens, LensRef, Optics};

#[derive(Clone, Debug, Lens /* derive to allow lenses to work */)]
pub struct CreditRecord {
    #[optic(ref)] // macro attribute to allow viewing this field
    customer_id: u64,
    name: String,
    dob: String,
    // other fields omitted
}

#[derive(Clone, Debug)]
pub struct Account {
    account_id: u32,
    account_type: String,
    // other fields omitted
}

#[derive(Clone, Debug, Lens)]
pub struct AccountRecord {
    #[optic(ref)]
    customer_id: u64,
    accounts: Vec<Account>,
}

fn unique_ids_lens<T>(iter: impl Iterator<Item = T>) -> HashSet<u64>
where
    T: LensRef<Optics![customer_id], u64>, // any type with this field
{
    iter.map(|r| *r.view_ref(optics!(customer_id))).collect()
}
```

The version of `unique_ids_lens` shown here allows any type to be in the iterator, so long as it has an attribute called `customer_id` which can be accessed by the function. This is how most functional programming languages operate on lenses.

Rather than macros, they achieve this with a technique known as “currying”. That is, they “partially construct” the function, leaving the type of the final parameter (the value being operated on) unfilled until the function is called. Thus it can be called with different types dynamically even from one place in the code. That is what the `optics!` and `view_ref` in the example above simulates.

The functional approach need not be restricted to accessing members. More powerful lenses can be created which both *set* and *get* data in a structure. But the concept really becomes interesting when used as a building block for composition. That is where the concept appears more clearly in Rust.

Prisms: A Higher-Order form of "Optics"

A simple function such as `unique_ids_lens` above operates on a single lens. A *prism* is a function that operates on a *family* of lenses. It is one conceptual level higher, using lenses as a building block, and continuing the metaphor, is part of a family of "optics". It is the main one that is useful in understanding Rust APIs, so will be the focus here.

The same way that traits allow "lens-like" design with static polymorphism and dynamic dispatch, prism-like designs appear in Rust APIs which split problems into multiple associated types to be composed. A good example of this is the traits in the parsing crate *Serde*.

Trying to understand the way *Serde* works by only reading the API is a challenge, especially the first time. Consider the `Deserializer` trait, implemented by some type in any library which parses a new format:

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

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

    fn deserialize_bool<V>(self, visitor: V) -> Result<V::Value, Self::Error>
    where
        V: Visitor<'de>;

    // remainder ommitted
}
```

For a trait that is just supposed to parse data from a format and return a value, this looks odd.

Why are all the return types type erased?

To understand that, we need to keep the lens concept in mind and look at the definition of the `Visitor` type that is passed in generically:

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

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

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

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

    // remainder omitted
}
```

The job of the `Visitor` type is to construct values in the *Serde* data model, which are represented by its associated `Value` type.

These values represent parts of the Rust value being deserialized. If this fails, it returns an `Error` type - an error type determined by the `Deserializer` when its methods were called.

This highlights that `Deserializer` is similar to `CustomerId` from earlier, allowing any format parser which implements it to create `Value`s based on what it parsed. The `Value` trait is acting like a lens in functional programming languages.

But unlike the `CustomerId` trait, the return types of `Visitor` methods are *generic*, and the concrete `Value` type is *determined by the Visitor itself*.

Instead of acting as one lens, it effectively acts as a family of lenses, one for each concrete type of `Visitor`.

The `Deserializer` API is based on having a generic set of "lenses" work across a set of other generic types for "observation". It is a *prism*.

For example, consider the identity record from earlier but simplified:

```
{ "name": "Jane Doe",
  "customer_id": 1048576332,
}
```

How would the *Serde* library deserialize this JSON into `struct CreditRecord`?

1. The user would call a library function to deserialize the data. This would create a `Deserializer` based on the JSON format.
2. Based on the fields in the struct, a `Visitor` would be created (more on that in a moment) which knows how to create each type in a generic data model that was needed to represent it: `u64` and `String`.

3. The deserializer would make calls to the `Visitor` as it parsed items.
4. The `Visitor` would indicate if the items found were expected, and if not, raise an error to indicate deserialization has failed.

For our very simple structure above, the expected pattern would be:

1. Visit a map (*Serde*'s equvialent to `HashMap` or JSON's dictionary).
2. Visit a string key called "name".
3. Visit a string value, which will go into the `name` field.
4. Visit a string key called "customer_id".
5. Visit a string value, which will go into the `customer_id` field.
6. Visit the end of the map.

But what determines which "observation" pattern is expected?

A functional programming language would be able to use currying to create reflection of each type based on the type itself. Rust does not support that, so every single type would need to have its own code written based on its fields and their properties.

Serde solves this usability challenge with a derive macro:

```
use serde::Deserialize;

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

That macro simply generates an impl block causing the struct to implement a trait called `Deserialize`.

It is defined this way:

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

This is the function that determines how to create the struct itself. Code is generated based on the struct's fields. When the parsing library is called - in our example, a JSON parsing library - it creates a `Deserializer` and calls `Type::deserialize` with it as a parameter.

The `deserialize` code will then create a `Visitor` which will have its calls "refracted" by the `Deserializer`. If everything goes well, eventually that `Visitor` will construct a value corresponding to the type being parsed and return it.

For a complete example, see the *Serde* documentation.

To wrap up, this is the power of *Serde*:

1. The structure being parsed is represented by an `impl` block for `Deserialize`
2. The input data format (e.g. JSON) is represented by a `Deserializer` called by `Deserialize`
3. The `Deserializer` acts like a prism which "refracts" lens-like `Visitor` calls which actually build the data value

The result is that types to be deserialized only implement the "top layer" of the API, and file formats only need to implement the "bottom layer". Each piece can then "just work" with the rest of the ecosystem, since generic types will bridge them.

To emphasize, the only reason this model works on any format and any type is because the `Deserializer` trait's output type **is specified by the implementor of `Visitor` it is passed**, rather than being tied to one specific type. This was not true in the account example earlier.

Rust's generic-inspired type system can bring it close to these concepts and use their power, as shown in this API design. But it may also need procedural macros to create bridges for its generics.

See Also

- lens-rs crate for a pre-built lenses implementation, with a cleaner interface than these examples
- serde itself, which makes these concepts intuitive for end users (i.e. defining the structs) without needing to undestand the details
- luminance is a crate for drawing computer graphics that uses lens API design, including proceducal macros to create full prisms for buffers of different pixel types that remain generic
- An Article about Lenses in Scala that is very readable even without Scala expertise.
- Paper: Profunctor Optics: Modular Data Accessors

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

Additional resources

A collection of complementary helpful content

Talks

- Design Patterns in Rust by Nicholas Cameron at the PDRust (2016)
- Writing Idiomatic Libraries in Rust by Pascal Hertleif at RustFest (2017)
- Rust Programming Techniques by Nicholas Cameron at LinuxConfAu (2018)

Books (Online)

- The Rust API Guidelines

Design principles

A brief overview over common design principles

SOLID

- Single Responsibility Principle (SRP): A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.
- Open/Closed Principle (OCP): "Software entities ... should be open for extension, but closed for modification."
- Liskov Substitution Principle (LSP): "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
- Interface Segregation Principle (ISP): "Many client-specific interfaces are better than one general-purpose interface."
- Dependency Inversion Principle (DIP): One should "depend upon abstractions, [not] concretions."

DRY (Don't Repeat Yourself)

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"

KISS principle

most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided

Law of Demeter (LoD)

a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents), in accordance with the principle of "information hiding"

Design by contract (DbC)

software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants

Encapsulation

bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.

Command-Query-Separation(CQS)

"Functions should not produce abstract side effects...only commands (procedures) will be permitted to produce side effects." - Bertrand Meyer: Object-Oriented Software Construction

Principle of least astonishment (POLA)

a component of a system should behave in a way that most users will expect it to behave. The behavior should not astonish or surprise users

Linguistic-Modular-Units

"Modules must correspond to syntactic units in the language used." - Bertrand Meyer: Object-Oriented Software Construction

Self-Documentation

"The designer of a module should strive to make all information about the module part of the module itself." - Bertrand Meyer: Object-Oriented Software Construction

Uniform-Access

“All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.” - Bertrand Meyer: Object-Oriented Software Construction

Single-Choice

“Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.” - Bertrand Meyer: Object-Oriented Software Construction

Persistence-Closure

“Whenever a storage mechanism stores an object, it must store with it the dependents of that object. Whenever a retrieval mechanism retrieves a previously stored object, it must also retrieve any dependent of that object that has not yet been retrieved.” - Bertrand Meyer: Object-Oriented Software Construction