CIS 1905

Lecture 4: Traits and Generics

How Rust uses traits to make expressive APIs.

Logistics

- HW2 Patch
  - Save local changes (if any)
  - Reaccept invite
- HW3
  - Find teammates on Ed (or in person)

Last class(es) recap

- Ownership
  - Clone and copy
- Structs / Impls
  - Rust splits data and behavior

This class: working with generics and traits

Generics

Generics

We've actually already seen generics before:

```
let v: Vec<i32> = Vec::new();
```

`Vec<T>` where `T` represents a generic type such as `usize`, `CustomStruct`, or anything you want!

Two Types of Generics

Function

```
pub fn get_first<T>(list:
&[T]) -> Option<&T> {
    if list.is_empty() {
        None
    } else {
        Some(&list[0])
    }
}
```

Struct

```
pub struct Vec<T> {…}

impl<T> Vec<T> {
    pub fn insert(&mut self,
index: usize, element: T) {
        …
    }
}
```

Generics are everywhere

```
enum Option<T> {
    Some(T),
    None,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
Vec<T>
```

Traits (Motivation)

A Tale of Two Functions

A physics simulation that uses single-precision floats.

```
struct Vec3
{
  x: f32,
  y: f32,
  z: f32
}
```

```
fn dot_product(a: Vec3, b: Vec3) -> f32
{
  (a.x * b.x) +
  (a.y * b.y) +
  (a.z * b.z)
}
```

A Tale of Two Functions

A setting that changes the physics to use double-precision.

```
struct Vec3
{
  x: f64,
  y: f64,
  z: f64
}
```

```
fn dot_product(a: Vec3, b: Vec3) -> f64
{
  (a.x * b.x) +
  (a.y * b.y) +
  (a.z * b.z)
}
```

A Tale of Two Functions

Addition in two places.

```
fn add(f32, f32) -> f32
```

```
fn add(f64, f64) -> f64
```

```
fn add(T, T) -> T
```

A Tale of Two Functions

Addition in two places.

```
fn add(f32, f32) -> f32
```

```
fn add(f64, f64) -> f64
```

```
fn add(T, T) -> T
```

What is T?

A Tale of Two Functions

Addition in two places.

```
fn add(f32, f32) -> f32
```

```
fn add(f64, f64) -> f64
```

```
fn add(T, T) -> T
```

What is **T**?

T is a stand-in for whatever type can go here.

A Tale of Two Functions

Multiplication in two places.

```
fn mul(f32, f32) -> f32
```

```
fn mul(f64, f64) -> f64
```

```
fn mul(T, T) -> T
```

A Tale of Two Functions

Multiplication in two places.

```
fn mul(f3
```

What’s wrong with this?

```
) -> f64
```

```
fn mul(T, T) -> T
```

T requires certain properties

How do you multiply two booleans? Strings? Custom types?

Traits

Trait Definitions

Introducing Traits!

- Trait name
- Required function signature

"self" means the data that we call a method on
"Self" means the type that the trait is being implemented for (or the type of an impl block generally)

```
pub trait Add {
  fn add(self, rhs: Self) -> Self;
}
pub trait Mul {
  fn mul(self, rhs: Self) -> Self;
}
```

Trait Implementations

Making them usable

- Trait name
- Required function signature

"self" means the data that we call a method on
"Self" means the type that the trait is being implemented for (or the type of an impl block generally)

```
impl Add for f32 {
  fn add(self, rhs: Self) -> Self {
    self + rhs
  }
}
```

Trait Usage

Instead of writing the same function body many times, we write what we want to *do* with the data, and then we write the application so that it can use any of the options we choose.

```
struct Vec3<T>
{
  x: T,
  y: T,
  z: T
}
```

```
fn dot_product<T>(a: Vec3<T>, b: Vec3<T>) -> T
where T: Add + Mul
{
  let x = a.x.mul(b.x);
  let y = a.y.mul(b.y);
  let z = a.z.mul(b.z);
  x.add(y).add(z)
}
```

Trait Usage

Here’s what it might look like if you chained the method calls.

```
struct Vec3<T>
{
    x: T,
    y: T,
    z: T
}
```

```
fn dot_product<T>(a: Vec3<T>, b: Vec3<T>) -> T
where T: Add + Mul
{
    (a.x.mul(b.x))
        .add(
            a.y.mul(b.y)
        ).add(
            a.z.mul(b.z)
        )
}
```

The Real Add and Mul Traits

Generics (+defaults)

Associated Types

```
pub trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}

pub trait Mul <Rhs = Self> {
    type Output;
    fn mul(self, rhs: Rhs) -> Self::Output;
}
```

The Real Add and Mul Traits

Generics (+defaults)

Associated Types

```
fn dot_product<T>(a: Vec3<T>, b: Vec3<T>) -> T
where T: Add<T, Output=T> + Mul<Output=T>
{
  (a.x * b.x) +
  (a.y * b.y) +
  (a.z * b.z)
}
```

The Real Add and Mul Traits

Generics (+defaults)

Associated Types

```
fn dot_product<T>(a: Vec3<T>, b: Vec3<T>) -> T
where T: Add<T, Output=T> + Mul<Output=T>
{
   (a.x * b.x) +
   (a.y * b.y) +
   (a.z * b.z)
}
```

Where T implements

The Real Add and Mul Traits

Generics (+defaults)

Associated Types

```
fn dot_product<T>(a: Vec3<T>, b: Vec3<T>) -> T
where T: Add<T, Output=T> + Mul<Output=T>
{
   (a.x * b.x) +
   (a.y * b.y) +
   (a.z * b.z)
}
```

= for assoc type bounds

The Real Add and Mul Traits

Generics (+defaults)

Associated Types

```
fn dot_product<T>(a: Vec3<T>, b: Vec3<T>) -> T
where T: Add<T, Output=T> + Mul<Output=T>
{
  (a.x * b.x) +
  (a.y * b.y) +
  (a.z * b.z)
}
```

+ For multiple bounds

LOUDER CROWDER
LOUDER CROWDER
"Traits are interfaces."
CHANGE MY MIND

The End

HW2 due on February 18th

Traits

- Default method implementations
- Associated types and constants
- Can implement traits for types that are “not yours”

So What?

Replacing two functions with one is nice, but it's nothing special. Other languages do this with things like templates, interfaces, or inheritance. There are limits to those approaches, though.

Templates (C++ before C++ 20)

- separate language than the generated code
- doesn't communicate requirements
- makes errors where fault is unclear
- and that's why Concepts are in C++ 20

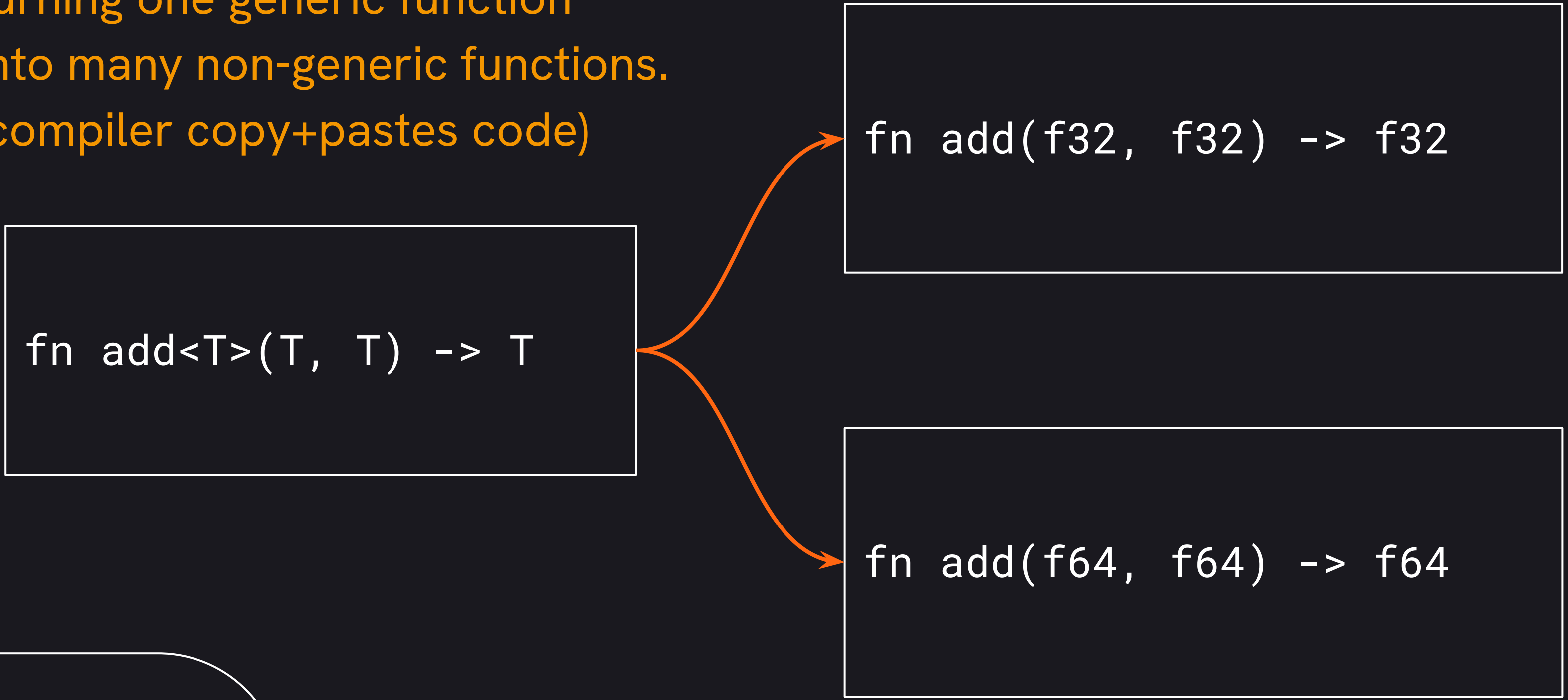
Inheritance (or interfaces)

- runtime cost (dynamic dispatch*)
- doesn't communicate requirements (such as associated types and some generics)
- conflicts between extensibility and manageable code is (see the SOLID principles)

What is Static Dispatch?

Monomorphization:

turning one generic function
into many non-generic functions.
(compiler copy+pastes code)



Dynamic Dispatch through Trait Objects

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // other methods including filter
}
```

```
fn chain_iter(iter: Box<dyn Iterator<Item=i32>>, use_evens: bool)->
Box<dyn Iterator<Item=i32>> {
  if use_evens {
    Box::new(iter.filter(|x| x % 2 == 0)) // this expression
  } else {
    iter // and this one can have the same type
  } // because the return point requires a trait object
}
```

Dynamic Dispatch through Trait Objects

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // other methods including filter
}
```

```
fn chain_iter(iter: Box<dyn Iterator<Item=i32>>, use_evens: bool)->
Box<dyn Iterator<Item=i32>> {
  if use_evens {
    Box::new(iter.filter(|x| x % 2 == 0)) // this expression
  } else {
    iter // and this one can have the same type
  } // because the return point requires a trait object
}
```

Closures provide anonymous functions

What is Box<dyn Iterator<Item=i32>>?

- Box<T>: a pointer to a heap allocated T
- dyn: use ***dynamic dispatch*** to make a trait object
- Iterator: a trait
- <Item=i32>: specifying that the associated type Item on the Iterator trait must be an i32

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // other methods
}
```

What is Dynamic Dispatch?

In order to have one function that works on many types without generating different functions, we have to get the information on how to handle those types from somewhere. Language-level support will build a **vtable** (virtual table) to store that info.

Function name	Pointer address
function_a	0xAAAA_BBBB
function_b	0xAAAA_BBD0
Drop::drop	0xAAAA_0000

Static Dispatch through impl Trait

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // other methods including filter
}
```

```
fn chain_iter(iter: impl Iterator<Item=i32>, use_evens: bool)->
impl Iterator<Item=i32> {
  if use_evens {
    iter.filter(|x| x % 2 == 0)) // this expression
  } else {
    iter // and this one don't have the same type
  } // and so they can't use the static dispatch here
}
```

Static Dispatch through impl Trait

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> O
    // other methods includ
}
```

impl Trait (param) means the **function** must be able to handle any type implementing Trait (like with generics!)

impl Trait (param)

```
fn chain_iter(iter: impl Iterator<Item=i32>, use_evens: bool)->
impl Iterator<Item=i32> {
  if use_evens {
    iter.filter(|x| x % 2 == 0)) // this expression
  } else {
    iter // and this one don't have the same type
  } // and so they can't use the static dispatch here
}
```

Static Dispatch through impl Trait

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -
    // other methods inc
}
```

impl Trait (return) means the ***caller*** must be able to handle any type implementing Trait

There is ***no*** equivalent to this with generics

impl Trait (return)

```
fn chain_iter(i   : impl Iterator<Item=i32>, use_evens: bool)->
impl Iterator<Item=i32> {
  if use_evens {
    iter.filter(|x| x % 2 == 0)) // this expression
  } else {
    iter // and this one don't have the same type
  } // and so they can't use the static dispatch here
}
```

Static Dispatch through impl Trait

```
enum Either<L, R> {
    Left(L),
    Right(R)
}
// and implement Iterator for Either
```

```
fn chain_iter(iter: impl Iterator<Item=i32>, use_evens: bool)->
impl Iterator<Item=i32> {
  if use_evens {
    Either::Left(iter.filter(|x| x % 2 == 0))) // this expression
  } else {
    Either::Right(iter) // and this one do have the same type
  } // so they can use the static dispatch here
}
```

Dynamic vs Static Dispatch

Traits allow both!

Dynamic

- Small code size
- Slower on modern computers

Both

- Compile time guarantees about what methods exist
- Clear boundary of responsibility

Static

- Many generated functions → larger code size
- Each individual function can be optimized with context

derive Macro

#[derive([Trait])]

#[derive(...)] is just magic that calls a macro to generate some code
to implement your trait automatically in some canonical or "obvious" way.
If you want something less obvious or different than the default,
you can write that code yourself.

Example usage:

```
#[derive(Clone, Debug)]
pub struct Person {
    name: String,
    age: u8,
    phone: [u8; 10], // fixed array of 10 digits
    favorite_color: String,
}
```

How does derive work?

When can you derive?

How do you make a trait derivable?

How does derive work?

When can you derive?

- Data type declaration

How do you make a trait derivable?

- Procedural (or “proc”) macros go in their own crate
- Write a function that takes Rust code tokens as input
- Output the code that you *would* write by hand, except programmatically replace pieces with information from the input
    - Loop over fields of a struct
    - Loop over variants of an enum
    - Grab and insert names of types and trait bounds

What does derive(Trait) do?

Derive macros generate the most straightforward (according to the macro author. Here we use Vec2 to save space.

```
#[derive(Debug)]
struct Vec2 {
  x: f64,
  y: f64
}
```

```
impl Debug for Vec2 {
  fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error> {
    f.write_str("Vec3 {");
    f.write_str("x: ");
    f64::fmt(&self.x, f);
    f.write_str(", ");
    f.write_str("y: ");
    f64::fmt(&self.y, f);
    f.write_str("}");
  }
}
```

What does derive(Trait) do?

Derive macros generate the most straightforward (according to the macro author. Here we use Vec2 to save space.

```
#[derive(Debug)]
struct Vec2 {
  x: f64,
  y: f64
}
```

```
impl Debug for Vec2 {
  fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error> {
    f.write_str("Vec2 {");
    f.write_str("x: ");
    f64::fmt(&self.x, f);
    f.write_str(", ");
    f.write_str("y: ");
    f64::fmt(&self.y, f);
    f.write_str("}");
  }
}
```

Looks like a loop over the fields!

Must-know Traits

From/Into

Convert between types without any chance of failure.

By convention, these impls never panic.

If you implement From<T> for U, then you also get Into<U> for T for free!

```
fn conv_addable<T1, T2, T3>(a: T1, b: T2) -> <T3 as Add>::Output
where
  T1: Into<T3>,
  T2: Into<T3>,
  T3: Add,
{
  a.into() + b.into()
}
```

How is it “for free?”

We call an impl that’s generic over types that satisfy certain trait bounds a “blanket” impl. Blanket impls work like other generics, where they copy+paste each time a generic instance used. There can usually only be one because they can conflict.

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

TryFrom/TryInto

Convert between types, but it may fail.

Just like From/Into,

if you implement TryFrom<T> for U, then you get TryInto<U> for T for free!

```
fn try_conv_addable<T>(a: T, b: f32) -> Result<f32, Box<dyn Error>>
where
  T1: TryInto<f32>,
  <T1 as TryInto>::Error: std::error::Error
{
  a.try_into()? + b
}
```

Fn Traits

Sometimes you want the user to be able to pass some behavior that you're generic over. It's nice to be able to pass a function as an argument and call it using the same syntax as a normal function call.

```rust
fn map_option<T, F, U>(opt: Option<T>, f: F) -> Option<U>
where
  F: FnOnce(T) -> U
{
  match opt {
    Some(value) => Some(f(value)),
    None => None
  }
}
```

Fn Traits explained

Closures, function items, and function pointers need to be callable.

The traits that govern this are:

- FnOnce: This function-like can be called at least once. This is implemented by all closures that compile.
- FnMut: This function-like can be called multiple times, but it *may* need exclusive permission to its state.
- Fn: This function-like can be called multiple times, and it only needs shared permissions to its state.

These traits get nice syntax based on their parameters and return type.

```
F: Fn(param1, param2) -> ret
```

This mirrors the syntax for declaring and using function items and pointers.

```
fn(param1, param2) -> ret
```

Fn Traits example revisited

Sometimes you want the user to be able to pass some behavior that you're generic over. It's nice to be able to pass a function as an argument and call it using the same syntax as a normal function call.

```
fn map_option<T, F, U>(opt: Option<T>, f: F) -> Option<U>
where
  F: FnOnce(T) -> U
{
  match opt {
    Some(value) => Some(f(value)),
    None => None
  }
}
```

Default

Provide a default value for a type. This helps in case an operation would fail.

- 0 for numbers
- "" for strings
- [ ] for list-like collections

```
fn unwrap_or_default<T>(x: Option<T>) -> T
where
  T: Default
{
  match opt {
    Some(value) => value,
    None => T::default()
  }
}
```

Clone

Make a meaningful duplicate of data.

For Vec, String, Hashmap, etc. this is a (relatively) expensive deep copy.

Generic data structures have Clone requirements on T in order to be Clone.

```
fn square_mul<T>(a: T) -> T
where
  T: Clone + Mul<T, Output=T>
{
  a.clone() * a
}
```

Copy

A special kind of Clone that says a clone can be made by bitwise copying.

For Vec, String, Hashmap, etc. *cannot* be Copy because it's incorrect.

Integers, floats, immutable references, etc. all impl Copy.

```
fn square_mul<T>(a: T) -> T
where
  T: Copy + Mul<T, Output=T>
{
  a * a
}
```

It looks like we gave ownership twice!

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

Copy

A special kind of Clone that says a clone can be made by bitwise copying.

For Vec, String, Hashmap, etc. *cannot* be Copy because it's incorrect.

Integers, floats, immutable references, etc. all impl Copy.

```
fn square_mul<T>(a: T) -> T
where
  T: Copy + Mul<T, Output=T>
{
  a * a
}
```

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

Declare a supertrait to enforce bounds

Supertraits

Sometimes a trait needs all the functionality of another trait in order to work. Instead of making all the methods over again and forcing implementers to do that, we can add bounds or use a “supertrait” to enforce that implementers must already implement the supertrait.

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

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

Marker Traits (std::marker)

Marker traits are used to mark types as having some property to the compiler.

Sized, Copy, Send

```
#[derive(Clone)]
struct Foo {
    a: i32,
    b: f32,
    c: bool,
}

impl Copy for Foo {}
```

Aside: `PhantomData` from `std::marker`

The rust compiler is flawed!!!!
You are required to *use* any generic type parameters you introduce.

What if you don’t want to use them?

`std::marker::PhantomData`

```rust
// If you get rid of the `PhantomData`, it won't compile
pub struct UnimplementedStructFromTheHomework<T> {
    _delete_me: PhantomData<T>,
}
```

Extern Crate: serde for easy serialization

Traits that let you specify how to use most formats

- JSON
- YAML
- XML
- Bincode
- RON (Rusty Object Notation)

Macros that derive those traits

Extern Crate: serde for

Using Python’s Pickle Library

The `pickle` module is part of the Python standard library and implements methods to serialize (pickling) and deserialize (unpickling) Python objects.

To get started with `pickle`, import it in Python:

```python
import pickle
```

Afterward, to serialize a Python object such as a dictionary and store the byte stream as a file, we car
pickle’s `dump()` method.

```python
test_dict = {"Hello": "World!"}
with open("test.pickle", "wb") as outfile:
  # "wb" argument opens the file in binary mode
  pickle.dump(test_dict, outfile)
```

The byte stream representing `test_dict` is now stored in the file “`test.pickle`”!

To recover the original object, we read the serialized byte stream from the file using pickle’s `load()` r

```python
with open("test.pickle", "rb") as infile:
  test_dict_reconstructed = pickle.load(infile)
```

Warning: Only unpickle data from sources you trust, as it is possible for arbitrary malicious code to b
executed during the unpickling process.

Putting them together, the following code helps you to verify that pickle can recover the same object:

```python
import pickle

# A test object
test_dict = {"Hello": "World!"}

# Serialization
with open("test.pickle", "wb") as outfile:
    pickle.dump(test_dict, outfile)
print("Written object", test_dict)

# Deserialization
with open("test.pickle", "rb") as infile:
    test_dict_reconstructed = pickle.load(infile)
print("Reconstructed object", test_dict_reconstructed)
```

Besides writing the serialized object intc
array type in Python using pickle’s dump

```python
test_dict_ba = pickle.dumps(test_d
```

Similarly, we can use pickle’s load meth

```python
test_dict_reconstructed_ba = pickl
```

FEATURES
Current ISO C++ status
Upcoming ISO C++ meetings
Upcoming C++ conferences
Compiler conformance status

NAVIGATION
FAQ Home
FAQ RSS Feed
FAQ Help

SEARCH THIS WIKI

GO TO PAGE

UPCOMING EVENTS
C++ (on Sea) Online
February 7-9
ISO C++ committee meeting
March 18-23, Tokyo, Japan
ACCU 2024
April 17-20, Bristol, UK
using std::cpp 2024
April 25-26, Leganes, Spain
C++ Now 2024
May 7-12, Aspen, CO, USA
ISO C++ committee meeting
June 24-29, St. Louis, MO, USA
C++ on Sea
July 2-5, Folkestone, Kent, UK

Wiki Home > Serialization and Unserialization

Serialization and Unserialization

Contents of this section:

- What’s this “serialization” thing all about?
- How do I select the best serialization technique?
- How do I decide whether to serialize to human-readable (“text”) c
- How do I serialize/unserialize simple types like numbers, charact
- How exactly do I read/write simple types in human-readable (“te
- How exactly do I read/write simple types in non-human-readable
- How do I serialize objects that aren’t part of an inheritance hierar
- How do I serialize objects that are part of an inheritance hierarch
- How do I serialize objects that contain pointers to other objects, b
- How do I serialize objects that contain pointers to other objects, b
- How do I serialize objects that contain pointers to other objects, a
- Are there any caveats when serializing / unserializing objects?
- What’s all this about graphs, trees, nodes, cycles, joins, and joins a

FAQ What’s this “serialization” thing all about?

It lets you take an object or group of objects, put them on a disk or s

Like the Transporter on Star Trek, it’s all about taking something co

FAQ How do I select the best serialization techniq

There are lots and lots (and lots) of if’s, and’s and but’s, and in reality

You are, of course, not limited to those five techniques. You will pro

There’s a lot here, so get ready!

1. Decide between human-readable (“text”) and non-human-readabl
2. Use the least sophisticated solution when the objects to be seriali
3. Use the second level of sophistication when the objects to be seri
4. Use the third level of sophistication when the objects to be serialized contain pointers to other objects, but when those pointers form a tree with no cycles and no joins.
5. Use the fourth level of sophistication when the objects to be serialized contain pointers to other objects, and when those pointers form a graph with no cycles, and with joins at the leaves only.
6. Use the most sophisticated solution when the objects to be serialized contain pointers to other objects, and when those pointers form a graph that might have cycles or joins.

Here’s that same information arranged like an algorithm:

1. The first step is to make an eyes-open decision between text- and binary-formats.
2. If your objects aren’t part of an inheritance hierarchy and don’t contain pointers, use solution #1.
3. Else if your objects don’t contain pointers to other objects, use solution #2.
4. Else if the graph of pointers within your objects contain neither cycles nor joins, use solution #3.
5. Else if the graph of pointers within your objects don’t contain cycles and if the only joins are to terminal (leaf) nodes, use solution #4.
6. Else use solution #5.

Remember: feel free to mix and match, to add to the above list, and, *if* you can justify the added expense, to use a more sophisticated technique than is minimally required.

One more thing: the issues of inheritance and of pointers within the objects are logically unrelated, so there’s no theoretical reason for #2 to be any less sophisticated than #3-5. However in practice it often (not always) works out that way. So please do not think of these categories as somehow sacred — they’re somewhat arbitrary, and yo

FAQ How do I decide whether to serialize to human-readable (“text”) or non-human-readable (“binary”) format?

Carefully.

There is no “right” answer to this question; it really depends on your goals. Here are a few of the pros/cons of human-readable (“text”) format vs. non-human-readable (“binary”) format:

- Text format is easier to “desk check.” That means you won’t have to write extra tools to debug the input and output; you can open the serialized output with a text editor to see if it looks right.
- Binary format typically uses fewer CPU cycles. However that is relevant only if your application is CPU bound and you intend to do serialization and/or unserialization on an inner loop/bottleneck. Remember: 90% of the CPU time is spent in 10% of the code, which means there won’t be any practical performance benefit unless your
- Text format lets you ignore programming issues like `sizeof` and little-endian vs. big-endian.

Java

```java
// Java code for serialization and deserialization
// of a Java object
import java.io.*;

class Demo implements java.io.Serializable
{
    public int a;
    public String b;

    // Default constructor
    public Demo(int a, String b)
    {
        this.a = a;
        this.b = b;
    }

}

class Test
{
    public static void main(String[] args)
    {
        Demo object = new Demo(1, "geeksforgeeks");
        String filename = "file.ser";

        // Serialization
        try
        {
            //Saving of object in a file
            FileOutputStream file = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(file);

            // Method for serialization of object
            out.writeObject(object);

            out.close();
            file.close();

            System.out.println("Object has been serialized");

        }

        catch(IOException ex)
```

Extern Crate: serde for easy serialization

```
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let point = Point { x: 1, y: 2 };
    let serialized = serde_json::to_string(&point).unwrap();

    println!("serialized = {}", serialized); // {"x":1,"y":2}
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    println!("deserialized = {:?}", deserialized); // Point { x: 1, y: 2 }

}
```

Library – num for numerical operations

Traits that encompass

- Normal arithmetic operators
- Checked, Wrapping, Saturating arithmetic
- All Integers, Floats, Numbers and their operations

Library - serde for easy serialization

Traits that let you specify how to use most formats

- JSON
- YAML
- XML
- Bincode
- RON (Rusty Object Notation)

Macros that derive those traits

Applications

Callbacks (GUI handlers, HTTP routes)

User-defined functions (interpreters)

Dynamically loaded libraries (plugins)

Some applications requires dynamic allocation + dispatch to handle different cases when connecting dynamic inputs

Traits: Live Coding

github.com/cis1905/snippets/tree/main/lec4

PollEv

pollev.com/cis1905rust915

Takeaways

Generics

- Monomorphization

Traits

- Derive