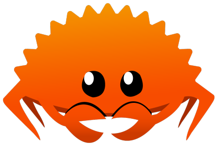
traits & generics

Lukas Prokop

February 27, 2020

RustGraz community



Prologue

std::mem::size_of_val of union

Last session: One `unsafe` superpower is “Access fields of unions”.
Claim: The size of a union is the size of its largest member.

```
pub const fn size_of<T>() -> usize
```

Returns the size of a type in bytes.

More specifically, this is the offset in bytes between successive elements in an array with that item type including alignment padding. Thus, for any type T and length n, `[T; n]` has a size of `n * size_of::<T>()`.

In general, the size of a type is not stable across compilations, but specific types such as primitives are.

→ via rust doc

std::mem::size_of_val of union

Reminder: a slide from December:

```
use std::mem;

fn main() {
  println!("{}", mem::size_of::<A>());
}
```

where A is `u8` (1), `u32` (4), `f64` (8), `&u8` (8), `String` (24), `&str` (16), `Vec<u8>` (24) or `&[char]` (16).

std::mem::size_of_val of union

Size of `#[repr(C)]` **items**

The C representation for items has a defined layout. With this layout, the size of items is also stable as long as all fields have a stable size.

Size of Unions

The size of a union is the size of its largest field.

Unlike C, zero sized unions are not rounded up to one byte in size.

→ via rust doc

std::mem::size_of_val of union

```rust
use std::mem::size_of;

#[repr(C)]
union Ambiguous {
  my_int: u32,
  addr: *const u32,
}

fn main() {
  println!("{}", size_of::<Ambiguous>());
  // ⇒ 8
}
```

alignment of struct

```
pub const fn align_of<T>() -> usize
```

Returns the ABI-required minimum alignment of a type.

Every reference to a value of the type T must be a multiple of this number.

This is the alignment used for struct fields. It may be smaller than the preferred alignment.

→ via rust doc

alignment of struct

Size of Structs

For structs, the size is determined by the following algorithm.

For each field in the struct ordered by declaration order:

1. Add the size of the field.
2. Round up the current size to the nearest multiple of the next field's alignment.

Finally, round the size of the struct to the nearest multiple of its alignment. The alignment of the struct is usually the largest alignment of all its fields; this can be changed with the use of `repr(align(N))` .

Unlike C, zero sized structs are not rounded up to one byte in size.

alignment of struct

```rust
use std::mem::{size_of,align_of};

fn main() {
  println!("{} => {}", size_of::<i32>(),
           align_of::<i32>());       // 4 => 4
  println!("{} => {}", size_of::<u8>(),
           align_of::<u8>());        // 1 => 1
  println!("{} => {}", size_of::<*mut i32>(),
           align_of::<*mut i32>());  // 8 => 8
  println!("{} => {}", size_of::<usize>(),
           align_of::<usize>());     // 8 => 8
  println!("{} => {}", size_of::<char>(),
           align_of::<char>());      // 4 => 4
}
```

alignment of struct

```
struct Data {
  a: i32,
  b: u8,
  c: *mut i32,
  d: usize,
  e: char,
}

struct SomeU8 {
  a: u8,
  b: u8,
}
```

alignment of struct

```
struct SomeI32 {
  a: i32,
  b: i32,
}

fn main() {
  println!("{} ⇒ {}", size_of::<Data>(),
           align_of::<Data>());    // 32 ⇒ 8
  println!("{} ⇒ {}", size_of::<SomeU8>(),
           align_of::<SomeU8>());  // 2 ⇒ 1
  println!("{} ⇒ {}", size_of::<SomeI32>(),
           align_of::<SomeI32>()); // 8 ⇒ 4
}
```

Dialogue

Goal for today

- Understanding how traits work
- Understanding how generics work
- Understanding a few type-system related properties/patterns
- Understanding how rust models usecases covered by OOP, FP, concepts, contracts, interfaces, … $\Rightarrow$ discuss type systems

Dialogue: traits

struct

```
struct Talk {
    desc: String,
    duration: u16,
}
```

struct

```
impl Talk {
  fn new(text: &str) -> Talk {
    Talk {
      desc: text.to_string(),
      duration: 45,
    }
  }
}
```

A method new implemented over struct type Talk .

struct - *not* like this!

```
fn Talk::new() -> Talk {
    Talk { desc: "foo", duration: 42 }
}

error: expected one of `(` or `<`, found `::`

impl Talk {
    fn new() -> Talk {
        Talk { /*…*/ }
    }
    fn new() -> Talk {
        Talk { /*…*/ }
    }
}

error: duplicate definitions with name `new`
```

traits

```
trait Submission {
    fn len(&self) -> u32;
    fn summary(&self) -> String;
}
```

traits cannot be instantiated

```rust
fn main() {
    let s: Submission;
}
```

```
warning: trait objects without an explicit `dyn` are deprecated
  --> src/main.rs:16:12
   |
16 |     let s: Submission;
   |            ^^^^^^^^^^ help: use `dyn`: `dyn Submission`
   |
   = note: `#[warn(bare_trait_objects)]` on by default

error[E0277]: the size for values of type `dyn Submission` cannot be known at compilation tim
  --> src/main.rs:16:9
   |
16 |     let s: Submission;
   |         ^ doesn't have a size known at compile-time
   |
   = help: the trait `std::marker::Sized` is not implemented for `dyn Submission`
   = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-types.html#dy
   = note: all local variables must have a statically known size
   = help: unsized locals are gated as an unstable feature
```

traits

```
impl Submission for Talk {
  fn len(&self) -> u32 {
    self.duration as u32
  }
  fn summary(&self) -> String {
    let dot = self.desc.find('.');
    match dot {
      Some(idx) => {
        let mut s = String::new();
        s.push_str(&self.desc[0..idx]);
        s.push_str(" …");
        s
      },
      None => self.desc.clone(),
} } }
```

traits

```
fn main() {
  let t = Talk::new(concat!("Rust ",
    "is a multi-paradigm system ",
    "programming language focused ",
    "on safety, especially safe ",
    "concurrency. Rust is syntactically ",
    "similar to C++, but is designed to ",
    "provide better memory safety while ",
    "maintaining high performance. In ",
    "this talk I want to introduce ",
    "listeners to the Rust programming ",
    "language."));
  println!("{}", t.summary());
}
```

traits

- If the first parameter of a method is `&self` or `&mut self`, an object is required.
- `t.summary()` is syntactic sugar for `Talk::summary(t)`
- Otherwise, we would call it a *static* method in OOP.
- `trait` keyword to declare traits
- `impl <trait> for <struct>` for implementation
- enums, structs, and unions can implement traits

traits coherence

> *One restriction to note with trait implementations is that we can implement a trait on a type only if either the trait or the type is local to our crate. This restriction is part of a property of programs called* coherence, *and more specifically the* orphan rule, *so named because the parent type is not present.*

Ensures that adding a dependency can't break your code.

traits coherence

Can we implement trait `fmt::Display` for our custom type? **Yes.**

Can we implement our custom trait `ToHTML` for type `Vec<String>`? **Yes.**

Can we implement trait `fmt::Display` for type `Vec<String>`? **No.**

traits coherence

Rust release 1.41.0 (released 2020-01-30) includes RFC 2451.

“Stabilize the `re_rebalance_coherence` feature” [PR 65879]

> *For better or worse, we allow implementing foreign traits for foreign types. For example, `impl From<Foo> for Vec<i32>` is something any crate can write, even though `From` is a foreign trait, and `Vec` is a foreign type. However, under the current coherence rules, we do not allow `impl<T> From<Foo> for Vec<T>`.*

traits default behavior

```
trait Submission {
  fn len(&self) -> u32;
  fn summary(&self) -> String;
  fn mentions_rust(&self) -> bool {
    true
  }
}
```

`mentions_rust` provides a default implementation. Can be overidden by any implementation. Adding one trait need not break backwards-compatibility (→ extensibility).

It isn't possible to call the default implementation from an overriding implementation of that same method.

`Submission` cannot have any mutable fields like `title: String`.

traits constants

```
trait Submission {
  const MAX_DURATION: u32 = 300;
  fn len(&self) -> u32;
  fn summary(&self) -> String;
  fn mentions_rust(&self) -> bool {
    true
  }
}
```

traits constants

Example via stackoverflow: Why does Rust assignment of const from trait to trait not work?

traits constants

```
trait A {
    const X: i32 = 1;
}
struct S;
impl A for S {}
trait B {
    const Y: i32 = A::X;
}
trait C {
    const Y: i32 = S::X;
}
fn main() {
}
```

traits constants

- Does not compile.
- In essence, any trait cannot depend on values of other traits.
- Constants are constants. Not default values (for other types).

traits constants

```rust
trait A {
    const X: i32 = 1;
}
struct S;
impl A for S {}
struct R;
impl A for R {
    const X: i32 = 42;
}

fn main() {
    println!("S: {}", S::X);        // S: 1
    println!("R: {}", R::X);        // R: 42
    // A::X alone is ambiguous
    println!("S: {}", <S as A>::X); // S: 1
    println!("R: {}", <R as A>::X); // R: 42
}
```

traits constants

```
trait A {
    fn get_x() -> i32 {
        1
    }
}
struct S;
impl A for S {}
struct R;
impl A for R {
    fn get_x() -> i32 {
        42
    }
}

fn main() {
    println!("S: {}", S::get_x());        // S: 1
    println!("R: {}", R::get_x());        // R: 42
    println!("S: {}", <S as A>::get_x()); // S: 1
    println!("R: {}", <R as A>::get_x()); // R: 42
}
```

traits default behavior

```rust
fn submit_to_glt(sub: impl Submission) {
  println!("Dear GLT-Team, we would like \
    to submit the following talk: {}",
    sub.summary());
}
```

The parameter can be *anything* satisfying the trait *Submission*.

traits and trait bounds

```
fn submit_to_glt<T: Submission>(sub: T) {
  println!("Dear GLT-Team, we would like \
    to submit the following talk: {}",
    sub.summary());
}
```

Another syntax. This is called *trait bound*.

traits and trait bound unions

```
fn represent_submission
    <T: Submission + Display>
    (sub: T) {
  // …
}
```

The parameter must implement Submission *and* Display.

traits and trait bounds with where

```
fn submit_to_glt<T>(sub: T)
    where T: Submission {
  println!("Dear GLT-Team, we would like \
    to submit the following talk: {}",
    sub.summary());
}
```

More generic example:

```
impl <A, D> MyTrait<A, D> for YourType where
    A: TraitB + TraitC,
    D: TraitE + TraitF {}
```

traits and trait bounds with bang

```
impl <A, D> MyTrait<A, D> for YourType where
    A: TraitB + TraitC,
    D: !TraitE + TraitF {}
```

This example does *not* compile. *Negative `!trait` bounds* were dropped (PR 586 “Negative bounds”).

traits and ?Sized

```rust
fn submit_to_glt<T: ?Sized>(sub: T) {
   // …
}
```

RFC 0490 DST syntax. Only works for the `Sized` trait.

Generics must be sized per default (its size is known at compile-time). But in some contexts it is allowed to be “maybe sized” (sized or unsized).

traits and trait bounds

```
fn compare(sub1: impl Submission,
           sub2: impl Submission) -> i8 {
  if sub1.mentions_rust() &&
    !sub2.mentions_rust() {
    -1
  } else if !sub1.mentions_rust() &&
            sub2.mentions_rust() {
    1
  } else {
    0
  }
}
```

traits and trait bounds

```
fn compare<T: Submission>
          (sub1: T, sub2: T) -> i8 {
  if sub1.mentions_rust() &&
    !sub2.mentions_rust() {
    -1
  } else if !sub1.mentions_rust() &&
            sub2.mentions_rust() {
    1
  } else {
    0
  }
}
```

traits and trait bounds

Question: Are these trait bounds the same?

traits and trait bounds

Question: Are these trait bounds the same?

- No.
- `impl Submission` requires two implementations of Submission
- `<T: Submission>` requires two times the *same* implementations of Submission

traits and trait bounds

```
struct Talk {
    desc: String,
    duration: u16,
}

struct Workshop {
    desc: String,
    duration: u16,
}

impl Submission for Talk { /* … */ }
impl Submission for Workshop { /* … */ }
```

traits and trait bounds

```
compare(
  Talk{
    desc: String::from("This talk …"),
    duration: 45
  }, Workshop{
    desc: String::from("In this workshop …"),
    duration: 90
  }
);
```

```
error[E0308]: mismatched types
[…]
expected struct `Talk`, found struct `Workshop`
```

traits and trait bounds

```rust
fn compare<T1: Submission, T2: Submission>
          (sub1: T1, sub2: T2) -> i8 {
  if sub1.mentions_rust() &&
    !sub2.mentions_rust() {
    -1
  } else if !sub1.mentions_rust() &&
             sub2.mentions_rust() {
    1
  } else {
    0
  }
}
```

This covers the `(Talk, Workshop)` case, of course.

traits and trait bounds

- A *marker trait* is a trait requiring implementation of 0 methods. Used to *mark* a property. See Send and Sync traits.
- We can also conditionally implement a trait for any type that implements another trait. Implementations of a trait on any type that satisfies the trait bounds are called *blanket implementations* and are extensively used in the Rust standard library.
- *Auto traits* are opt-in, built-in traits (OIBIT) which are unstable and not covered here (“automatically implemented traits”). “Auto trait implementation” occur in the standard library.

standard library traits

Iterator trait discussed before.

Add trait implements operator overloading of the addition operator `+`.

Deref trait is used for immutable dereferencing operations, like `*val`. `fn deref(&self) -> &Self::Target` is required.

Drop trait is used to run some code when a value goes out of scope. This is sometimes called a *destructor*. `fn drop(&mut self)` is required.

standard library traits

Send trait	is a marker trait for types that can be transferred across thread boundaries.
Sync trait	is a marker trait for types where it is safe to share references between threads.
Display trait	format trait for an empty format; `{}`.
Debug trait	provides the output in a programmer-facing, debugging context; `{:?}`.

Both require:

```
fmt(&self, f: &mut Formatter)
   -> Result<(), Error>
```

traits

Question: Can we provide references to traits?

Not discussed today. Topic *trait objects* and *dynamic dispatch*.

Dialogue: generics

type systems

How can we maintain the DRY (Don’t repeat yourself) principle in programming?

Opinionated answer:

strongly typed language Generics

dynamically typed language Duck typing

generics introduction

```
use std::mem::{size_of,align_of};
fn main() {
  println!("{} => {}", size_of::<i32>(),
           align_of::<i32>());       // 4 => 4
  println!("{} => {}", size_of::<u8>(),
           align_of::<u8>());        // 1 => 1
  println!("{} => {}", size_of::<*mut i32>(),
           align_of::<*mut i32>());  // 8 => 8
  println!("{} => {}", size_of::<usize>(),
           align_of::<usize>());     // 8 => 8
  println!("{} => {}", size_of::<char>(),
           align_of::<char>());      // 4 => 4
}
```

A more compact version would be nice.

generics introduction

```
  println!("{} ⇒ {}", size_of::<i32>(),
           align_of::<i32>());       // 4 ⇒ 4
```

Goal: make `i32` parametrizable. This example does *not* compile:

```
use std::mem::{size_of,align_of};
fn show(T) {
  println!("{} ⇒ {}", size_of::<T>(),
           align_of::<T>());
}
fn main() {
  show::<i32>();
  show::<u8>();
  // …
}
```

generics introduction

```
use std::mem::{size_of,align_of};
fn show<T>() {
  println!("{} ⇒ {}",
    size_of::<T>(),
    align_of::<T>());
}
fn main() {
  show::<i32>();       // 4 ⇒ 4
  show::<u8>();        // 1 ⇒ 1
  // …
}
```

This example compiles.

generics introduction

We are generic over *types*, not *values*!

So we cannot define a vector with N elements, where the vector is generic over N (length is value, not type).

generics and monomorphization

```rust
fn first<A, B>(tuple: (A, B)) -> A {
  let (a, b) = tuple;
  return a;
}

fn main() {
  assert_eq!(1, first((1, 9)));
}
```

generics and monomorphization

How are generics implemented? Generics are implemented using *monomorphization*. Monomorphization means specialized versions are generated for each type.

```
assert_eq!(1, first((1, 9)));
// version for integers is generated
assert_eq!('l', first(('l', 'p')));
// version for characters is generated
assert_eq!(3.14159, first((3.14159, 2.718281)));
// version for floats is generated
```

Monomorphization is how C++ templates are implemented (unlike Java interfaces).

generics and monomorphization

```
fn main() {
    let mut i = (1, 9);
    let mut c = ('l', 'p');
    let mut f = (3.14159, 2.718281);

    // avoids `first` to be inlined
    i.1 += 99;
    c.1 = 'r';
    f.1 = 42.0;

    assert_eq!(1, first(i));
    assert_eq!('l', first(c));
    assert_eq!(3.14159, first(f));
}
```

generics and monomorphization

Advantage: fast, specialized/optimized versions
Disadvantage: executable size increases

Only the first `first` call: 2,770,592 bytes
With two `first` calls: 2,772,032 bytes
With three `first` calls: 2,804,176 bytes

Differences are +1440 and +32144 bytes. Cannot tell why it is a non-linear increase, but the point is that the executable size increases.

Dialogue: Algebraic data types

algebraic data types

In computer programming, especially functional programming and type theory, an algebraic data type (ADT) is a kind of composite type, i.e., a type formed by combining other types.

—*Wikipedia*

algebraic data types

```
use std::fmt;

enum List {
    Nil,
    Cons(Box<List>, u64),
}
```

Algebraic? Sums and products. `List` is the sum of `Nil` and `Cons(_)`. `Cons` is the product of `Box<List>` and `u64`. Boxing? Avoids `recursive type `List` has infinite size`.

Article: Algebraic data types in four languages (namely Haskell, Scala, rust, and TypeScript)

algebraic data types

```
impl fmt::Display for List {
  fn fmt(&self, f: &mut fmt::Formatter<'_>)
     -> fmt::Result {
    match self {
      List::Cons(inner, item)
        => write!(f, "(cons {} {})", item, inner),
      List::Nil
        => write!(f, "nil"),
    }
  }
}
```

Recognize that Cons is addressed by `List::Cons`.

algebraic data types

Call:

```rust
fn main() {
  println!("{}", List::Cons(Box::new(
    List::Cons(Box::new(List::Nil), 1)), 2));
}
```

Output:

```
(cons 2 (cons 1 nil))
```

algebraic data types

```
use std::fmt;

enum Tree {
    Node(Box<Tree>, Box<Tree>),
    Leaf(u32),
    Empty,
}
```

algebraic data types

```
impl fmt::Display for Tree {
  fn fmt(&self, f: &mut fmt::Formatter<'_>)
     -> fmt::Result {
    match self {
      Tree::Node(left, right)
        => write!(f, "({}, {})", left, right),
      Tree::Leaf(item)
        => write!(f, "({})", item),
      Tree::Empty
        => write!(f, "_"),
    }
  }
}
```

algebraic data types

Call:

```rust
fn main() {
    println!("{}", Tree::Node(
        Box::new(Tree::Leaf(42)),
        Box::new(Tree::Empty)
    ));
}
```

Output:

```
((42), _)
```

Dialogue: Type inference

type inference

```
fn main() {
    let a = vec![135, 145, 4, 0];
    println!("{}", a[0] + 4f64);
}
```

type inference

```
error[E0277]: cannot add `f64` to `{integer}`
 --> src/main.rs:3:25
  |
3 |  println!("{}", a[0] + 4f64);
  |                      ^ no implementation for `{integer} + f64`
  |
  = help: the trait `std::ops::Add<f64>` is
          not implemented for `{integer}`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0277`.
error: could not compile `playground`.

To learn more, run the command again with --verbose.
```

The rust compiler keeps data types abstract as long as possible. The bit size of {integer} is not defined.

type inference

This enables *type inference*. The type of a value is *inferred* by the context it is used in.

```rust
fn main() {
  let a = 3;
  println!("{}", a + 4u32);
}
```

The type `u32` for `a` is inferred.

type inference

```rust
fn main() {
  let a = -3;
  println!("{}", a + 4u32);
}
```

Gives an error, because there is no solution for type inference.

```
error[E0277]: the trait bound
  `u32: std::ops::Neg` is not satisfied
```

type inference

I struggled with a type problem once. Reproduction of the problem didn’t show a problem, because type inference kicked in. Reduced example:

```rust
fn main() {
  let l = 5;
  println!("{} => {}", 'c',
    &format!("{:016b}", 0b01011)[16-l..16]);
}
```

Does it compile?

type inference

Yes, **output:**

```
c ⇒ 01011
```

BTW, out of bounds errors are either compile-time errors (few cases) or run-time errors (all other cases). Same for debug and release mode.

type inference

```rust
fn main() {
  let l = 5u8;
  println!("{} => {}", 'c',
    &format!("{:016b}", 0b01011)[16-l..16]);
}
```

Recognize that `l` is enforced with type `5u8`.

Does it compile?

type inference

No, **output:**

```
the type `std::string::String` cannot
be indexed by `std::ops::Range<u8>`
```

The first example inferred type **`usize`** for `l`. The second example uses **`u8`**. Indexing requires **`usize`**.

Dialogue: Type aliasing

type aliasing

```
type Point = (u8, u8);

fn main() {
  let p: Point = (41, 68);
  assert_eq!(p, (41, 68));
}
```

`Point` is a type alias of `(u8, u8)`.

type aliasing

```rust
type Point = (u8, u8);
type Two = (u8, u8);

fn main() {
  let p: Point = (41, 68);
  let t: Two = (41, 68);
  assert_eq!(p, (41, 68));
  assert_eq!(p, t);
}
```

Off-Topic reminder: `p.0` accesses the first **`u8`**, i.e. 41 (other languages use [] syntactically for indexing).

type aliasing: edge cases

```rust
struct A {
  a: u8,
  b: char,
}
type X = A;
impl X {
  fn inc(&mut self) {
    self.a += 1;
  }
}
fn main() {
  let mut inst = A {a: 42, b: 'l'};
  inst.inc();
  println!("{}", inst.a); // 43
}
```

type aliasing: edge cases

```rust
type ExtendedUint = u32;

impl ExtendedUint {
    fn inc(&self) -> u32 {
        self.0 + 1
    }
}

fn main() {
    let h: ExtendedUint = 41;
    println!("{:?}", h.inc());
}
```

Does it compile?

type aliasing: edge cases

```
error[E0390]: only a single inherent implementation
              marked with `#[lang = "u32"]` is allowed
              for the `u32` primitive
 --> src/main.rs:3:1
  |
3 | / impl ExtendedUint {
4 | |     fn inc(&self) -> u32 {
5 | |         self.0 + 1
6 | |     }
7 | | }
  | |_^
  |
help: consider using a trait to implement these methods
 --> src/main.rs:3:1
  |
3 | / impl ExtendedUint {
4 | |     fn inc(&self) -> u32 {
5 | |         self.0 + 1
6 | |     }
7 | | }
  | |_^
error: aborting due to previous error
```

Dialogue: type systems

Haskell

Haskell type classes

Rust’s traits are based on Haskell’s type classes.

> *Type classes are defined by specifying a set of function or constant names, together with their respective types, that must exist for every type that belongs to the class.*
> —*Wikipedia: Type class*

Haskell type class example

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Haskell: default behavior

```haskell
class Monoid a where
    mempty  :: a
    mappend :: a -> a -> a

    -- | Fold a list using the monoid.
    --
    -- For most types, the default definition
    -- for 'mconcat' will be used, but the
    -- function is included in the class
    -- definition so that an optimized version
    -- can be provided for specific types.
    mconcat :: [a] -> a
    mconcat = foldr mappend mempty
```

→ via Haskell base: Monoid

Haskell: minimal complete definition

In Haskell documentation, you often find a *minimal complete definition* list:

> ***Minimal complete definition***
> *foldMap | foldr*

In order to implement Data.Foldable, you must implement at least `foldMap` and `foldr`. Otherwise the default implementations might not work.

I have never seen such documented requirements yet in rust.

Haskell: differences

What is different from rust’s traits?

1. Haskell does not bind the implementation to some object (i.e. `&self`)
2. Initially rust did not have support for higher-order typeclasses. Now it has. Haskell always had.

based on stackoverflow

Haskell: algebraic data types

First example:

```haskell
data List a = Nil | Cons a (List a)
```

Second example:

```haskell
data Tree = Empty
          | Leaf Int
          | Node Tree Tree
```

Haskell: algebraic data types

Pattern matching:

```
hasChild :: Tree -> Bool
hasChild Empty      = False
hasChild (Leaf _)   = False
hasChild (Node _ _) = True
```

C++

C++: templates and concepts

- C++ extends C with classes (compare with The Essence of C++)
- Interfaces somewhat correspond to abstract classes
- Templates were introduced at about 1995; allow code generation and metaprogramming; unrestricted
- Strong notion of subtyping, polymorphism and data encapsulation
- Lots of inspiration for rust
- C++20 introduces concepts (first discussed in 1998)

C++: type inference

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    auto x = 4;
    auto y = 3.37;
    auto ptr = &x;
    cout << typeid(x).name() << endl    // i
         << typeid(y).name() << endl    // d
         << typeid(ptr).name() << endl; // Pi
    return 0;
}
```

auto keyword since C++11.

→ via geeksforgeeks.org

C++: default behavior through subtyping

```
struct A {
  int x;
  // user-defined default constructor
  A(int x = 1): x(x) {}
};

struct B: A {
  // B::B() is inherited from A::A()
};
```

→ via C++: default_constructor

C++: abstract class

```
class Box {
  public:
    virtual double getVolume() = 0;

  private:
    double length;
    double breadth;
    double height;
};
```

A class becomes abstract, because one of its methods is abstract (unimplemented).

→ via tutorialspoint.com

C++: templates

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

// …
GetMax(myAge, yourAge) // integers
GetMax(lakeTemperature, seaTemperature) // float
```

→ via cplusplus.com

C++: templates

```
template <unsigned int n>
struct factorial {
  enum { value = n * factorial<n - 1>::value };
};

template <>
struct factorial<0> {
  enum { value = 1 };
};
```

→ via C++ templates are Turing-complete

C++: concepts

```cpp
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    { a == b } -> std::boolean;
    { a != b } -> std::boolean;
};
```

→ via Wikipedia: Concepts (C++)

Java

Java and interfaces

```
interface Bicycle {
    void speedUp(int increment);
}

class ACMEBicycle implements Bicycle {
    int speed = 0;
    void speedUp(int increment) {
        speed = speed + increment;
    }
}
```

→ via docs.oracle.com

Java and interfaces

- `Type implements Interface` declare implementation of an interface
- list of methods and constants required to satisfy this interface
- emulates multiple inheritance, embedded in subtyping hierarchy
- class can implement some interface incompletely ($\rightarrow$ abstract class)
- `Interface extends Interface1, Interface2` defines inheritance for interfaces
- Java 8: default and static methods may have implementation in interface definition
- Java 9: private and private static methods

Go

Go: interfaces and contracts

```go
package main

import (
  "fmt"
)

type Adder interface {
  add() uint32
}
```

Go: interfaces and contracts

```go
type Add struct {
  x uint32
  y uint32
}

func (a Add) add() uint32 {
  return a.x + a.y
}

func main() {
  var sum Adder = Add{10, 32}
  fmt.Println(sum.add())
}
```

Go: interfaces and contracts

- Go has interfaces (no constant types, no subtyping, but embedding)
- Go types implement an interface if they implement the corresponding method signatures implicitly (no `implements` keyword)
- Go does not have generics (see sort, code duplication and casting is common)
- Lack of generics was identified as one of the Top 3 major challenges in the 2018 developer survey

Go: interfaces and contracts

A draft for Go 2.x:

```go
func Print(s []T) {
  for _, v := range s {
    // cannot coerce to string implicitly.
    // must implement String() → no guarantee.
    fmt.Println(v)
  }
}
```

→ via go2draft contracts on github

Go: interfaces and contracts

A draft for Go 2.x:

```
contract stringer(T) {
  T String() string
}

func Stringify(type T stringer)(s []T)
              (ret []string)
{
  for _, v := range s {
    ret = append(ret, v.String())
  }
  return ret
}
```

Go: interfaces and contracts

- Draft is 6 months old
- Current 1.14 release (yesterday) does not feature contracts
- Go contracts are based on C++ concepts
- Summary: Golang concepts from an OOP point of view

Nominal type system versus structural type system

Nominal type system:

- “compatibility and equivalence of data types is determined by explicit declarations and/or the name of the types”
- Who implements What , name is normative

Structural type system:

- “type compatibility and equivalence are determined by the type’s actual structure or definition”
- If Who has the structure of What , Who implements What

My point: Go contracts / C++ concepts are structural features of their respective type system

Epilogue

Opinionated summary

Rust is multiparadigmatic.

imperative	all features except for goto
OOP	no subtyping provided, but default implementations for code reuse. No polymorphism, but generics. Data encapsulation via `pub`.
FP	Iterator trait features many basic function. Higher order functions via generics, Fn/FnOnce/FnMut traits, and closures. But no dependent types or alike.
decl./DSL	not really, but macros allow some flexibility and operator overloading through traits

Quiz - part 1

rust traits are based on

`Type implements Traits` is written as

traits can be implemented for

generics parametrize something with

the orphan rule states

monomorphisation means

Quiz - part 1

rust traits are based on Haskell’s type classes

`Type implements Traits` is written as

traits can be implemented for

generics parametrize something with

the orphan rule states

monomorphisation means

Quiz - part 1

rust traits are based on Haskell's type classes

`Type implements Traits` is written as

```
impl Trait for Type
```

traits can be implemented for

generics parametrize something with

the orphan rule states

monomorphisation means

Quiz - part 1

rust traits are based on Haskell's type classes

`Type implements Traits` is written as

```
          impl Trait for Type
```

traits can be implemented for `struct`, `enum`, `union`

generics parametrize something with

the orphan rule states

monomorphisation means

Quiz - part 1

rust traits are based on Haskell's type classes

`Type implements Traits` is written as

`impl Trait for Type`

traits can be implemented for `struct`, `enum`, `union`

generics parametrize something with some type

the orphan rule states

monomorphisation means

Quiz - part 1

rust traits are based on Haskell's type classes

`Type implements Traits` is written as

```
impl Trait for Type
```

traits can be implemented for `struct`, `enum`, `union`

generics parametrize something with some type

the orphan rule states a child (implementation) must know its parents (trait, struct) [to implement a trait]

monomorphisation means

Quiz - part 1

rust traits are based on Haskell’s type classes

`Type implements Traits` is written as
`impl Trait for Type`

traits can be implemented for `struct`, `enum`, `union`

generics parametrize something with some type

the orphan rule states a child (implementation) must know its parents (trait, struct) [to implement a trait]

monomorphisation means the source code is generated for each type that is used, just like C++ templates

Quiz - part 2

Originally, C++ designed templates to be

C++ templates are

C++20 allows compile-time protocols with

A difference between Rust traits and Java interfaces is

Go interfaces are not nominal, but

Algebraic data types are

Marker traits are

Quiz - part 2

Originally, C++ designed templates to be unrestricted

C++ templates are

C++20 allows compile-time protocols with

A difference between Rust traits and Java interfaces is

Go interfaces are not nominal, but

Algebraic data types are

Marker traits are

Quiz - part 2

Originally, C++ designed templates to be unrestricted

C++ templates are turing-complete

C++20 allows compile-time protocols with

A difference between Rust traits and Java interfaces is

Go interfaces are not nominal, but

Algebraic data types are

Marker traits are

Quiz - part 2

Originally, C++ designed templates to be unrestricted

C++ templates are turing-complete

C++20 allows compile-time protocols with concepts

A difference between Rust traits and Java interfaces is

Go interfaces are not nominal, but

Algebraic data types are

Marker traits are

Quiz - part 2

Originally, C++ designed templates to be unrestricted

C++ templates are turing-complete

C++20 allows compile-time protocols with concepts

A difference between Rust traits and Java interfaces is
incomplete implementation

Go interfaces are not nominal, but

Algebraic data types are

Marker traits are

Quiz - part 2

Originally, C++ designed templates to be unrestricted

C++ templates are turing-complete

C++20 allows compile-time protocols with concepts

A difference between Rust traits and Java interfaces is
incomplete implementation

Go interfaces are not nominal, but structural

Algebraic data types are

Marker traits are

Quiz - part 2

Originally, C++ designed templates to be unrestricted

C++ templates are turing-complete

C++20 allows compile-time protocols with concepts

A difference between Rust traits and Java interfaces is
incomplete implementation

Go interfaces are not nominal, but structural

Algebraic data types are product and sum of sets of data types

Marker traits are

Quiz - part 2

Originally, C++ designed templates to be unrestricted

C++ templates are turing-complete

C++20 allows compile-time protocols with concepts

A difference between Rust traits and Java interfaces is incomplete implementation

Go interfaces are not nominal, but structural

Algebraic data types are product and sum of sets of data types

Marker traits are traits that don't require any methods to be implemented

Next time

Next meetup Wed, 2020/03/25.

March ? (mostly depends on David’s content)

My next topics:

- trait objects, dyn keyword
- typestate pattern, newtype idiom
- modularization in crates, and modules. pub keyword

April Hacker Jeopardy

Thank you!