Rust error codes index

This page lists all the error codes emitted by the Rust compiler.

Error code E0001

Note: this error code is no longer emitted by the compiler.

This error suggests that the expression arm corresponding to the noted pattern will never be reached as for all possible values of the expression being matched, one of the preceding patterns will match.

This means that perhaps some of the preceding patterns are too general, this one is too specific or the ordering is incorrect.

For example, the following `match` block has too many arms:

```
match Some(0) {
    Some(bar) => {/* ... */}
    x => {/* ... */} // This handles the `None` case
    _ => {/* ... */} // All possible cases have already been handled
}
```

`match` blocks have their patterns matched in order, so, for example, putting a wildcard arm above a more specific arm will make the latter arm irrelevant.

Ensure the ordering of the match arm is correct and remove any superfluous arms.

Error code E0002

Note: this error code is no longer emitted by the compiler.

This error indicates that an empty match expression is invalid because the type it is matching on is non-empty (there exist values of this type). In safe code it is impossible to create an instance of an empty type, so empty match expressions are almost never desired. This error is typically fixed by adding one or more cases to the match expression.

An example of an empty type is `enum Empty { }`. So, the following will work:

```
enum Empty {}

fn foo(x: Empty) {
    match x {
        // empty
    }
}
```

However, this won't:

```
fn foo(x: Option<String>) {
    match x {
        // empty
    }
}
```

Error code E0004

This error indicates that the compiler cannot guarantee a matching pattern for one or more possible inputs to a match expression. Guaranteed matches are required in order to assign values to match expressions, or alternatively, determine the flow of execution.

Erroneous code example:

```
enum Terminator {
    HastaLaVistaBaby,
    TalkToMyHand,
}

let x = Terminator::HastaLaVistaBaby;

match x { // error: non-exhaustive patterns: `HastaLaVistaBaby` not covered
    Terminator::TalkToMyHand => {}
}
```

If you encounter this error you must alter your patterns so that every possible value of the input type is matched. For types with a small number of variants (like enums) you should probably cover all cases explicitly. Alternatively, the underscore `_` wildcard pattern can be added after all other patterns to match "anything else". Example:

```
enum Terminator {
    HastaLaVistaBaby,
    TalkToMyHand,
}

let x = Terminator::HastaLaVistaBaby;

match x {
    Terminator::TalkToMyHand => {}
    Terminator::HastaLaVistaBaby => {}
}

// or:

match x {
    Terminator::TalkToMyHand => {}
    _ => {}
}
```

Error code E0005

Patterns used to bind names must be irrefutable, that is, they must guarantee that a name will be extracted in all cases.

Erroneous code example:

```
let x = Some(1);
let Some(y) = x;
// error: refutable pattern in local binding: `None` not covered
```

If you encounter this error you probably need to use a `match` or `if let` to deal with the possibility of failure. Example:

```
let x = Some(1);

match x {
    Some(y) => {
        // do something
    },
    None => {}
}

// or:

if let Some(y) = x {
    // do something
}
```

Error code E0007

Note: this error code is no longer emitted by the compiler.

This error indicates that the bindings in a match arm would require a value to be moved into more than one location, thus violating unique ownership. Code like the following is invalid as it requires the entire `Option<String>` to be moved into a variable called `op_string` while simultaneously requiring the inner `String` to be moved into a variable called `s`.

Erroneous code example:

```
#![feature(bindings_after_at)]

let x = Some("s".to_string());

match x {
    op_string @ Some(s) => {}, // error: use of moved value
    None => {},
}
```

See also the error E0303.

Error code E0009

Note: this error code is no longer emitted by the compiler.

In a pattern, all values that don't implement the `Copy` trait have to be bound the same way. The goal here is to avoid binding simultaneously by-move and by-ref.

This limitation may be removed in a future version of Rust.

Erroneous code example:

```
#![feature(move_ref_pattern)]

struct X { x: (), }

let x = Some((X { x: () }, X { x: () }));
match x {
    Some((y, ref z)) => {}, // error: cannot bind by-move and by-ref in the
                            //        same pattern
    None => panic!()
}
```

You have two solutions:

Solution #1: Bind the pattern's values the same way.

```
struct X { x: (), }

let x = Some((X { x: () }, X { x: () }));
match x {
    Some((ref y, ref z)) => {},
    // or Some((y, z)) => {}
    None => panic!()
}
```

Solution #2: Implement the `Copy` trait for the `X` structure.

However, please keep in mind that the first solution should be preferred.

```
#[derive(Clone, Copy)]
struct X { x: (), }

let x = Some((X { x: () }, X { x: () }));
match x {
    Some((y, ref z)) => {},
    None => panic!()
}
```

Error code E0010

The value of statics and constants must be known at compile time, and they live for the entire lifetime of a program. Creating a boxed value allocates memory on the heap at runtime, and therefore cannot be done at compile time.

Erroneous code example:

```
const CON : Vec<i32> = vec![1, 2, 3];
```

Error code E0013

Static and const variables can refer to other const variables. But a const variable cannot refer to a static variable.

Erroneous code example:

```
static X: i32 = 42;
const Y: i32 = X;
```

In this example, `Y` cannot refer to `X`. To fix this, the value can be extracted as a const and then used:

```
const A: i32 = 42;
static X: i32 = A;
const Y: i32 = A;
```

Error code E0014

Note: this error code is no longer emitted by the compiler.

Constants can only be initialized by a constant value or, in a future version of Rust, a call to a const function. This error indicates the use of a path (like a::b, or x) denoting something other than one of these allowed items.

Erroneous code example:

```
const FOO: i32 = { let x = 0; x }; // 'x' isn't a constant nor a function!
```

To avoid it, you have to replace the non-constant value:

```
const FOO: i32 = { const X : i32 = 0; X };
// or even:
const FOO2: i32 = { 0 }; // but brackets are useless here
```

Error code E0015

A non- `const` function was called in a `const` context.

Erroneous code example:

```
fn create_some() -> Option<u8> {
    Some(1)
}

// error: cannot call non-const fn `create_some` in constants
const FOO: Option<u8> = create_some();
```

All functions used in a `const` context (constant or static expression) must be marked `const` .

To fix this error, you can declare `create_some` as a constant function:

```
// declared as a `const` function:
const fn create_some() -> Option<u8> {
    Some(1)
}

const FOO: Option<u8> = create_some(); // no error!
```

Error code E0023

A pattern attempted to extract an incorrect number of fields from a variant.

Erroneous code example:

```
enum Fruit {
    Apple(String, String),
    Pear(u32),
}

let x = Fruit::Apple(String::new(), String::new());

match x {
    Fruit::Apple(a) => {}, // error!
    _ => {}
}
```

A pattern used to match against an enum variant must provide a sub-pattern for each field of the enum variant.

Here the `Apple` variant has two fields, and should be matched against like so:

```
enum Fruit {
    Apple(String, String),
    Pear(u32),
}

let x = Fruit::Apple(String::new(), String::new());

// Correct.
match x {
    Fruit::Apple(a, b) => {},
    _ => {}
}
```

Matching with the wrong number of fields has no sensible interpretation:

```
enum Fruit {
    Apple(String, String),
    Pear(u32),
}

let x = Fruit::Apple(String::new(), String::new());

// Incorrect.
match x {
    Fruit::Apple(a) => {},
    Fruit::Apple(a, b, c) => {},
}
```

Check how many fields the enum was declared with and ensure that your pattern uses the same number.

Error code E0025

Each field of a struct can only be bound once in a pattern.

Erroneous code example:

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

fn main(){
    let x = Foo { a:1, b:2 };

    let Foo { a: x, a: y } = x;
    // error: field `a` bound multiple times in the pattern
}
```

Each occurrence of a field name binds the value of that field, so to fix this error you will have to remove or alter the duplicate uses of the field name. Perhaps you misspelled another field name? Example:

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

fn main(){
    let x = Foo { a:1, b:2 };

    let Foo { a: x, b: y } = x; // ok!
}
```

Error code E0026

A struct pattern attempted to extract a nonexistent field from a struct.

Erroneous code example:

```
struct Thing {
    x: u32,
    y: u32,
}

let thing = Thing { x: 0, y: 0 };

match thing {
    Thing { x, z } => {} // error: `Thing::z` field doesn't exist
}
```

If you are using shorthand field patterns but want to refer to the struct field by a different name, you should rename it explicitly. Struct fields are identified by the name used before the colon `:` so struct patterns should resemble the declaration of the struct type being matched.

```
struct Thing {
    x: u32,
    y: u32,
}

let thing = Thing { x: 0, y: 0 };

match thing {
    Thing { x, y: z } => {} // we renamed `y` to `z`
}
```

Error code E0027

A pattern for a struct fails to specify a sub-pattern for every one of the struct's fields.

Erroneous code example:

```
struct Dog {
    name: String,
    age: u32,
}

let d = Dog { name: "Rusty".to_string(), age: 8 };

// This is incorrect.
match d {
    Dog { age: x } => {}
}
```

To fix this error, ensure that each field from the struct's definition is mentioned in the pattern, or use `..` to ignore unwanted fields. Example:

```
struct Dog {
    name: String,
    age: u32,
}

let d = Dog { name: "Rusty".to_string(), age: 8 };

match d {
    Dog { name: ref n, age: x } => {}
}

// This is also correct (ignore unused fields).
match d {
    Dog { age: x, .. } => {}
}
```

Error code E0029

Something other than numbers and characters has been used for a range.

Erroneous code example:

```
let string = "salutations !";

// The ordering relation for strings cannot be evaluated at compile time,
// so this doesn't work:
match string {
    "hello" ..= "world" => {}
    _ => {}
}

// This is a more general version, using a guard:
match string {
    s if s >= "hello" && s <= "world" => {}
    _ => {}
}
```

In a match expression, only numbers and characters can be matched against a range. This is because the compiler checks that the range is non-empty at compile-time, and is unable to evaluate arbitrary comparison functions. If you want to capture values of an orderable type between two end-points, you can use a guard.

Error code E0030

When matching against a range, the compiler verifies that the range is non-empty. Range patterns include both end-points, so this is equivalent to requiring the start of the range to be less than or equal to the end of the range.

Erroneous code example:

```
match 5u32 {
    // This range is ok, albeit pointless.
    1 ..= 1 => {}
    // This range is empty, and the compiler can tell.
    1000 ..= 5 => {}
}
```

Error code E0033

A trait type has been dereferenced.

Erroneous code example:

```
let trait_obj: &SomeTrait = &"some_value";

// This tries to implicitly dereference to create an unsized local variable.
let &invalid = trait_obj;

// You can call methods without binding to the value being pointed at.
trait_obj.method_one();
trait_obj.method_two();
```

A pointer to a trait type cannot be implicitly dereferenced by a pattern. Every trait defines a type, but because the size of trait implementers isn't fixed, this type has no compile-time size. Therefore, all accesses to trait types must be through pointers. If you encounter this error you should try to avoid dereferencing the pointer.

You can read more about trait objects in the Trait Objects section of the Reference.

Error code E0034

The compiler doesn't know what method to call because more than one method has the same prototype.

Erroneous code example:

```
struct Test;

trait Trait1 {
    fn foo();
}

trait Trait2 {
    fn foo();
}

impl Trait1 for Test { fn foo() {} }
impl Trait2 for Test { fn foo() {} }

fn main() {
    Test::foo() // error, which foo() to call?
}
```

To avoid this error, you have to keep only one of them and remove the others. So let's take our example and fix it:

```
struct Test;

trait Trait1 {
    fn foo();
}

impl Trait1 for Test { fn foo() {} }

fn main() {
    Test::foo() // and now that's good!
}
```

However, a better solution would be using fully explicit naming of type and trait:

```
struct Test;

trait Trait1 {
    fn foo();
}

trait Trait2 {
    fn foo();
}

impl Trait1 for Test { fn foo() {} }
impl Trait2 for Test { fn foo() {} }

fn main() {
    <Test as Trait1>::foo()
}
```

One last example:

```
trait F {
    fn m(&self);
}

trait G {
    fn m(&self);
}

struct X;

impl F for X { fn m(&self) { println!("I am F"); } }
impl G for X { fn m(&self) { println!("I am G"); } }

fn main() {
    let f = X;

    F::m(&f); // it displays "I am F"
    G::m(&f); // it displays "I am G"
}
```

Error code E0038

For any given trait `Trait` there may be a related *type* called the *trait object type* which is typically written as `dyn Trait`. In earlier editions of Rust, trait object types were written as plain `Trait` (just the name of the trait, written in type positions) but this was a bit too confusing, so we now write `dyn Trait`.

Some traits are not allowed to be used as trait object types. The traits that are allowed to be used as trait object types are called "object-safe" traits. Attempting to use a trait object type for a trait that is not object-safe will trigger error E0038.

Two general aspects of trait object types give rise to the restrictions:

1. Trait object types are dynamically sized types (DSTs), and trait objects of these types can only be accessed through pointers, such as `&dyn Trait` or `Box<dyn Trait>`. The size of such a pointer is known, but the size of the `dyn Trait` object pointed-to by the pointer is *opaque* to code working with it, and different trait objects with the same trait object type may have different sizes.

2. The pointer used to access a trait object is paired with an extra pointer to a "virtual method table" or "vtable", which is used to implement dynamic dispatch to the object's implementations of the trait's methods. There is a single such vtable for each trait implementation, but different trait objects with the same trait object type may point to vtables from different implementations.

The specific conditions that violate object-safety follow, most of which relate to missing size information and vtable polymorphism arising from these aspects.

The trait requires `Self: Sized`

Traits that are declared as `Trait: Sized` or which otherwise inherit a constraint of `Self:Sized` are not object-safe.

The reasoning behind this is somewhat subtle. It derives from the fact that Rust requires (and defines) that every trait object type `dyn Trait` automatically implements `Trait`. Rust does this to simplify error reporting and ease interoperation between static and dynamic polymorphism. For example, this code works:

```
trait Trait {
}

fn static_foo<T:Trait + ?Sized>(b: &T) {
}

fn dynamic_bar(a: &dyn Trait) {
    static_foo(a)
}
```

This code works because `dyn Trait`, if it exists, always implements `Trait`.

However as we know, any `dyn Trait` is also unsized, and so it can never implement a sized trait like `Trait:Sized`. So, rather than allow an exception to the rule that `dyn Trait` always implements `Trait`, Rust chooses to prohibit such a `dyn Trait` from existing at all.

Only unsized traits are considered object-safe.

Generally, `Self: Sized` is used to indicate that the trait should not be used as a trait object. If the trait comes from your own crate, consider removing this restriction.

Method references the `Self` type in its parameters or return type

This happens when a trait has a method like the following:

```
trait Trait {
    fn foo(&self) -> Self;
}

impl Trait for String {
    fn foo(&self) -> Self {
        "hi".to_owned()
    }
}

impl Trait for u8 {
    fn foo(&self) -> Self {
        1
    }
}
```

(Note that `&self` and `&mut self` are okay, it's additional `Self` types which cause this problem.)

In such a case, the compiler cannot predict the return type of `foo()` in a situation like the following:

```
trait Trait {
    fn foo(&self) -> Self;
}

fn call_foo(x: Box<dyn Trait>) {
    let y = x.foo(); // What type is y?
    // ...
}
```

If only some methods aren't object-safe, you can add a `where Self: Sized` bound on them to mark them as explicitly unavailable to trait objects. The functionality will still be available to all other implementers, including `Box<dyn Trait>` which is itself sized (assuming you `impl Trait for Box<dyn Trait>`).

```
trait Trait {
    fn foo(&self) -> Self where Self: Sized;
    // more functions
}
```

Now, `foo()` can no longer be called on a trait object, but you will now be allowed to make a trait object, and that will be able to call any object-safe methods. With such a bound, one can still call `foo()` on types implementing that trait that aren't behind trait objects.

Method has generic type parameters

As mentioned before, trait objects contain pointers to method tables. So, if we have:

```
trait Trait {
    fn foo(&self);
}

impl Trait for String {
    fn foo(&self) {
        // implementation 1
    }
}

impl Trait for u8 {
    fn foo(&self) {
        // implementation 2
    }
}
// ...
```

At compile time each implementation of `Trait` will produce a table containing the various methods (and other items) related to the implementation, which will be used as the virtual method table for a `dyn Trait` object derived from that implementation.

This works fine, but when the method gains generic parameters, we can have a problem.

Usually, generic parameters get *monomorphized*. For example, if I have

```
fn foo<T>(x: T) {
    // ...
}
```

The machine code for `foo::<u8>()`, `foo::<bool>()`, `foo::<String>()`, or any other instantiation is different. Hence the compiler generates the implementation on-demand. If you call `foo()` with a `bool` parameter, the compiler will only generate code for `foo::<bool>()`. When we have additional type parameters, the number of monomorphized implementations the compiler generates does not grow drastically, since the compiler will only generate an implementation if the function is called with fully concrete arguments (i.e., arguments which do not contain any generic parameters).

However, with trait objects we have to make a table containing *every* object that implements the trait. Now, if it has type parameters, we need to add implementations for every type that implements the trait, and there could theoretically be an infinite number of types.

For example, with:

```
trait Trait {
    fn foo<T>(&self, on: T);
    // more methods
}

impl Trait for String {
    fn foo<T>(&self, on: T) {
        // implementation 1
    }
}

impl Trait for u8 {
    fn foo<T>(&self, on: T) {
        // implementation 2
    }
}

// 8 more implementations
```

Now, if we have the following code:

```
fn call_foo(thing: Box<dyn Trait>) {
    thing.foo(true); // this could be any one of the 8 types above
    thing.foo(1);
    thing.foo("hello");
}
```

We don't just need to create a table of all implementations of all methods of `Trait`, we need to create such a table, for each different type fed to `foo()`. In this case this turns out to be (10 types implementing `Trait`)*(3 types being fed to `foo()`) = 30 implementations!

With real world traits these numbers can grow drastically.

To fix this, it is suggested to use a `where Self: Sized` bound similar to the fix for the sub-error above if you do not intend to call the method with type parameters:

```
trait Trait {
    fn foo<T>(&self, on: T) where Self: Sized;
    // more methods
}
```

If this is not an option, consider replacing the type parameter with another trait object (e.g., if `T: OtherTrait`, use `on: Box<dyn OtherTrait>`). If the number of types you intend to feed to this method is limited, consider manually listing out the methods of different types.

Method has no receiver

Methods that do not take a `self` parameter can't be called since there won't be a way to get a pointer to the method table for them.

```
trait Foo {
    fn foo() -> u8;
}
```

This could be called as `<Foo as Foo>::foo()`, which would not be able to pick an implementation.

Adding a `Self: Sized` bound to these methods will generally make this compile.

```
trait Foo {
    fn foo() -> u8 where Self: Sized;
}
```

Trait contains associated constants

Just like static functions, associated constants aren't stored on the method table. If the trait or any subtrait contain an associated constant, they cannot be made into an object.

```
trait Foo {
    const X: i32;
}

impl Foo {}
```

A simple workaround is to use a helper method instead:

```
trait Foo {
    fn x(&self) -> i32;
}
```

Trait uses `Self` as a type parameter in the supertrait listing

This is similar to the second sub-error, but subtler. It happens in situations like the following:

```
trait Super<A: ?Sized> {}

trait Trait: Super<Self> {
}

struct Foo;

impl Super<Foo> for Foo{}

impl Trait for Foo {}

fn main() {
    let x: Box<dyn Trait>;
}
```

Here, the supertrait might have methods as follows:

```
trait Super<A: ?Sized> {
    fn get_a(&self) -> &A; // note that this is object safe!
}
```

If the trait `Trait` was deriving from something like `Super<String>` or `Super<T>` (where `Foo` itself is `Foo<T>`), this is okay, because given a type `get_a()` will definitely return an object of that type.

However, if it derives from `Super<Self>`, even though `Super` is object safe, the method `get_a()` would return an object of unknown type when called on the function. `Self` type parameters let us make object safe traits no longer safe, so they are forbidden when specifying supertraits.

There's no easy fix for this. Generally, code will need to be refactored so that you no longer need to derive from `Super<Self>`.

Error code E0040

It is not allowed to manually call destructors in Rust.

Erroneous code example:

```
struct Foo {
    x: i32,
}

impl Drop for Foo {
    fn drop(&mut self) {
        println!("kaboom");
    }
}

fn main() {
    let mut x = Foo { x: -7 };
    x.drop(); // error: explicit use of destructor method
}
```

It is unnecessary to do this since `drop` is called automatically whenever a value goes out of scope. However, if you really need to drop a value by hand, you can use the `std::mem::drop` function:

```
struct Foo {
    x: i32,
}
impl Drop for Foo {
    fn drop(&mut self) {
        println!("kaboom");
    }
}
fn main() {
    let mut x = Foo { x: -7 };
    drop(x); // ok!
}
```

Error code E0044

You cannot use type or const parameters on foreign items.

Example of erroneous code:

```
extern "C" { fn some_func<T>(x: T); }
```

To fix this, replace the generic parameter with the specializations that you need:

```
extern "C" { fn some_func_i32(x: i32); }
extern "C" { fn some_func_i64(x: i64); }
```

Error code E0045

Variadic parameters have been used on a non-C ABI function.

Erroneous code example:

```
extern "Rust" {
    fn foo(x: u8, ...); // error!
}
```

Rust only supports variadic parameters for interoperability with C code in its FFI. As such, variadic parameters can only be used with functions which are using the C ABI. To fix such code, put them in an extern "C" block:

```
extern "C" {
    fn foo (x: u8, ...);
}
```

Error code E0046

Items are missing in a trait implementation.

Erroneous code example:

```
trait Foo {
    fn foo();
}

struct Bar;

impl Foo for Bar {}
// error: not all trait items implemented, missing: `foo`
```

When trying to make some type implement a trait `Foo`, you must, at minimum, provide implementations for all of `Foo`'s required methods (meaning the methods that do not have default implementations), as well as any required trait items like associated types or constants. Example:

```
trait Foo {
    fn foo();
}

struct Bar;

impl Foo for Bar {
    fn foo() {} // ok!
}
```

Error code E0049

An attempted implementation of a trait method has the wrong number of type or const parameters.

Erroneous code example:

```
trait Foo {
    fn foo<T: Default>(x: T) -> Self;
}

struct Bar;

// error: method `foo` has 0 type parameters but its trait declaration has 1
// type parameter
impl Foo for Bar {
    fn foo(x: bool) -> Self { Bar }
}
```

For example, the `Foo` trait has a method `foo` with a type parameter `T`, but the implementation of `foo` for the type `Bar` is missing this parameter. To fix this error, they must have the same type parameters:

```
trait Foo {
    fn foo<T: Default>(x: T) -> Self;
}

struct Bar;

impl Foo for Bar {
    fn foo<T: Default>(x: T) -> Self { // ok!
        Bar
    }
}
```

Error code E0050

An attempted implementation of a trait method has the wrong number of function parameters.

Erroneous code example:

```
trait Foo {
    fn foo(&self, x: u8) -> bool;
}

struct Bar;

// error: method `foo` has 1 parameter but the declaration in trait
`Foo::foo`
// has 2
impl Foo for Bar {
    fn foo(&self) -> bool { true }
}
```

For example, the `Foo` trait has a method `foo` with two function parameters (`&self` and `u8`), but the implementation of `foo` for the type `Bar` omits the `u8` parameter. To fix this error, they must have the same parameters:

```
trait Foo {
    fn foo(&self, x: u8) -> bool;
}

struct Bar;

impl Foo for Bar {
    fn foo(&self, x: u8) -> bool { // ok!
        true
    }
}
```

Error code E0053

The parameters of any trait method must match between a trait implementation and the trait definition.

Erroneous code example:

```
trait Foo {
    fn foo(x: u16);
    fn bar(&self);
}

struct Bar;

impl Foo for Bar {
    // error, expected u16, found i16
    fn foo(x: i16) { }

    // error, types differ in mutability
    fn bar(&mut self) { }
}
```

Error code E0054

It is not allowed to cast to a bool.

Erroneous code example:

```
let x = 5;

// Not allowed, won't compile
let x_is_nonzero = x as bool;
```

If you are trying to cast a numeric type to a bool, you can compare it with zero instead:

```
let x = 5;

// Ok
let x_is_nonzero = x != 0;
```

Error code E0055

During a method call, a value is automatically dereferenced as many times as needed to make the value's type match the method's receiver. The catch is that the compiler will only attempt to dereference a number of times up to the recursion limit (which can be set via the `recursion_limit` attribute).

For a somewhat artificial example:

```
#![recursion_limit="4"]

struct Foo;

impl Foo {
    fn foo(&self) {}
}

fn main() {
    let foo = Foo;
    let ref_foo = &&&&&Foo;

    // error, reached the recursion limit while auto-dereferencing
`&&&&&Foo`
    ref_foo.foo();
}
```

One fix may be to increase the recursion limit. Note that it is possible to create an infinite recursion of dereferencing, in which case the only fix is to somehow break the recursion.

Error code E0057

An invalid number of arguments was given when calling a closure.

Erroneous code example:

```
let f = |x| x * 3;
let a = f();        // invalid, too few parameters
let b = f(4);       // this works!
let c = f(2, 3);    // invalid, too many parameters
```

When invoking closures or other implementations of the function traits `Fn`, `FnMut` or `FnOnce` using call notation, the number of parameters passed to the function must match its definition.

A generic function must be treated similarly:

```
fn foo<F: Fn()>(f: F) {
    f(); // this is valid, but f(3) would not work
}
```

Error code E0059

The built-in function traits are generic over a tuple of the function arguments. If one uses angle-bracket notation (`Fn<(T,), Output=U>`) instead of parentheses (`Fn(T) -> U`) to denote the function trait, the type parameter should be a tuple. Otherwise function call notation cannot be used and the trait will not be implemented by closures.

The most likely source of this error is using angle-bracket notation without wrapping the function argument type into a tuple, for example:

```
#![feature(unboxed_closures)]

fn foo<F: Fn<i32>>(f: F) -> F::Output { f(3) }
```

It can be fixed by adjusting the trait bound like this:

```
#![feature(unboxed_closures)]

fn foo<F: Fn<(i32,)>>(f: F) -> F::Output { f(3) }
```

Note that `(T,)` always denotes the type of a 1-tuple containing an element of type `T` . The comma is necessary for syntactic disambiguation.

Error code E0060

External C functions are allowed to be variadic. However, a variadic function takes a minimum number of arguments. For example, consider C's variadic `printf` function:

```
use std::os::raw::{c_char, c_int};

extern "C" {
    fn printf(_: *const c_char, ...) -> c_int;
}

unsafe { printf(); } // error!
```

Using this declaration, it must be called with at least one argument, so simply calling `printf()` is invalid. But the following uses are allowed:

```
unsafe {
    use std::ffi::CString;

    let fmt = CString::new("test\n").unwrap();
    printf(fmt.as_ptr());

    let fmt = CString::new("number = %d\n").unwrap();
    printf(fmt.as_ptr(), 3);

    let fmt = CString::new("%d, %d\n").unwrap();
    printf(fmt.as_ptr(), 10, 5);
}
```

Error code E0061

An invalid number of arguments was passed when calling a function.

Erroneous code example:

```
fn f(u: i32) {}

f(); // error!
```

The number of arguments passed to a function must match the number of arguments specified in the function signature.

For example, a function like:

```
fn f(a: u16, b: &str) {}
```

Must always be called with exactly two arguments, e.g., `f(2, "test")` .

Note that Rust does not have a notion of optional function arguments or variadic functions (except for its C-FFI).

Error code E0062

A struct's or struct-like enum variant's field was specified more than once.

Erroneous code example:

```
struct Foo {
    x: i32,
}

fn main() {
    let x = Foo {
                x: 0,
                x: 0, // error: field `x` specified more than once
            };
}
```

This error indicates that during an attempt to build a struct or struct-like enum variant, one of the fields was specified more than once. Each field should be specified exactly one time. Example:

```
struct Foo {
    x: i32,
}

fn main() {
    let x = Foo { x: 0 }; // ok!
}
```

Error code E0063

A struct's or struct-like enum variant's field was not provided.

Erroneous code example:

```
struct Foo {
    x: i32,
    y: i32,
}

fn main() {
    let x = Foo { x: 0 }; // error: missing field: `y`
}
```

Each field should be specified exactly once. Example:

```
struct Foo {
    x: i32,
    y: i32,
}

fn main() {
    let x = Foo { x: 0, y: 0 }; // ok!
}
```

Error code E0067

An invalid left-hand side expression was used on an assignment operation.

Erroneous code example:

```
12 += 1; // error!
```

You need to have a place expression to be able to assign it something. For example:

```
let mut x: i8 = 12;
x += 1; // ok!
```

Error code E0069

The compiler found a function whose body contains a `return;` statement but whose return type is not `()`.

Erroneous code example:

```
// error
fn foo() -> u8 {
    return;
}
```

Since `return;` is just like `return ();`, there is a mismatch between the function's return type and the value being returned.

Error code E0070

An assignment operator was used on a non-place expression.

Erroneous code examples:

```
struct SomeStruct {
    x: i32,
    y: i32,
}

const SOME_CONST: i32 = 12;

fn some_other_func() {}

fn some_function() {
    SOME_CONST = 14; // error: a constant value cannot be changed!
    1 = 3; // error: 1 isn't a valid place!
    some_other_func() = 4; // error: we cannot assign value to a function!
    SomeStruct::x = 12; // error: SomeStruct a structure name but it is used
                        //        like a variable!
}
```

The left-hand side of an assignment operator must be a place expression. A place expression represents a memory location and can be a variable (with optional namespacing), a dereference, an indexing expression or a field reference.

More details can be found in the Expressions section of the Reference.

And now let's give working examples:

```
struct SomeStruct {
    x: i32,
    y: i32,
}
let mut s = SomeStruct { x: 0, y: 0 };

s.x = 3; // that's good !

// ...

fn some_func(x: &mut i32) {
    *x = 12; // that's good !
}
```

Error code E0071

A structure-literal syntax was used to create an item that is not a structure or enum variant.

Example of erroneous code:

```
type U32 = u32;
let t = U32 { value: 4 }; // error: expected struct, variant or union type,
                          // found builtin type `u32`
```

To fix this, ensure that the name was correctly spelled, and that the correct form of initializer was used.

For example, the code above can be fixed to:

```
type U32 = u32;
let t: U32 = 4;
```

or:

```
struct U32 { value: u32 }
let t = U32 { value: 4 };
```

Error code E0072

A recursive type has infinite size because it doesn't have an indirection.

Erroneous code example:

```
struct ListNode {
    head: u8,
    tail: Option<ListNode>, // error: no indirection here so impossible to
                            //        compute the type's size
}
```

When defining a recursive struct or enum, any use of the type being defined from inside the definition must occur behind a pointer (like `Box`, `&` or `Rc`). This is because structs and enums must have a well-defined size, and without the pointer, the size of the type would need to be unbounded.

In the example, the type cannot have a well-defined size, because it needs to be arbitrarily large (since we would be able to nest `ListNode`s to any depth). Specifically,

```
size of `ListNode` = 1 byte for `head`
                   + 1 byte for the discriminant of the `Option`
                   + size of `ListNode`
```

One way to fix this is by wrapping `ListNode` in a `Box`, like so:

```
struct ListNode {
    head: u8,
    tail: Option<Box<ListNode>>,
}
```

This works because `Box` is a pointer, so its size is well-known.

Error code E0073

Note: this error code is no longer emitted by the compiler.

You cannot define a struct (or enum) `Foo` that requires an instance of `Foo` in order to make a new `Foo` value. This is because there would be no way a first instance of `Foo` could be made to initialize another instance!

Here's an example of a struct that has this problem:

```
struct Foo { x: Box<Foo> } // error
```

One fix is to use `Option`, like so:

```
struct Foo { x: Option<Box<Foo>> }
```

Now it's possible to create at least one instance of `Foo`: `Foo { x: None }`.

Error code E0074

Note: this error code is no longer emitted by the compiler.

When using the `#[simd]` attribute on a tuple struct, the components of the tuple struct must all be of a concrete, nongeneric type so the compiler can reason about how to use SIMD with them. This error will occur if the types are generic.

This will cause an error:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Bad<T>(T, T, T, T);
```

This will not:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Good(u32, u32, u32, u32);
```

Error code E0075

A `#[simd]` attribute was applied to an empty tuple struct.

Erroneous code example:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Bad; // error!
```

The `#[simd]` attribute can only be applied to non empty tuple structs, because it doesn't make sense to try to use SIMD operations when there are no values to operate on.

Fixed example:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Good(u32); // ok!
```

Error code E0076

All types in a tuple struct aren't the same when using the `#[simd]` attribute.

Erroneous code example:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Bad(u16, u32, u32 u32); // error!
```

When using the `#[simd]` attribute to automatically use SIMD operations in tuple struct, the types in the struct must all be of the same type, or the compiler will trigger this error.

Fixed example:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Good(u32, u32, u32, u32); // ok!
```

Error code E0077

A tuple struct's element isn't a machine type when using the `#[simd]` attribute.

Erroneous code example:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Bad(String); // error!
```

When using the `#[simd]` attribute on a tuple struct, the elements in the tuple must be machine types so SIMD operations can be applied to them.

Fixed example:

```
#![feature(repr_simd)]

#[repr(simd)]
struct Good(u32, u32, u32, u32); // ok!
```

Error code E0080

A constant value failed to get evaluated.

Erroneous code example:

```
enum Enum {
    X = (1 << 500),
    Y = (1 / 0),
}
```

This error indicates that the compiler was unable to sensibly evaluate a constant expression that had to be evaluated. Attempting to divide by 0 or causing an integer overflow are two ways to induce this error.

Ensure that the expressions given can be evaluated as the desired integer type.

See the Discriminants section of the Reference for more information about setting custom integer types on enums using the `repr` attribute.

Error code E0081

A discriminant value is present more than once.

Erroneous code example:

```
enum Enum {
    P = 3,
    X = 3, // error!
    Y = 5,
}
```

Enum discriminants are used to differentiate enum variants stored in memory. This error indicates that the same value was used for two or more variants, making it impossible to distinguish them.

```
enum Enum {
    P,
    X = 3, // ok!
    Y = 5,
}
```

Note that variants without a manually specified discriminant are numbered from top to bottom starting from 0, so clashes can occur with seemingly unrelated variants.

```
enum Bad {
    X,
    Y = 0, // error!
}
```

Here `X` will have already been specified the discriminant 0 by the time `Y` is encountered, so a conflict occurs.

Error code E0084

An unsupported representation was attempted on a zero-variant enum.

Erroneous code example:

```
#[repr(i32)]
enum NightsWatch {} // error: unsupported representation for zero-variant
enum
```

It is impossible to define an integer type to be used to represent zero-variant enum values because there are no zero-variant enum values. There is no way to construct an instance of the following type using only safe code. So you have two solutions. Either you add variants in your enum:

```
#[repr(i32)]
enum NightsWatch {
    JonSnow,
    Commander,
}
```

or you remove the integer representation of your enum:

```
enum NightsWatch {}
```

Error code E0087

Note: this error code is no longer emitted by the compiler.

Too many type arguments were supplied for a function. For example:

```
fn foo<T>() {}

fn main() {
    foo::<f64, bool>(); // error: wrong number of type arguments:
                        //        expected 1, found 2
}
```

The number of supplied arguments must exactly match the number of defined type parameters.

Error code E0088

Note: this error code is no longer emitted by the compiler.

You gave too many lifetime arguments. Erroneous code example:

```
fn f() {}

fn main() {
    f::<'static>() // error: wrong number of lifetime arguments:
                   //        expected 0, found 1
}
```

Please check you give the right number of lifetime arguments. Example:

```
fn f() {}

fn main() {
    f() // ok!
}
```

It's also important to note that the Rust compiler can generally determine the lifetime by itself. Example:

```
struct Foo {
    value: String
}

impl Foo {
    // it can be written like this
    fn get_value<'a>(&'a self) -> &'a str { &self.value }
    // but the compiler works fine with this too:
    fn without_lifetime(&self) -> &str { &self.value }
}

fn main() {
    let f = Foo { value: "hello".to_owned() };

    println!("{}", f.get_value());
    println!("{}", f.without_lifetime());
}
```

Error code E0089

Note: this error code is no longer emitted by the compiler.

Too few type arguments were supplied for a function. For example:

```
fn foo<T, U>() {}

fn main() {
    foo::<f64>(); // error: wrong number of type arguments: expected 2,
found 1
}
```

Note that if a function takes multiple type arguments but you want the compiler to infer some of them, you can use type placeholders:

```
fn foo<T, U>(x: T) {}

fn main() {
    let x: bool = true;
    foo::<f64>(x);    // error: wrong number of type arguments:
                      //        expected 2, found 1
    foo::<_, f64>(x); // same as `foo::<bool, f64>(x)`
}
```

Error code E0090

Note: this error code is no longer emitted by the compiler.

You gave too few lifetime arguments. Example:

```
fn foo<'a: 'b, 'b: 'a>() {}

fn main() {
    foo::<'static>(); // error: wrong number of lifetime arguments:
                      //        expected 2, found 1
}
```

Please check you give the right number of lifetime arguments. Example:

```
fn foo<'a: 'b, 'b: 'a>() {}

fn main() {
    foo::<'static, 'static>();
}
```

Error code E0091

An unnecessary type or const parameter was given in a type alias.

Erroneous code example:

```
type Foo<T> = u32; // error: type parameter `T` is unused
// or:
type Foo<A,B> = Box<A>; // error: type parameter `B` is unused
```

Please check you didn't write too many parameters. Example:

```
type Foo = u32; // ok!
type Foo2<A> = Box<A>; // ok!
```

Error code E0092

An undefined atomic operation function was declared.

Erroneous code example:

```
#![feature(intrinsics)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    fn atomic_foo(); // error: unrecognized atomic operation
                     //        function
}
```

Please check you didn't make a mistake in the function's name. All intrinsic functions are defined in `compiler/rustc_codegen_llvm/src/intrinsic.rs` and in `library/core/src/intrinsics.rs` in the Rust source code. Example:

```
#![feature(intrinsics)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    fn atomic_fence_seqcst(); // ok!
}
```

Error code E0093

An unknown intrinsic function was declared.

Erroneous code example:

```
#![feature(intrinsics)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    fn foo(); // error: unrecognized intrinsic function: `foo`
}

fn main() {
    unsafe {
        foo();
    }
}
```

Please check you didn't make a mistake in the function's name. All intrinsic functions are defined in `compiler/rustc_codegen_llvm/src/intrinsic.rs` and in `library/core/src/intrinsics.rs` in the Rust source code. Example:

```
#![feature(intrinsics)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    fn atomic_fence_seqcst(); // ok!
}

fn main() {
    unsafe {
        atomic_fence_seqcst();
    }
}
```

Error code E0094

An invalid number of generic parameters was passed to an intrinsic function.

Erroneous code example:

```
#![feature(intrinsics, rustc_attrs)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    #[rustc_safe_intrinsic]
    fn size_of<T, U>() -> usize; // error: intrinsic has wrong number
                                 //        of type parameters
}
```

Please check that you provided the right number of type parameters and verify with the function declaration in the Rust source code. Example:

```
#![feature(intrinsics, rustc_attrs)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    #[rustc_safe_intrinsic]
    fn size_of<T>() -> usize; // ok!
}
```

Error code E0106

This error indicates that a lifetime is missing from a type. If it is an error inside a function signature, the problem may be with failing to adhere to the lifetime elision rules (see below).

Erroneous code examples:

```
struct Foo1 { x: &bool }
              // ^ expected lifetime parameter
struct Foo2<'a> { x: &'a bool } // correct

struct Bar1 { x: Foo2 }
              // ^^^^ expected lifetime parameter
struct Bar2<'a> { x: Foo2<'a> } // correct

enum Baz1 { A(u8), B(&bool), }
                  // ^ expected lifetime parameter
enum Baz2<'a> { A(u8), B(&'a bool), } // correct

type MyStr1 = &str;
           // ^ expected lifetime parameter
type MyStr2<'a> = &'a str; // correct
```

Lifetime elision is a special, limited kind of inference for lifetimes in function signatures which allows you to leave out lifetimes in certain cases. For more background on lifetime elision see the book.

The lifetime elision rules require that any function signature with an elided output lifetime must either have:

- exactly one input lifetime
- or, multiple input lifetimes, but the function must also be a method with a `&self` or `&mut self` receiver

In the first case, the output lifetime is inferred to be the same as the unique input lifetime. In the second case, the lifetime is instead inferred to be the same as the lifetime on `&self` or `&mut self`.

Here are some examples of elision errors:

```
// error, no input lifetimes
fn foo() -> &str { }

// error, `x` and `y` have distinct lifetimes inferred
fn bar(x: &str, y: &str) -> &str { }

// error, `y`'s lifetime is inferred to be distinct from `x`'s
fn baz<'a>(x: &'a str, y: &str) -> &str { }
```

Error code E0107

An incorrect number of generic arguments was provided.

Erroneous code example:

```
struct Foo<T> { x: T }

struct Bar { x: Foo }             // error: wrong number of type arguments:
                                  //        expected 1, found 0
struct Baz<S, T> { x: Foo<S, T> } // error: wrong number of type arguments:
                                  //        expected 1, found 2

fn foo<T, U>(x: T, y: U) {}
fn f() {}

fn main() {
    let x: bool = true;
    foo::<bool>(x);                 // error: wrong number of type
arguments:
                                    //        expected 2, found 1
    foo::<bool, i32, i32>(x, 2, 4); // error: wrong number of type
arguments:
                                    //        expected 2, found 3
    f::<'static>();                 // error: wrong number of lifetime
arguments
                                    //        expected 0, found 1
}
```

When using/declaring an item with generic arguments, you must provide the exact same number:

```
struct Foo<T> { x: T }

struct Bar<T> { x: Foo<T> }               // ok!
struct Baz<S, T> { x: Foo<S>, y: Foo<T> } // ok!

fn foo<T, U>(x: T, y: U) {}
fn f() {}

fn main() {
    let x: bool = true;
    foo::<bool, u32>(x, 12);              // ok!
    f();                                  // ok!
}
```

Error code E0109

You tried to provide a generic argument to a type which doesn't need it.

Erroneous code example:

```
type X = u32<i32>; // error: type arguments are not allowed for this type
type Y = bool<'static>; // error: lifetime parameters are not allowed on
                        //        this type
```

Check that you used the correct argument and that the definition is correct.

Example:

```
type X = u32; // ok!
type Y = bool; // ok!
```

Note that generic arguments for enum variant constructors go after the variant, not after the enum. For example, you would write `Option::None::<u32>` , rather than `Option::<u32>::None` .

Error code E0110

Note: this error code is no longer emitted by the compiler.

You tried to provide a lifetime to a type which doesn't need it. See `E0109` for more details.

Error code E0116

An inherent implementation was defined for a type outside the current crate.

Erroneous code example:

```
impl Vec<u8> { } // error
```

You can only define an inherent implementation for a type in the same crate where the type was defined. For example, an `impl` block as above is not allowed since `Vec` is defined in the standard library.

To fix this problem, you can either:

- define a trait that has the desired associated functions/types/constants and implement the trait for the type in question
- define a new type wrapping the type and define an implementation on the new type

Note that using the `type` keyword does not work here because `type` only introduces a type alias:

```
type Bytes = Vec<u8>;

impl Bytes { } // error, same as above
```

Error code E0117

Only traits defined in the current crate can be implemented for arbitrary types.

Erroneous code example:

```
impl Drop for u32 {}
```

This error indicates a violation of one of Rust's orphan rules for trait implementations. The rule prohibits any implementation of a foreign trait (a trait defined in another crate) where

- the type that is implementing the trait is foreign
- all of the parameters being passed to the trait (if there are any) are also foreign.

To avoid this kind of error, ensure that at least one local type is referenced by the `impl`:

```
pub struct Foo; // you define your type in your crate

impl Drop for Foo { // and you can implement the trait on it!
    // code of trait implementation here
}

impl From<Foo> for i32 { // or you use a type from your crate as
                         // a type parameter
    fn from(i: Foo) -> i32 {
        0
    }
}
```

Alternatively, define a trait locally and implement that instead:

```
trait Bar {
    fn get(&self) -> usize;
}

impl Bar for u32 {
    fn get(&self) -> usize { 0 }
}
```

For information on the design of the orphan rules, see RFC 1023.

Error code E0118

An inherent implementation was defined for something which isn't a struct, enum, union, or trait object.

Erroneous code example:

```
impl<T> T { // error: no nominal type found for inherent implementation
    fn get_state(&self) -> String {
        // ...
    }
}
```

To fix this error, please implement a trait on the type or wrap it in a struct. Example:

```
// we create a trait here
trait LiveLongAndProsper {
    fn get_state(&self) -> String;
}

// and now you can implement it on T
impl<T> LiveLongAndProsper for T {
    fn get_state(&self) -> String {
        "He's dead, Jim!".to_owned()
    }
}
```

Alternatively, you can create a newtype. A newtype is a wrapping tuple-struct. For example, `NewType` is a newtype over `Foo` in `struct NewType(Foo)`. Example:

```
struct TypeWrapper<T>(T);

impl<T> TypeWrapper<T> {
    fn get_state(&self) -> String {
        "Fascinating!".to_owned()
    }
}
```

Error code E0119

There are conflicting trait implementations for the same type.

Erroneous code example:

```
trait MyTrait {
    fn get(&self) -> usize;
}

impl<T> MyTrait for T {
    fn get(&self) -> usize { 0 }
}

struct Foo {
    value: usize
}

impl MyTrait for Foo { // error: conflicting implementations of trait
                       //        `MyTrait` for type `Foo`
    fn get(&self) -> usize { self.value }
}
```

When looking for the implementation for the trait, the compiler finds both the `impl<T> MyTrait for T` where T is all types and the `impl MyTrait for Foo` . Since a trait cannot be implemented multiple times, this is an error. So, when you write:

```
trait MyTrait {
    fn get(&self) -> usize;
}

impl<T> MyTrait for T {
    fn get(&self) -> usize { 0 }
}
```

This makes the trait implemented on all types in the scope. So if you try to implement it on another one after that, the implementations will conflict. Example:

```
trait MyTrait {
    fn get(&self) -> usize;
}

impl<T> MyTrait for T {
    fn get(&self) -> usize { 0 }
}

struct Foo;

fn main() {
    let f = Foo;

    f.get(); // the trait is implemented so we can use it
}
```

Error code E0120

Drop was implemented on a trait, which is not allowed: only structs and enums can implement Drop.

Erroneous code example:

```
trait MyTrait {}

impl Drop for MyTrait {
    fn drop(&mut self) {}
}
```

A workaround for this problem is to wrap the trait up in a struct, and implement Drop on that:

```
trait MyTrait {}
struct MyWrapper<T: MyTrait> { foo: T }

impl <T: MyTrait> Drop for MyWrapper<T> {
    fn drop(&mut self) {}
}
```

Alternatively, wrapping trait objects requires something:

```
trait MyTrait {}

//or Box<MyTrait>, if you wanted an owned trait object
struct MyWrapper<'a> { foo: &'a MyTrait }

impl <'a> Drop for MyWrapper<'a> {
    fn drop(&mut self) {}
}
```

Error code E0121

The type placeholder `_` was used within a type on an item's signature.

Erroneous code example:

```
fn foo() -> _ { 5 } // error

static BAR: _ = "test"; // error
```

In those cases, you need to provide the type explicitly:

```
fn foo() -> i32 { 5 } // ok!

static BAR: &str = "test"; // ok!
```

The type placeholder `_` can be used outside item's signature as follows:

```
let x = "a4a".split('4')
    .collect::<Vec<_>>(); // No need to precise the Vec's generic type.
```

Error code E0124

A struct was declared with two fields having the same name.

Erroneous code example:

```
struct Foo {
    field1: i32,
    field1: i32, // error: field is already declared
}
```

Please verify that the field names have been correctly spelled. Example:

```
struct Foo {
    field1: i32,
    field2: i32, // ok!
}
```

Error code E0128

A type parameter with default value is using forward declared identifier.

Erroneous code example:

```
struct Foo<T = U, U = ()> {
    field1: T,
    field2: U,
}
// error: generic parameters with a default cannot use forward declared
//        identifiers
```

Type parameter defaults can only use parameters that occur before them. Since type parameters are evaluated in-order, this issue could be fixed by doing:

```
struct Foo<U = (), T = U> {
    field1: T,
    field2: U,
}
```

Please also verify that this wasn't because of a name-clash and rename the type parameter if so.

Error code E0130

A pattern was declared as an argument in a foreign function declaration.

Erroneous code example:

```
extern "C" {
    fn foo((a, b): (u32, u32)); // error: patterns aren't allowed in foreign
                                //        function declarations
}
```

To fix this error, replace the pattern argument with a regular one. Example:

```
struct SomeStruct {
    a: u32,
    b: u32,
}

extern "C" {
    fn foo(s: SomeStruct); // ok!
}
```

Or:

```
extern "C" {
    fn foo(a: (u32, u32)); // ok!
}
```

Error code E0131

The `main` function was defined with generic parameters.

Erroneous code example:

```
fn main<T>() { // error: main function is not allowed to have generic
parameters
}
```

It is not possible to define the `main` function with generic parameters. It must not take any arguments.

Error code E0132

A function with the `start` attribute was declared with type parameters.

Erroneous code example:

```
#![feature(start)]

#[start]
fn f<T>() {}
```

It is not possible to declare type parameters on a function that has the `start` attribute. Such a function must have the following type signature (for more information, view the unstable book):

```
fn(isize, *const *const u8) -> isize;
```

Example:

```
#![feature(start)]

#[start]
fn my_start(argc: isize, argv: *const *const u8) -> isize {
    0
}
```

Error code E0133

Unsafe code was used outside of an unsafe block.

Erroneous code example:

```
unsafe fn f() { return; } // This is the unsafe code

fn main() {
    f(); // error: call to unsafe function requires unsafe function or block
}
```

Using unsafe functionality is potentially dangerous and disallowed by safety checks. Examples:

- Dereferencing raw pointers
- Calling functions via FFI
- Calling functions marked unsafe

These safety checks can be relaxed for a section of the code by wrapping the unsafe instructions with an `unsafe` block. For instance:

```
unsafe fn f() { return; }

fn main() {
    unsafe { f(); } // ok!
}
```

See the unsafe section of the Book for more details.

Unsafe code in functions

Unsafe code is currently accepted in unsafe functions, but that is being phased out in favor of requiring unsafe blocks here too.

```
unsafe fn f() { return; }

unsafe fn g() {
    f(); // Is accepted, but no longer recommended
    unsafe { f(); } // Recommended way to write this
}
```

Linting against this is controlled via the `unsafe_op_in_unsafe_fn` lint, which is `allow` by default but will be upgraded to `warn` in a future edition.

Error code E0136

Note: this error code is no longer emitted by the compiler.

More than one `main` function was found.

Erroneous code example:

```
fn main() {
    // ...
}

// ...

fn main() { // error!
    // ...
}
```

A binary can only have one entry point, and by default that entry point is the `main()` function. If there are multiple instances of this function, please rename one of them.

Error code E0137

Note: this error code is no longer emitted by the compiler.

More than one function was declared with the `#[main]` attribute.

Erroneous code example:

```
#![feature(main)]

#[main]
fn foo() {}

#[main]
fn f() {} // error: multiple functions with a `#[main]` attribute
```

This error indicates that the compiler found multiple functions with the `#[main]` attribute. This is an error because there must be a unique entry point into a Rust program. Example:

```
#![feature(main)]

#[main]
fn f() {} // ok!
```

Error code E0138

More than one function was declared with the `#[start]` attribute.

Erroneous code example:

```
#![feature(start)]

#[start]
fn foo(argc: isize, argv: *const *const u8) -> isize {}

#[start]
fn f(argc: isize, argv: *const *const u8) -> isize {}
// error: multiple 'start' functions
```

This error indicates that the compiler found multiple functions with the `#[start]` attribute. This is an error because there must be a unique entry point into a Rust program. Example:

```
#![feature(start)]

#[start]
fn foo(argc: isize, argv: *const *const u8) -> isize { 0 } // ok!
```

Error code E0139

Note: this error code is no longer emitted by the compiler.

There are various restrictions on transmuting between types in Rust; for example types being transmuted must have the same size. To apply all these restrictions, the compiler must know the exact types that may be transmuted. When type parameters are involved, this cannot always be done.

So, for example, the following is not allowed:

```rust
use std::mem::transmute;

struct Foo<T>(Vec<T>);

fn foo<T>(x: Vec<T>) {
    // we are transmuting between Vec<T> and Foo<F> here
    let y: Foo<T> = unsafe { transmute(x) };
    // do something with y
}
```

In this specific case there's a good chance that the transmute is harmless (but this is not guaranteed by Rust). However, when alignment and enum optimizations come into the picture, it's quite likely that the sizes may or may not match with different type parameter substitutions. It's not possible to check this for *all* possible types, so `transmute()` simply only accepts types without any unsubstituted type parameters.

If you need this, there's a good chance you're doing something wrong. Keep in mind that Rust doesn't guarantee much about the layout of different structs (even two structs with identical declarations may have different layouts). If there is a solution that avoids the transmute entirely, try it instead.

If it's possible, hand-monomorphize the code by writing the function for each possible type substitution. It's possible to use traits to do this cleanly, for example:

```
use std::mem::transmute;

struct Foo<T>(Vec<T>);

trait MyTransmutableType: Sized {
    fn transmute(_: Vec<Self>) -> Foo<Self>;
}

impl MyTransmutableType for u8 {
    fn transmute(x: Vec<u8>) -> Foo<u8> {
        unsafe { transmute(x) }
    }
}

impl MyTransmutableType for String {
    fn transmute(x: Vec<String>) -> Foo<String> {
        unsafe { transmute(x) }
    }
}

// ... more impls for the types you intend to transmute

fn foo<T: MyTransmutableType>(x: Vec<T>) {
    let y: Foo<T> = <T as MyTransmutableType>::transmute(x);
    // do something with y
}
```

Each impl will be checked for a size match in the transmute as usual, and since there are no unbound type parameters involved, this should compile unless there is a size mismatch in one of the impls.

It is also possible to manually transmute:

```
unsafe {
    ptr::read(&v as *const _ as *const SomeType) // `v` transmuted to
`SomeType`
}
```

Note that this does not move `v` (unlike `transmute`), and may need a call to `mem::forget(v)` in case you want to avoid destructors being called.

Error code E0152

A lang item was redefined.

Erroneous code example:

```
#![feature(lang_items)]

#[lang = "owned_box"]
struct Foo<T>(T); // error: duplicate lang item found: `owned_box`
```

Lang items are already implemented in the standard library. Unless you are writing a free-standing application (e.g., a kernel), you do not need to provide them yourself.

You can build a free-standing crate by adding `#![no_std]` to the crate attributes:

```
#![no_std]
```

See also this section of the Rustonomicon.

Error code E0154

Note: this error code is no longer emitted by the compiler.

Imports (`use` statements) are not allowed after non-item statements, such as variable declarations and expression statements.

Here is an example that demonstrates the error:

```
fn f() {
    // Variable declaration before import
    let x = 0;
    use std::io::Read;
    // ...
}
```

The solution is to declare the imports at the top of the block, function, or file.

Here is the previous example again, with the correct order:

```
fn f() {
    use std::io::Read;
    let x = 0;
    // ...
}
```

See the Declaration Statements section of the reference for more information about what constitutes an item declaration and what does not.

Error code E0158

An associated `const`, `const` parameter or `static` has been referenced in a pattern.

Erroneous code example:

```
enum Foo {
    One,
    Two
}

trait Bar {
    const X: Foo;
}

fn test<A: Bar>(arg: Foo) {
    match arg {
        A::X => println!("A::X"), // error: E0158: associated consts cannot
be
                                  //        referenced in patterns
        Foo::Two => println!("Two")
    }
}
```

Associated `const`s cannot be referenced in patterns because it is impossible for the compiler to prove exhaustiveness (that some pattern will always match). Take the above example, because Rust does type checking in the *generic* method, not the *monomorphized* specific instance. So because `Bar` could have theoretically infinite implementations, there's no way to always be sure that `A::X` is `Foo::One`. So this code must be rejected. Even if code can be proven exhaustive by a programmer, the compiler cannot currently prove this.

The same holds true of `const` parameters and `static`s.

If you want to match against an associated `const`, `const` parameter or `static` consider using a guard instead:

```
trait Trait {
    const X: char;
}

static FOO: char = 'j';

fn test<A: Trait, const Y: char>(arg: char) {
    match arg {
        c if c == A::X => println!("A::X"),
        c if c == Y => println!("Y"),
        c if c == FOO => println!("FOO"),
        _ => ()
    }
}
```

Error code E0161

A value was moved whose size was not known at compile time.

Erroneous code example:

```
trait Bar {
    fn f(self);
}

impl Bar for i32 {
    fn f(self) {}
}

fn main() {
    let b: Box<dyn Bar> = Box::new(0i32);
    b.f();
    // error: cannot move a value of type dyn Bar: the size of dyn Bar
cannot
    //        be statically determined
}
```

In Rust, you can only move a value when its size is known at compile time.

To work around this restriction, consider "hiding" the value behind a reference: either `&x` or `&mut x`. Since a reference has a fixed size, this lets you move it around as usual. Example:

```
trait Bar {
    fn f(&self);
}

impl Bar for i32 {
    fn f(&self) {}
}

fn main() {
    let b: Box<dyn Bar> = Box::new(0i32);
    b.f();
    // ok!
}
```

Error code E0162

Note: this error code is no longer emitted by the compiler.

An `if let` pattern attempts to match the pattern, and enters the body if the match was successful. If the match is irrefutable (when it cannot fail to match), use a regular `let`-binding instead. For instance:

```
struct Irrefutable(i32);
let irr = Irrefutable(0);

// This fails to compile because the match is irrefutable.
if let Irrefutable(x) = irr {
    // This body will always be executed.
    // ...
}
```

Try this instead:

```
struct Irrefutable(i32);
let irr = Irrefutable(0);

let Irrefutable(x) = irr;
println!("{}", x);
```

Error code E0164

Something which is neither a tuple struct nor a tuple variant was used as a pattern.

Erroneous code example:

```
enum A {
    B,
    C,
}

impl A {
    fn new() {}
}

fn bar(foo: A) {
    match foo {
        A::new() => (), // error!
        _ => {}
    }
}
```

This error means that an attempt was made to match something which is neither a tuple struct nor a tuple variant. Only these two elements are allowed as a pattern:

```
enum A {
    B,
    C,
}

impl A {
    fn new() {}
}

fn bar(foo: A) {
    match foo {
        A::B => (), // ok!
        _ => {}
    }
}
```

Error code E0165

Note: this error code is no longer emitted by the compiler.

A `while let` pattern attempts to match the pattern, and enters the body if the match was successful. If the match is irrefutable (when it cannot fail to match), use a regular `let`-binding inside a `loop` instead. For instance:

```
struct Irrefutable(i32);
let irr = Irrefutable(0);

// This fails to compile because the match is irrefutable.
while let Irrefutable(x) = irr {
    // ...
}
```

Try this instead:

```
struct Irrefutable(i32);
let irr = Irrefutable(0);

loop {
    let Irrefutable(x) = irr;
    // ...
}
```

Error code E0170

A pattern binding is using the same name as one of the variants of a type.

Erroneous code example:

```
enum Method {
    GET,
    POST,
}

fn is_empty(s: Method) -> bool {
    match s {
        GET => true,
        _ => false
    }
}

fn main() {}
```

Enum variants are qualified by default. For example, given this type:

```
enum Method {
    GET,
    POST,
}
```

You would match it using:

```
enum Method {
    GET,
    POST,
}

let m = Method::GET;

match m {
    Method::GET => {},
    Method::POST => {},
}
```

If you don't qualify the names, the code will bind new variables named "GET" and "POST" instead. This behavior is likely not what you want, so `rustc` warns when that happens.

Qualified names are good practice, and most code works well with them. But if you prefer them unqualified, you can import the variants into scope:

```
use Method::*;
enum Method { GET, POST }
```

If you want others to be able to import variants from your module directly, use `pub use`:

```
pub use Method::*;
pub enum Method { GET, POST }
```

Error code E0178

The `+` type operator was used in an ambiguous context.

Erroneous code example:

```
trait Foo {}

struct Bar<'a> {
    x: &'a Foo + 'a,     // error!
    y: &'a mut Foo + 'a, // error!
    z: fn() -> Foo + 'a, // error!
}
```

In types, the `+` type operator has low precedence, so it is often necessary to use parentheses:

```
trait Foo {}

struct Bar<'a> {
    x: &'a (Foo + 'a),     // ok!
    y: &'a mut (Foo + 'a), // ok!
    z: fn() -> (Foo + 'a), // ok!
}
```

More details can be found in RFC 438.

Error code E0183

Manual implementation of a `Fn*` trait.

Erroneous code example:

```
struct MyClosure {
    foo: i32
}

impl FnOnce<()> for MyClosure {  // error
    type Output = ();
    extern "rust-call" fn call_once(self, args: ()) -> Self::Output {
        println!("{}", self.foo);
    }
}
```

Manually implementing `Fn`, `FnMut` or `FnOnce` is unstable and requires `#![feature(fn_traits, unboxed_closures)]`.

```
#![feature(fn_traits, unboxed_closures)]

struct MyClosure {
    foo: i32
}

impl FnOnce<()> for MyClosure {  // ok!
    type Output = ();
    extern "rust-call" fn call_once(self, args: ()) -> Self::Output {
        println!("{}", self.foo);
    }
}
```

The arguments must be a tuple representing the argument list. For more info, see the tracking issue:

Error code E0184

The `Copy` trait was implemented on a type with a `Drop` implementation.

Erroneous code example:

```
#[derive(Copy)]
struct Foo; // error!

impl Drop for Foo {
    fn drop(&mut self) {
    }
}
```

Explicitly implementing both `Drop` and `Copy` trait on a type is currently disallowed. This feature can make some sense in theory, but the current implementation is incorrect and can lead to memory unsafety (see issue #20126), so it has been disabled for now.

Error code E0185

An associated function for a trait was defined to be static, but an implementation of the trait declared the same function to be a method (i.e., to take a `self` parameter).

Erroneous code example:

```
trait Foo {
    fn foo();
}

struct Bar;

impl Foo for Bar {
    // error, method `foo` has a `&self` declaration in the impl, but not in
    // the trait
    fn foo(&self) {}
}
```

When a type implements a trait's associated function, it has to use the same signature. So in this case, since `Foo::foo` does not take any argument and does not return anything, its implementation on `Bar` should be the same:

```
trait Foo {
    fn foo();
}

struct Bar;

impl Foo for Bar {
    fn foo() {} // ok!
}
```

Error code E0186

An associated function for a trait was defined to be a method (i.e., to take a `self` parameter), but an implementation of the trait declared the same function to be static.

Erroneous code example:

```
trait Foo {
    fn foo(&self);
}

struct Bar;

impl Foo for Bar {
    // error, method `foo` has a `&self` declaration in the trait, but not
in
    // the impl
    fn foo() {}
}
```

When a type implements a trait's associated function, it has to use the same signature. So in this case, since `Foo::foo` takes `self` as argument and does not return anything, its implementation on `Bar` should be the same:

```
trait Foo {
    fn foo(&self);
}

struct Bar;

impl Foo for Bar {
    fn foo(&self) {} // ok!
}
```

Error code E0191

An associated type wasn't specified for a trait object.

Erroneous code example:

```
trait Trait {
    type Bar;
}

type Foo = dyn Trait; // error: the value of the associated type `Bar` (from
                      //        the trait `Trait`) must be specified
```

Trait objects need to have all associated types specified. Please verify that all associated types of the trait were specified and the correct trait was used. Example:

```
trait Trait {
    type Bar;
}

type Foo = dyn Trait<Bar=i32>; // ok!
```

Error code E0192

Note: this error code is no longer emitted by the compiler.

A negative impl was added on a trait implementation.

Erroneous code example:

```
trait Trait {
    type Bar;
}

struct Foo;

impl !Trait for Foo { } //~ ERROR

fn main() {}
```

Negative impls are only allowed for auto traits. For more information see the opt-in builtin traits RFC.

Error code E0193

Note: this error code is no longer emitted by the compiler.

`where` clauses must use generic type parameters: it does not make sense to use them otherwise. An example causing this error:

```
trait Foo {
    fn bar(&self);
}

#[derive(Copy,Clone)]
struct Wrapper<T> {
    Wrapped: T
}

impl Foo for Wrapper<u32> where Wrapper<u32>: Clone {
    fn bar(&self) { }
}
```

This use of a `where` clause is strange - a more common usage would look something like the following:

```
trait Foo {
    fn bar(&self);
}

#[derive(Copy,Clone)]
struct Wrapper<T> {
    Wrapped: T
}
impl <T> Foo for Wrapper<T> where Wrapper<T>: Clone {
    fn bar(&self) { }
}
```

Here, we're saying that the implementation exists on Wrapper only when the wrapped type `T` implements `Clone`. The `where` clause is important because some types will not implement `Clone`, and thus will not get this method.

In our erroneous example, however, we're referencing a single concrete type. Since we know for certain that `Wrapper<u32>` implements `Clone`, there's no reason to also specify it in a `where` clause.

Error code E0195

The lifetime parameters of the method do not match the trait declaration.

Erroneous code example:

```rust
trait Trait {
    fn bar<'a,'b:'a>(x: &'a str, y: &'b str);
}

struct Foo;

impl Trait for Foo {
    fn bar<'a,'b>(x: &'a str, y: &'b str) {
    // error: lifetime parameters or bounds on method `bar`
    // do not match the trait declaration
    }
}
```

The lifetime constraint `'b` for `bar()` implementation does not match the trait declaration. Ensure lifetime declarations match exactly in both trait declaration and implementation. Example:

```rust
trait Trait {
    fn t<'a,'b:'a>(x: &'a str, y: &'b str);
}

struct Foo;

impl Trait for Foo {
    fn t<'a,'b:'a>(x: &'a str, y: &'b str) { // ok!
    }
}
```

Error code E0197

An inherent implementation was marked unsafe.

Erroneous code example:

```
struct Foo;

unsafe impl Foo { } // error!
```

Inherent implementations (one that do not implement a trait but provide methods associated with a type) are always safe because they are not implementing an unsafe trait. Removing the `unsafe` keyword from the inherent implementation will resolve this error.

```
struct Foo;

impl Foo { } // ok!
```

Error code E0198

A negative implementation was marked as unsafe.

Erroneous code example:

```
struct Foo;

unsafe impl !Clone for Foo { } // error!
```

A negative implementation is one that excludes a type from implementing a particular trait. Not being able to use a trait is always a safe operation, so negative implementations are always safe and never need to be marked as unsafe.

This will compile:

```
#![feature(auto_traits)]

struct Foo;

auto trait Enterprise {}

impl !Enterprise for Foo { }
```

Please note that negative impls are only allowed for auto traits.

Error code E0199

A trait implementation was marked as unsafe while the trait is safe.

Erroneous code example:

```
struct Foo;

trait Bar { }

unsafe impl Bar for Foo { } // error!
```

Safe traits should not have unsafe implementations, therefore marking an implementation for a safe trait unsafe will cause a compiler error. Removing the unsafe marker on the trait noted in the error will resolve this problem:

```
struct Foo;

trait Bar { }

impl Bar for Foo { } // ok!
```

Error code E0200

An unsafe trait was implemented without an unsafe implementation.

Erroneous code example:

```
struct Foo;

unsafe trait Bar { }

impl Bar for Foo { } // error!
```

Unsafe traits must have unsafe implementations. This error occurs when an implementation for an unsafe trait isn't marked as unsafe. This may be resolved by marking the unsafe implementation as unsafe.

```
struct Foo;

unsafe trait Bar { }

unsafe impl Bar for Foo { } // ok!
```

Error code E0201

Two associated items (like methods, associated types, associated functions, etc.) were defined with the same identifier.

Erroneous code example:

```
struct Foo(u8);

impl Foo {
    fn bar(&self) -> bool { self.0 > 5 }
    fn bar() {} // error: duplicate associated function
}

trait Baz {
    type Quux;
    fn baz(&self) -> bool;
}

impl Baz for Foo {
    type Quux = u32;

    fn baz(&self) -> bool { true }

    // error: duplicate method
    fn baz(&self) -> bool { self.0 > 5 }

    // error: duplicate associated type
    type Quux = u32;
}
```

Note, however, that items with the same name are allowed for inherent `impl` blocks that don't overlap:

```
struct Foo<T>(T);

impl Foo<u8> {
    fn bar(&self) -> bool { self.0 > 5 }
}

impl Foo<bool> {
    fn bar(&self) -> bool { self.0 }
}
```

Error code E0203

Having multiple relaxed default bounds is unsupported.

Erroneous code example:

```
struct Bad<T: ?Sized + ?Send>{
    inner: T
}
```

Here the type `T` cannot have a relaxed bound for multiple default traits (`Sized` and `Send`). This can be fixed by only using one relaxed bound.

```
struct Good<T: ?Sized>{
    inner: T
}
```

Error code E0204

The `Copy` trait was implemented on a type which contains a field that doesn't implement the `Copy` trait.

Erroneous code example:

```
struct Foo {
    foo: Vec<u32>,
}

impl Copy for Foo { } // error!
```

The `Copy` trait is implemented by default only on primitive types. If your type only contains primitive types, you'll be able to implement `Copy` on it. Otherwise, it won't be possible.

Here's another example that will fail:

```
#[derive(Copy)] // error!
struct Foo<'a> {
    ty: &'a mut bool,
}
```

This fails because `&mut T` is not `Copy`, even when `T` is `Copy` (this differs from the behavior for `&T`, which is always `Copy`).

Error code E0205

Note: this error code is no longer emitted by the compiler.

An attempt to implement the `Copy` trait for an enum failed because one of the variants does not implement `Copy`. To fix this, you must implement `Copy` for the mentioned variant. Note that this may not be possible, as in the example of

```
enum Foo {
    Bar(Vec<u32>),
    Baz,
}

impl Copy for Foo { }
```

This fails because `Vec<T>` does not implement `Copy` for any `T`.

Here's another example that will fail:

```
#[derive(Copy)]
enum Foo<'a> {
    Bar(&'a mut bool),
    Baz,
}
```

This fails because `&mut T` is not `Copy`, even when `T` is `Copy` (this differs from the behavior for `&T`, which is always `Copy`).

Error code E0206

The `Copy` trait was implemented on a type which is neither a struct, an enum, nor a union.

Erroneous code example:

```
#[derive(Copy, Clone)]
struct Bar;

impl Copy for &'static mut Bar { } // error!
```

You can only implement `Copy` for a struct, an enum, or a union. The previous example will fail because `&'static mut Bar` is not a struct, an enum, or a union.

Error code E0207

A type, const or lifetime parameter that is specified for `impl` is not constrained.

Erroneous code example:

```
struct Foo;

impl<T: Default> Foo {
    // error: the type parameter `T` is not constrained by the impl trait,
self
    // type, or predicates [E0207]
    fn get(&self) -> T {
        <T as Default>::default()
    }
}
```

Any type or const parameter of an `impl` must meet at least one of the following criteria:

- it appears in the *implementing type* of the impl, e.g. `impl<T> Foo<T>`
- for a trait impl, it appears in the *implemented trait*, e.g. `impl<T> SomeTrait<T> for Foo`
- it is bound as an associated type, e.g. `impl<T, U> SomeTrait for T where T: AnotherTrait<AssocType=U>`

Any unconstrained lifetime parameter of an `impl` is not supported if the lifetime parameter is used by an associated type.

Error example 1

Suppose we have a struct `Foo` and we would like to define some methods for it. The previous code example has a definition which leads to a compiler error:

The problem is that the parameter `T` does not appear in the implementing type (`Foo`) of the impl. In this case, we can fix the error by moving the type parameter from the `impl` to the method `get` :

```
struct Foo;

// Move the type parameter from the impl to the method
impl Foo {
    fn get<T: Default>(&self) -> T {
        <T as Default>::default()
    }
}
```

Error example 2

As another example, suppose we have a `Maker` trait and want to establish a type `FooMaker` that makes `Foo` s:

```
trait Maker {
    type Item;
    fn make(&mut self) -> Self::Item;
}

struct Foo<T> {
    foo: T
}

struct FooMaker;

impl<T: Default> Maker for FooMaker {
// error: the type parameter `T` is not constrained by the impl trait, self
// type, or predicates [E0207]
    type Item = Foo<T>;

    fn make(&mut self) -> Foo<T> {
        Foo { foo: <T as Default>::default() }
    }
}
```

This fails to compile because `T` does not appear in the trait or in the implementing type.

One way to work around this is to introduce a phantom type parameter into `FooMaker` , like so:

```rust
use std::marker::PhantomData;

trait Maker {
    type Item;
    fn make(&mut self) -> Self::Item;
}

struct Foo<T> {
    foo: T
}

// Add a type parameter to `FooMaker`
struct FooMaker<T> {
    phantom: PhantomData<T>,
}

impl<T: Default> Maker for FooMaker<T> {
    type Item = Foo<T>;

    fn make(&mut self) -> Foo<T> {
        Foo {
            foo: <T as Default>::default(),
        }
    }
}
```

Another way is to do away with the associated type in `Maker` and use an input type parameter instead:

```rust
// Use a type parameter instead of an associated type here
trait Maker<Item> {
    fn make(&mut self) -> Item;
}

struct Foo<T> {
    foo: T
}

struct FooMaker;

impl<T: Default> Maker<Foo<T>> for FooMaker {
    fn make(&mut self) -> Foo<T> {
        Foo { foo: <T as Default>::default() }
    }
}
```

Error example 3

Suppose we have a struct `Foo` and we would like to define some methods for it. The following code example has a definition which leads to a compiler error:

```
struct Foo;

impl<const T: i32> Foo {
    // error: the const parameter `T` is not constrained by the impl trait,
self
    // type, or predicates [E0207]
    fn get(&self) -> i32 {
        i32::default()
    }
}
```

The problem is that the const parameter `T` does not appear in the implementing type (`Foo`) of the impl. In this case, we can fix the error by moving the type parameter from the `impl` to the method `get` :

```
struct Foo;

// Move the const parameter from the impl to the method
impl Foo {
    fn get<const T: i32>(&self) -> i32 {
        i32::default()
    }
}
```

Error example 4

Suppose we have a struct `Foo` and a struct `Bar` that uses lifetime `'a` . We would like to implement trait `Contains` for `Foo` . The trait `Contains` have the associated type `B` . The following code example has a definition which leads to a compiler error:

```
struct Foo;
struct Bar<'a>;

trait Contains {
    type B;

    fn get(&self) -> i32;
}

impl<'a> Contains for Foo {
    type B = Bar<'a>;

    // error: the lifetime parameter `'a` is not constrained by the impl
trait,
    // self type, or predicates [E0207]
    fn get(&self) -> i32 {
        i32::default()
    }
}
```

Please note that unconstrained lifetime parameters are not supported if they are being used by an associated type.

Additional information

For more information, please see RFC 447.

Error code E0208

This error code is internal to the compiler and will not be emitted with normal Rust code.

Note: this error code is no longer emitted by the compiler.

This error code shows the variance of a type's generic parameters.

Erroneous code example:

```
// NOTE: this feature is perma-unstable and should *only* be used for
//       testing purposes.
#![allow(internal_features)]
#![feature(rustc_attrs)]

#[rustc_variance]
struct Foo<'a, T> { // error: deliberate error to display type's variance
    t: &'a mut T,
}
```

which produces the following error:

```
error: [-, o]
 --> <anon>:4:1
  |
4 | struct Foo<'a, T> {
  | ^^^^^^^^^^^^^^^^^
```

Note that while `#[rustc_variance]` still exists and is used within the compiler, it no longer is marked as `E0208` and instead has no error code.

This error is deliberately triggered with the `#[rustc_variance]` attribute (`#![feature(rustc_attrs)]` must be enabled) and helps to show you the variance of the type's generic parameters. You can read more about variance and subtyping in this section of the Rustonomicon. For a more in depth look at variance (including a more complete list of common variances) see this section of the Reference. For information on how variance is implemented in the compiler, see this section of `rustc-dev-guide` .

This error can be easily fixed by removing the `#[rustc_variance]` attribute, the compiler's suggestion to comment it out can be applied automatically with `rustfix` .

Error code E0210

This error indicates a violation of one of Rust's orphan rules for trait implementations. The rule concerns the use of type parameters in an implementation of a foreign trait (a trait defined in another crate), and states that type parameters must be "covered" by a local type.

When implementing a foreign trait for a foreign type, the trait must have one or more type parameters. A type local to your crate must appear before any use of any type parameters.

To understand what this means, it is perhaps easier to consider a few examples.

If `ForeignTrait` is a trait defined in some external crate `foo`, then the following trait `impl` is an error:

```
extern crate foo;
use foo::ForeignTrait;

impl<T> ForeignTrait for T { } // error
```

To work around this, it can be covered with a local type, `MyType`:

```
struct MyType<T>(T);
impl<T> ForeignTrait for MyType<T> { } // Ok
```

Please note that a type alias is not sufficient.

For another example of an error, suppose there's another trait defined in `foo` named `ForeignTrait2` that takes two type parameters. Then this `impl` results in the same rule violation:

```
struct MyType2;
impl<T> ForeignTrait2<T, MyType<T>> for MyType2 { } // error
```

The reason for this is that there are two appearances of type parameter `T` in the `impl` header, both as parameters for `ForeignTrait2`. The first appearance is uncovered, and so runs afoul of the orphan rule.

Consider one more example:

```
impl<T> ForeignTrait2<MyType<T>, T> for MyType2 { } // Ok
```

This only differs from the previous `impl` in that the parameters `T` and `MyType<T>` for `ForeignTrait2` have been swapped. This example does *not* violate the orphan rule; it is permitted.

To see why that last example was allowed, you need to understand the general rule. Unfortunately this rule is a bit tricky to state. Consider an `impl`:

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

where `P1, ..., Pm` are the type parameters of the `impl` and `T0, ..., Tn` are types. One of the types `T0, ..., Tn` must be a local type (this is another orphan rule, see the explanation for E0117).

Both of the following must be true:

1. At least one of the types `T0..=Tn` must be a local type. Let `Ti` be the first such type.
2. No uncovered type parameters `P1..=Pm` may appear in `T0..Ti` (excluding `Ti`).

For information on the design of the orphan rules, see RFC 2451 and RFC 1023.

Error code E0211

Note: this error code is no longer emitted by the compiler.

You used a function or type which doesn't fit the requirements for where it was used. Erroneous code examples:

```
#![feature(intrinsics, rustc_attrs)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    #[rustc_safe_intrinsic]
    fn size_of<T>(); // error: intrinsic has wrong type
}

// or:

fn main() -> i32 { 0 }
// error: main function expects type: `fn() {main}`: expected (), found i32

// or:

let x = 1u8;
match x {
    0u8..=3i8 => (),
    // error: mismatched types in range: expected u8, found i8
    _ => ()
}

// or:

use std::rc::Rc;
struct Foo;

impl Foo {
    fn x(self: Rc<Foo>) {}
    // error: mismatched self type: expected `Foo`: expected struct
    //        `Foo`, found struct `alloc::rc::Rc`
}
```

For the first code example, please check the function definition. Example:

```
#![feature(intrinsics, rustc_attrs)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    #[rustc_safe_intrinsic]
    fn size_of<T>() -> usize; // ok!
}
```

The second case example is a bit particular: the main function must always have this definition:

```
fn main();
```

They never take parameters and never return types.

For the third example, when you match, all patterns must have the same type as the type you're matching on. Example:

```
let x = 1u8;

match x {
    0u8..=3u8 => (), // ok!
    _ => ()
}
```

And finally, for the last example, only `Box<Self>`, `&Self`, `Self`, or `&mut Self` work as explicit self parameters. Example:

```
struct Foo;

impl Foo {
    fn x(self: Box<Foo>) {} // ok!
}
```

Error code E0212

Cannot use the associated type of a trait with uninferred generic parameters.

Erroneous code example:

```
pub trait Foo<T> {
    type A;

    fn get(&self, t: T) -> Self::A;
}

fn foo2<I : for<'x> Foo<&'x isize>>(
    field: I::A) {} // error!
```

In this example, we have to instantiate `'x`, and we don't know what lifetime to instantiate it with. To fix this, spell out the precise lifetimes involved. Example:

```
pub trait Foo<T> {
    type A;

    fn get(&self, t: T) -> Self::A;
}

fn foo3<I : for<'x> Foo<&'x isize>>(
    x: <I as Foo<&isize>>::A) {} // ok!


fn foo4<'a, I : for<'x> Foo<&'x isize>>(
    x: <I as Foo<&'a isize>>::A) {} // ok!
```

Error code E0214

A generic type was described using parentheses rather than angle brackets.

Erroneous code example:

```
let v: Vec(&str) = vec!["foo"];
```

This is not currently supported: `v` should be defined as `Vec<&str>`. Parentheses are currently only used with generic types when defining parameters for `Fn`-family traits.

The previous code example fixed:

```
let v: Vec<&str> = vec!["foo"];
```

Error code E0220

The associated type used was not defined in the trait.

Erroneous code example:

```
trait T1 {
    type Bar;
}

type Foo = T1<F=i32>; // error: associated type `F` not found for `T1`

// or:

trait T2 {
    type Bar;

    // error: Baz is used but not declared
    fn return_bool(&self, _: &Self::Bar, _: &Self::Baz) -> bool;
}
```

Make sure that you have defined the associated type in the trait body. Also, verify that you used the right trait or you didn't misspell the associated type name. Example:

```
trait T1 {
    type Bar;
}

type Foo = T1<Bar=i32>; // ok!

// or:

trait T2 {
    type Bar;
    type Baz; // we declare `Baz` in our trait.

    // and now we can use it here:
    fn return_bool(&self, _: &Self::Bar, _: &Self::Baz) -> bool;
}
```

Error code E0221

An attempt was made to retrieve an associated type, but the type was ambiguous.

Erroneous code example:

```
trait T1 {}
trait T2 {}

trait Foo {
    type A: T1;
}

trait Bar : Foo {
    type A: T2;
    fn do_something() {
        let _: Self::A;
    }
}
```

In this example, `Foo` defines an associated type `A`. `Bar` inherits that type from `Foo`, and defines another associated type of the same name. As a result, when we attempt to use `Self::A`, it's ambiguous whether we mean the `A` defined by `Foo` or the one defined by `Bar`.

There are two options to work around this issue. The first is simply to rename one of the types. Alternatively, one can specify the intended type using the following syntax:

```
trait T1 {}
trait T2 {}

trait Foo {
    type A: T1;
}

trait Bar : Foo {
    type A: T2;
    fn do_something() {
        let _: <Self as Bar>::A;
    }
}
```

Error code E0222

An attempt was made to constrain an associated type.

Erroneous code example:

```
pub trait Vehicle {
    type Color;
}

pub trait Box {
    type Color;
}

pub trait BoxCar : Box + Vehicle {}

fn dent_object<COLOR>(c: dyn BoxCar<Color=COLOR>) {} // Invalid constraint
```

In this example, `BoxCar` has two supertraits: `Vehicle` and `Box`. Both of these traits define an associated type `Color`. `BoxCar` inherits two types with that name from both supertraits. Because of this, we need to use the fully qualified path syntax to refer to the appropriate `Color` associated type, either `<BoxCar as Vehicle>::Color` or `<BoxCar as Box>::Color`, but this syntax is not allowed to be used in a function signature.

In order to encode this kind of constraint, a `where` clause and a new type parameter are needed:

```
pub trait Vehicle {
    type Color;
}

pub trait Box {
    type Color;
}

pub trait BoxCar : Box + Vehicle {}

// Introduce a new `CAR` type parameter
fn foo<CAR, COLOR>(
    c: CAR,
) where
    // Bind the type parameter `CAR` to the trait `BoxCar`
    CAR: BoxCar,
    // Further restrict `<BoxCar as Vehicle>::Color` to be the same as the
    // type parameter `COLOR`
    CAR: Vehicle<Color = COLOR>,
    // We can also simultaneously restrict the other trait's associated type
    CAR: Box<Color = COLOR>
{}
```

Error code E0223

An attempt was made to retrieve an associated type, but the type was ambiguous.

Erroneous code example:

```
trait Trait { type X; }

fn main() {
    let foo: Trait::X;
}
```

The problem here is that we're attempting to take the associated type of `X` from `Trait`. Unfortunately, the type of `X` is not defined, because it's only made concrete in implementations of the trait. A working version of this code might look like:

```
trait Trait { type X; }

struct Struct;
impl Trait for Struct {
    type X = u32;
}

fn main() {
    let foo: <Struct as Trait>::X;
}
```

This syntax specifies that we want the associated type `X` from `Struct`'s implementation of `Trait`.

Due to internal limitations of the current compiler implementation we cannot simply use `Struct::X`.

Error code E0224

A trait object was declared with no traits.

Erroneous code example:

```
type Foo = dyn 'static +;
```

Rust does not currently support this.

To solve, ensure that the trait object has at least one trait:

```
type Foo = dyn 'static + Copy;
```

Error code E0225

Multiple types were used as bounds for a closure or trait object.

Erroneous code example:

```
fn main() {
    let _: Box<dyn std::io::Read + std::io::Write>;
}
```

Rust does not currently support this.

Auto traits such as Send and Sync are an exception to this rule: It's possible to have bounds of one non-builtin trait, plus any number of auto traits. For example, the following compiles correctly:

```
fn main() {
    let _: Box<dyn std::io::Read + Send + Sync>;
}
```

Error code E0226

More than one explicit lifetime bound was used on a trait object.

Example of erroneous code:

```
trait Foo {}

type T<'a, 'b> = dyn Foo + 'a + 'b; // error: Trait object `arg` has two
                                    //        lifetime bound, 'a and 'b.
```

Here `T` is a trait object with two explicit lifetime bounds, 'a and 'b.

Only a single explicit lifetime bound is permitted on trait objects. To fix this error, consider removing one of the lifetime bounds:

```
trait Foo {}

type T<'a> = dyn Foo + 'a;
```

Error code E0227

This error indicates that the compiler is unable to determine whether there is exactly one unique region in the set of derived region bounds.

Example of erroneous code:

```
trait Foo<'foo>: 'foo {}
trait Bar<'bar>: 'bar {}

trait FooBar<'foo, 'bar>: Foo<'foo> + Bar<'bar> {}

struct Baz<'foo, 'bar> {
    baz: dyn FooBar<'foo, 'bar>,
}
```

Here, `baz` can have either `'foo` or `'bar` lifetimes.

To resolve this error, provide an explicit lifetime:

```
trait Foo<'foo>: 'foo {}
trait Bar<'bar>: 'bar {}

trait FooBar<'foo, 'bar>: Foo<'foo> + Bar<'bar> {}

struct Baz<'foo, 'bar, 'baz>
where
    'baz: 'foo + 'bar,
{
    obj: dyn FooBar<'foo, 'bar> + 'baz,
}
```

Error code E0228

The lifetime bound for this object type cannot be deduced from context and must be specified.

Erroneous code example:

```
trait Trait { }

struct TwoBounds<'a, 'b, T: Sized + 'a + 'b> {
    x: &'a i32,
    y: &'b i32,
    z: T,
}

type Foo<'a, 'b> = TwoBounds<'a, 'b, dyn Trait>;
```

When a trait object is used as a type argument of a generic type, Rust will try to infer its lifetime if unspecified. However, this isn't possible when the containing type has more than one lifetime bound.

The above example can be resolved by either reducing the number of lifetime bounds to one or by making the trait object lifetime explicit, like so:

```
trait Trait { }

struct TwoBounds<'a, 'b, T: Sized + 'a + 'b> {
    x: &'a i32,
    y: &'b i32,
    z: T,
}

type Foo<'a, 'b> = TwoBounds<'a, 'b, dyn Trait + 'b>;
```

For more information, see RFC 599 and its amendment RFC 1156.

Error code E0229

An associated type binding was done outside of the type parameter declaration and `where` clause.

Erroneous code example:

```
pub trait Foo {
    type A;
    fn boo(&self) -> <Self as Foo>::A;
}

struct Bar;

impl Foo for isize {
    type A = usize;
    fn boo(&self) -> usize { 42 }
}

fn baz<I>(x: &<I as Foo<A=Bar>>::A) {}
// error: associated type bindings are not allowed here
```

To solve this error, please move the type bindings in the type parameter declaration:

```
fn baz<I: Foo<A=Bar>>(x: &<I as Foo>::A) {} // ok!
```

Or in the `where` clause:

```
fn baz<I>(x: &<I as Foo>::A) where I: Foo<A=Bar> {}
```

Error code E0230

The `#[rustc_on_unimplemented]` attribute lets you specify a custom error message for when a particular trait isn't implemented on a type placed in a position that needs that trait. For example, when the following code is compiled:

```
#![feature(rustc_attrs)]
#![allow(internal_features)]

#[rustc_on_unimplemented = "error on `{Self}` with params `<{A},{B}>`"] //
error
trait BadAnnotation<A> {}
```

There will be an error about `bool` not implementing `Index<u8>`, followed by a note saying "the type `bool` cannot be indexed by `u8`".

As you can see, you can specify type parameters in curly braces for substitution with the actual types (using the regular format string syntax) in a given situation. Furthermore, `{Self}` will substitute to the type (in this case, `bool`) that we tried to use.

This error appears when the curly braces contain an identifier which doesn't match with any of the type parameters or the string `Self`. This might happen if you misspelled a type parameter, or if you intended to use literal curly braces. If it is the latter, escape the curly braces with a second curly brace of the same type; e.g., a literal `{` is `{{`.

Error code E0231

The `#[rustc_on_unimplemented]` attribute lets you specify a custom error message for when a particular trait isn't implemented on a type placed in a position that needs that trait. For example, when the following code is compiled:

```
#![feature(rustc_attrs)]
#![allow(internal_features)]

#[rustc_on_unimplemented = "error on `{Self}` with params `<{A},{}>`"] //
error!
trait BadAnnotation<A> {}
```

there will be an error about `bool` not implementing `Index<u8>`, followed by a note saying "the type `bool` cannot be indexed by `u8`".

As you can see, you can specify type parameters in curly braces for substitution with the actual types (using the regular format string syntax) in a given situation. Furthermore, `{Self}` will substitute to the type (in this case, `bool`) that we tried to use.

This error appears when the curly braces do not contain an identifier. Please add one of the same name as a type parameter. If you intended to use literal braces, use `{{` and `}}` to escape them.

Error code E0232

The `#[rustc_on_unimplemented]` attribute lets you specify a custom error message for when a particular trait isn't implemented on a type placed in a position that needs that trait. For example, when the following code is compiled:

```
#![feature(rustc_attrs)]
#![allow(internal_features)]

#[rustc_on_unimplemented(lorem="")] // error!
trait BadAnnotation {}
```

there will be an error about `bool` not implementing `Index<u8>`, followed by a note saying "the type `bool` cannot be indexed by `u8`".

For this to work, some note must be specified. An empty attribute will not do anything, please remove the attribute or add some helpful note for users of the trait.

Error code E0243

Note: this error code is no longer emitted by the compiler.

This error indicates that not enough type parameters were found in a type or trait.

For example, the `Foo` struct below is defined to be generic in `T`, but the type parameter is missing in the definition of `Bar`:

```
struct Foo<T> { x: T }

struct Bar { x: Foo }
```

Error code E0244

Note: this error code is no longer emitted by the compiler.

This error indicates that too many type parameters were found in a type or trait.

For example, the `Foo` struct below has no type parameters, but is supplied with two in the definition of `Bar`:

```
struct Foo { x: bool }

struct Bar<S, T> { x: Foo<S, T> }
```

Error code E0251

Note: this error code is no longer emitted by the compiler.

Two items of the same name cannot be imported without rebinding one of the items under a new local name.

An example of this error:

```
use foo::baz;
use bar::*; // error, do `use foo::baz as quux` instead on the previous line

fn main() {}

mod foo {
    pub struct baz;
}

mod bar {
    pub mod baz {}
}
```

Error code E0252

Two items of the same name cannot be imported without rebinding one of the items under a new local name.

Erroneous code example:

```
use foo::baz;
use bar::baz; // error, do `use bar::baz as quux` instead

fn main() {}

mod foo {
    pub struct baz;
}

mod bar {
    pub mod baz {}
}
```

You can use aliases in order to fix this error. Example:

```
use foo::baz as foo_baz;
use bar::baz; // ok!

fn main() {}

mod foo {
    pub struct baz;
}

mod bar {
    pub mod baz {}
}
```

Or you can reference the item with its parent:

```
use bar::baz;

fn main() {
    let x = foo::baz; // ok!
}

mod foo {
    pub struct baz;
}

mod bar {
    pub mod baz {}
}
```

Error code E0253

Attempt was made to import an unimportable value. This can happen when trying to import a method from a trait.

Erroneous code example:

```
mod foo {
    pub trait MyTrait {
        fn do_something();
    }
}

use foo::MyTrait::do_something;
// error: `do_something` is not directly importable

fn main() {}
```

It's invalid to directly import methods belonging to a trait or concrete type.

Error code E0254

Attempt was made to import an item whereas an extern crate with this name has already been imported.

Erroneous code example:

```
extern crate core;

mod foo {
    pub trait core {
        fn do_something();
    }
}

use foo::core;  // error: an extern crate named `core` has already
                //        been imported in this module

fn main() {}
```

To fix this issue, you have to rename at least one of the two imports. Example:

```
extern crate core as libcore; // ok!

mod foo {
    pub trait core {
        fn do_something();
    }
}

use foo::core;

fn main() {}
```

Error code E0255

You can't import a value whose name is the same as another value defined in the module.

Erroneous code example:

```
use bar::foo; // error: an item named `foo` is already in scope

fn foo() {}

mod bar {
     pub fn foo() {}
}

fn main() {}
```

You can use aliases in order to fix this error. Example:

```
use bar::foo as bar_foo; // ok!

fn foo() {}

mod bar {
     pub fn foo() {}
}

fn main() {}
```

Or you can reference the item with its parent:

```
fn foo() {}

mod bar {
     pub fn foo() {}
}

fn main() {
    bar::foo(); // we get the item by referring to its parent
}
```

Error code E0256

Note: this error code is no longer emitted by the compiler.

You can't import a type or module when the name of the item being imported is the same as another type or submodule defined in the module.

An example of this error:

```
use foo::Bar; // error

type Bar = u32;

mod foo {
    pub mod Bar { }
}

fn main() {}
```

Error code E0259

The name chosen for an external crate conflicts with another external crate that has been imported into the current module.

Erroneous code example:

```
extern crate core;
extern crate std as core;

fn main() {}
```

The solution is to choose a different name that doesn't conflict with any external crate imported into the current module.

Correct example:

```
extern crate core;
extern crate std as other_name;

fn main() {}
```

Error code E0260

The name for an item declaration conflicts with an external crate's name.

Erroneous code example:

```
extern crate core;

struct core;

fn main() {}
```

There are two possible solutions:

Solution #1: Rename the item.

```
extern crate core;

struct xyz;
```

Solution #2: Import the crate with a different name.

```
extern crate core as xyz;

struct abc;
```

See the Declaration Statements section of the reference for more information about what constitutes an item declaration and what does not.

Error code E0261

An undeclared lifetime was used.

Erroneous code example:

```
// error, use of undeclared lifetime name `'a`
fn foo(x: &'a str) { }

struct Foo {
    // error, use of undeclared lifetime name `'a`
    x: &'a str,
}
```

These can be fixed by declaring lifetime parameters:

```
struct Foo<'a> {
    x: &'a str,
}

fn foo<'a>(x: &'a str) {}
```

Impl blocks declare lifetime parameters separately. You need to add lifetime parameters to an impl block if you're implementing a type that has a lifetime parameter of its own. For example:

```
struct Foo<'a> {
    x: &'a str,
}

// error,  use of undeclared lifetime name `'a`
impl Foo<'a> {
    fn foo<'a>(x: &'a str) {}
}
```

This is fixed by declaring the impl block like this:

```
struct Foo<'a> {
    x: &'a str,
}

// correct
impl<'a> Foo<'a> {
    fn foo(x: &'a str) {}
}
```

Error code E0262

An invalid name was used for a lifetime parameter.

Erroneous code example:

```
// error, invalid lifetime parameter name `'static`
fn foo<'static>(x: &'static str) { }
```

Declaring certain lifetime names in parameters is disallowed. For example, because the `'static` lifetime is a special built-in lifetime name denoting the lifetime of the entire program, this is an error:

Error code E0263

Note: this error code is no longer emitted by the compiler.

A lifetime was declared more than once in the same scope.

Erroneous code example:

```
fn foo<'a, 'b, 'a>(x: &'a str, y: &'b str, z: &'a str) { // error!
}
```

Two lifetimes cannot have the same name. To fix this example, change the second `'a` lifetime into something else (`'c` for example):

```
fn foo<'a, 'b, 'c>(x: &'a str, y: &'b str, z: &'c str) { // ok!
}
```

Error code E0264

An unknown external lang item was used.

Erroneous code example:

```
#![feature(lang_items)]
#![allow(internal_features)]

extern "C" {
    #[lang = "cake"] // error: unknown external lang item: `cake`
    fn cake();
}
```

A list of available external lang items is available in `src/librustc_middle/middle/weak_lang_items.rs`. Example:

```
#![feature(lang_items)]
#![allow(internal_features)]

extern "C" {
    #[lang = "panic_impl"] // ok!
    fn cake();
}
```

Error code E0267

A loop keyword (`break` or `continue`) was used inside a closure but outside of any loop.

Erroneous code example:

```
let w = || { break; }; // error: `break` inside of a closure
```

`break` and `continue` keywords can be used as normal inside closures as long as they are also contained within a loop. To halt the execution of a closure you should instead use a return statement. Example:

```
let w = || {
    for _ in 0..10 {
        break;
    }
};

w();
```

Error code E0268

A loop keyword (`break` or `continue`) was used outside of a loop.

Erroneous code example:

```
fn some_func() {
    break; // error: `break` outside of a loop
}
```

Without a loop to break out of or continue in, no sensible action can be taken. Please verify that you are using `break` and `continue` only in loops. Example:

```
fn some_func() {
    for _ in 0..10 {
        break; // ok!
    }
}
```

Error code E0271

A type mismatched an associated type of a trait.

Erroneous code example:

```
trait Trait { type AssociatedType; }

fn foo<T>(t: T) where T: Trait<AssociatedType=u32> {
//                    ~~~~~~~~ ~~~~~~~~~~~~~~~~~~
//                        |            |
//         This says `foo` can         |
//           only be used with         |
//              some type that         |
//         implements `Trait`.         |
//                                     |
//                             This says not only must
//                             `T` be an impl of `Trait`
//                             but also that the impl
//                             must assign the type `u32`
//                             to the associated type.
    println!("in foo");
}

impl Trait for i8 { type AssociatedType = &'static str; }
//~~~~~~~~~~~~~~~   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//      |                             |
// `i8` does have                     |
// implementation                     |
// of `Trait`...                      |
//                     ... but it is an implementation
//                     that assigns `&'static str` to
//                     the associated type.

foo(3_i8);
// Here, we invoke `foo` with an `i8`, which does not satisfy
// the constraint `<i8 as Trait>::AssociatedType=u32`, and
// therefore the type-checker complains with this error code.
```

The issue can be resolved by changing the associated type:

1. in the `foo` implementation:

```
trait Trait { type AssociatedType; }

fn foo<T>(t: T) where T: Trait<AssociatedType = &'static str> {
    println!("in foo");
}

impl Trait for i8 { type AssociatedType = &'static str; }

foo(3_i8);
```

2. in the `Trait` implementation for `i8`:

```
trait Trait { type AssociatedType; }

fn foo<T>(t: T) where T: Trait<AssociatedType = u32> {
    println!("in foo");
}

impl Trait for i8 { type AssociatedType = u32; }

foo(3_i8);
```

Error code E0275

An evaluation of a trait requirement overflowed.

Erroneous code example:

```
trait Foo {}

struct Bar<T>(T);

impl<T> Foo for T where Bar<T>: Foo {}
```

This error occurs when there was a recursive trait requirement that overflowed before it could be evaluated. This often means that there is an unbounded recursion in resolving some type bounds.

To determine if a `T` is `Foo`, we need to check if `Bar<T>` is `Foo`. However, to do this check, we need to determine that `Bar<Bar<T>>` is `Foo`. To determine this, we check if `Bar<Bar<Bar<T>>>` is `Foo`, and so on. This is clearly a recursive requirement that can't be resolved directly.

Consider changing your trait bounds so that they're less self-referential.

Error code E0276

A trait implementation has stricter requirements than the trait definition.

Erroneous code example:

```
trait Foo {
    fn foo<T>(x: T);
}

impl Foo for bool {
    fn foo<T>(x: T) where T: Copy {}
}
```

Here, all types implementing `Foo` must have a method `foo<T>(x: T)` which can take any type `T`. However, in the `impl` for `bool`, we have added an extra bound that `T` is `Copy`, which isn't compatible with the original trait.

Consider removing the bound from the method or adding the bound to the original method definition in the trait.

Error code E0277

You tried to use a type which doesn't implement some trait in a place which expected that trait.

Erroneous code example:

```
// here we declare the Foo trait with a bar method
trait Foo {
    fn bar(&self);
}

// we now declare a function which takes an object implementing the Foo
trait
fn some_func<T: Foo>(foo: T) {
    foo.bar();
}

fn main() {
    // we now call the method with the i32 type, which doesn't implement
    // the Foo trait
    some_func(5i32); // error: the trait bound `i32 : Foo` is not satisfied
}
```

In order to fix this error, verify that the type you're using does implement the trait. Example:

```
trait Foo {
    fn bar(&self);
}

// we implement the trait on the i32 type
impl Foo for i32 {
    fn bar(&self) {}
}

fn some_func<T: Foo>(foo: T) {
    foo.bar(); // we can now use this method since i32 implements the
               // Foo trait
}

fn main() {
    some_func(5i32); // ok!
}
```

Or in a generic context, an erroneous code example would look like:

```
fn some_func<T>(foo: T) {
    println!("{:?}", foo); // error: the trait `core::fmt::Debug` is not
                           //        implemented for the type `T`
}

fn main() {
    // We now call the method with the i32 type,
    // which *does* implement the Debug trait.
    some_func(5i32);
}
```

Note that the error here is in the definition of the generic function. Although we only call it with a parameter that does implement `Debug`, the compiler still rejects the function. It must work with all possible input types. In order to make this example compile, we need to restrict the generic type we're accepting:

```
use std::fmt;

// Restrict the input type to types that implement Debug.
fn some_func<T: fmt::Debug>(foo: T) {
    println!("{:?}", foo);
}

fn main() {
    // Calling the method is still fine, as i32 implements Debug.
    some_func(5i32);

    // This would fail to compile now:
    // struct WithoutDebug;
    // some_func(WithoutDebug);
}
```

Rust only looks at the signature of the called function, as such it must already specify all requirements that will be used for every type parameter.

Error code E0281

Note: this error code is no longer emitted by the compiler.

You tried to supply a type which doesn't implement some trait in a location which expected that trait. This error typically occurs when working with `Fn`-based types. Erroneous code example:

```
fn foo<F: Fn(usize)>(x: F) { }

fn main() {
    // type mismatch: ... implements the trait `core::ops::Fn<(String,)>`,
    // but the trait `core::ops::Fn<(usize,)>` is required
    // [E0281]
    foo(|y: String| { });
}
```

The issue in this case is that `foo` is defined as accepting a `Fn` with one argument of type `String`, but the closure we attempted to pass to it requires one arguments of type `usize`.

Error code E0282

The compiler could not infer a type and asked for a type annotation.

Erroneous code example:

```
let x = Vec::new();
```

This error indicates that type inference did not result in one unique possible type, and extra information is required. In most cases this can be provided by adding a type annotation. Sometimes you need to specify a generic type parameter manually.

In the example above, type `Vec` has a type parameter `T`. When calling `Vec::new`, barring any other later usage of the variable `x` that allows the compiler to infer what type `T` is, the compiler needs to be told what it is.

The type can be specified on the variable:

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

The type can also be specified in the path of the expression:

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

In cases with more complex types, it is not necessary to annotate the full type. Once the ambiguity is resolved, the compiler can infer the rest:

```
let x: Vec<_> = "hello".chars().rev().collect();
```

Another way to provide the compiler with enough information, is to specify the generic type parameter:

```
let x = "hello".chars().rev().collect::<Vec<char>>();
```

Again, you need not specify the full type if the compiler can infer it:

```
let x = "hello".chars().rev().collect::<Vec<_>>();
```

Apart from a method or function with a generic type parameter, this error can occur when a type parameter of a struct or trait cannot be inferred. In that case it is not always possible to use a type annotation, because all candidates have the same return type. For instance:

```
struct Foo<T> {
    num: T,
}

impl<T> Foo<T> {
    fn bar() -> i32 {
        0
    }

    fn baz() {
        let number = Foo::bar();
    }
}
```

This will fail because the compiler does not know which instance of `Foo` to call `bar` on. Change `Foo::bar()` to `Foo::<T>::bar()` to resolve the error.

Error code E0283

The compiler could not infer a type and asked for a type annotation.

Erroneous code example:

```
let x = "hello".chars().rev().collect();
```

This error indicates that type inference did not result in one unique possible type, and extra information is required. In most cases this can be provided by adding a type annotation. Sometimes you need to specify a generic type parameter manually.

A common example is the `collect` method on `Iterator`. It has a generic type parameter with a `FromIterator` bound, which for a `char` iterator is implemented by `Vec` and `String` among others. Consider the following snippet that reverses the characters of a string:

In the first code example, the compiler cannot infer what the type of `x` should be: `Vec<char>` and `String` are both suitable candidates. To specify which type to use, you can use a type annotation on `x`:

```
let x: Vec<char> = "hello".chars().rev().collect();
```

It is not necessary to annotate the full type. Once the ambiguity is resolved, the compiler can infer the rest:

```
let x: Vec<_> = "hello".chars().rev().collect();
```

Another way to provide the compiler with enough information, is to specify the generic type parameter:

```
let x = "hello".chars().rev().collect::<Vec<char>>();
```

Again, you need not specify the full type if the compiler can infer it:

```
let x = "hello".chars().rev().collect::<Vec<_>>();
```

We can see a self-contained example below:

```
struct Foo;

impl Into<u32> for Foo {
    fn into(self) -> u32 { 1 }
}

let foo = Foo;
let bar: u32 = foo.into() * 1u32;
```

This error can be solved by adding type annotations that provide the missing information to the compiler. In this case, the solution is to specify the trait's type parameter:

```
struct Foo;

impl Into<u32> for Foo {
    fn into(self) -> u32 { 1 }
}

let foo = Foo;
let bar: u32 = Into::<u32>::into(foo) * 1u32;
```

Error code E0284

This error occurs when the compiler is unable to unambiguously infer the return type of a function or method which is generic on return type, such as the `collect` method for `Iterator`s.

For example:

```
fn main() {
    let n: u32 = 1;
    let mut d: u64 = 2;
    d = d + n.into();
}
```

Here we have an addition of `d` and `n.into()`. Hence, `n.into()` can return any type `T` where `u64: Add<T>`. On the other hand, the `into` method can return any type where `u32: Into<T>`.

The author of this code probably wants `into()` to return a `u64`, but the compiler can't be sure that there isn't another type `T` where both `u32: Into<T>` and `u64: Add<T>`.

To resolve this error, use a concrete type for the intermediate expression:

```
fn main() {
    let n: u32 = 1;
    let mut d: u64 = 2;
    let m: u64 = n.into();
    d = d + m;
}
```

Error code E0297

Note: this error code is no longer emitted by the compiler.

Patterns used to bind names must be irrefutable. That is, they must guarantee that a name will be extracted in all cases. Instead of pattern matching the loop variable, consider using a `match` or `if let` inside the loop body. For instance:

```
let xs : Vec<Option<i32>> = vec![Some(1), None];

// This fails because `None` is not covered.
for Some(x) in xs {
    // ...
}
```

Match inside the loop instead:

```
let xs : Vec<Option<i32>> = vec![Some(1), None];

for item in xs {
    match item {
        Some(x) => {},
        None => {},
    }
}
```

Or use `if let`:

```
let xs : Vec<Option<i32>> = vec![Some(1), None];

for item in xs {
    if let Some(x) = item {
        // ...
    }
}
```

Error code E0301

Note: this error code is no longer emitted by the compiler.

Mutable borrows are not allowed in pattern guards, because matching cannot have side effects. Side effects could alter the matched object or the environment on which the match depends in such a way, that the match would not be exhaustive. For instance, the following would not match any arm if mutable borrows were allowed:

```
match Some(()) {
    None => { },
    option if option.take().is_none() => {
        /* impossible, option is `Some` */
    },
    Some(_) => { } // When the previous match failed, the option became
`None`.
}
```

Error code E0302

Note: this error code is no longer emitted by the compiler.

Assignments are not allowed in pattern guards, because matching cannot have side effects. Side effects could alter the matched object or the environment on which the match depends in such a way, that the match would not be exhaustive. For instance, the following would not match any arm if assignments were allowed:

```
match Some(()) {
    None => { },
    option if { option = None; false } => { },
    Some(_) => { } // When the previous match failed, the option became
`None`.
}
```

Error code E0303

Note: this error code is no longer emitted by the compiler.

Sub-bindings, e.g. `ref x @ Some(ref y)` are now allowed under `#![feature(bindings_after_at)]` and checked to make sure that memory safety is upheld.

In certain cases it is possible for sub-bindings to violate memory safety. Updates to the borrow checker in a future version of Rust may remove this restriction, but for now patterns must be rewritten without sub-bindings.

Before:

```
match Some("hi".to_string()) {
    ref op_string_ref @ Some(s) => {},
    None => {},
}
```

After:

```
match Some("hi".to_string()) {
    Some(ref s) => {
        let op_string_ref = &Some(s);
        // ...
    },
    None => {},
}
```

The `op_string_ref` binding has type `&Option<&String>` in both cases.

See also Issue 14587.

Error code E0307

The `self` parameter in a method has an invalid "receiver type".

Erroneous code example:

```
struct Foo;
struct Bar;

trait Trait {
    fn foo(&self);
}

impl Trait for Foo {
    fn foo(self: &Bar) {}
}
```

Methods take a special first parameter, of which there are three variants: `self`, `&self`, and `&mut self`. These are syntactic sugar for `self: Self`, `self: &Self`, and `self: &mut Self` respectively.

```
trait Trait {
    fn foo(&self);
//         ^^^^^ `self` here is a reference to the receiver object
}

impl Trait for Foo {
    fn foo(&self) {}
//         ^^^^^ the receiver type is `&Foo`
}
```

The type `Self` acts as an alias to the type of the current trait implementer, or "receiver type". Besides the already mentioned `Self`, `&Self` and `&mut Self` valid receiver types, the following are also valid: `self: Box<Self>`, `self: Rc<Self>`, `self: Arc<Self>`, and `self: Pin<P>` (where P is one of the previous types except `Self`). Note that `Self` can also be the underlying implementing type, like `Foo` in the following example:

```
impl Trait for Foo {
    fn foo(self: &Foo) {}
}
```

This error will be emitted by the compiler when using an invalid receiver type, like in the following example:

```
impl Trait for Foo {
    fn foo(self: &Bar) {}
}
```

The nightly feature Arbitrary self types extends the accepted set of receiver types to also include any type that can dereference to `Self`:

```
#![feature(arbitrary_self_types)]

struct Foo;
struct Bar;

// Because you can dereference `Bar` into `Foo`...
impl std::ops::Deref for Bar {
    type Target = Foo;

    fn deref(&self) -> &Foo {
        &Foo
    }
}

impl Foo {
    fn foo(self: Bar) {}
//         ^^^^^^^^^ ...it can be used as the receiver type
}
```

Error code E0308

Expected type did not match the received type.

Erroneous code examples:

```
fn plus_one(x: i32) -> i32 {
    x + 1
}

plus_one("Not a number");
//       ^^^^^^^^^^^^^^ expected `i32`, found `&str`

if "Not a bool" {
// ^^^^^^^^^^^^ expected `bool`, found `&str`
}

let x: f32 = "Not a float";
//     ---   ^^^^^^^^^^^^^ expected `f32`, found `&str`
//     |
//     expected due to this
```

This error occurs when an expression was used in a place where the compiler expected an expression of a different type. It can occur in several cases, the most common being when calling a function and passing an argument which has a different type than the matching type in the function declaration.

Error code E0309

A parameter type is missing an explicit lifetime bound and may not live long enough.

Erroneous code example:

```
// This won't compile because the applicable impl of
// `SomeTrait` (below) requires that `T: 'a`, but the struct does
// not have a matching where-clause.
struct Foo<'a, T> {
    foo: <T as SomeTrait<'a>>::Output,
}

trait SomeTrait<'a> {
    type Output;
}

impl<'a, T> SomeTrait<'a> for T
where
    T: 'a,
{
    type Output = u32;
}
```

The type definition contains some field whose type requires an outlives annotation. Outlives annotations (e.g., `T: 'a`) are used to guarantee that all the data in `T` is valid for at least the lifetime `'a`. This scenario most commonly arises when the type contains an associated type reference like `<T as SomeTrait<'a>>::Output`, as shown in the previous code.

There, the where clause `T: 'a` that appears on the impl is not known to be satisfied on the struct. To make this example compile, you have to add a where-clause like `T: 'a` to the struct definition:

```
struct Foo<'a, T>
where
    T: 'a,
{
    foo: <T as SomeTrait<'a>>::Output
}

trait SomeTrait<'a> {
    type Output;
}

impl<'a, T> SomeTrait<'a> for T
where
    T: 'a,
{
    type Output = u32;
}
```

Error code E0310

A parameter type is missing a lifetime constraint or has a lifetime that does not live long enough.

Erroneous code example:

```
// This won't compile because T is not constrained to the static lifetime
// the reference needs
struct Foo<T> {
    foo: &'static T
}
```

Type parameters in type definitions have lifetimes associated with them that represent how long the data stored within them is guaranteed to live. This lifetime must be as long as the data needs to be alive, and missing the constraint that denotes this will cause this error.

This will compile, because it has the constraint on the type parameter:

```
struct Foo<T: 'static> {
    foo: &'static T
}
```

Error code E0311

This error occurs when there is an unsatisfied outlives bound involving an elided region and a generic type parameter or associated type.

Erroneous code example:

```
fn no_restriction<T>(x: &()) -> &() {
    with_restriction::<T>(x)
}

fn with_restriction<'a, T: 'a>(x: &'a ()) -> &'a () {
    x
}
```

Why doesn't this code compile? It helps to look at the lifetime bounds that are automatically added by the compiler. For more details see the documentation for lifetime elision.

The compiler elides the lifetime of `x` and the return type to some arbitrary lifetime `'anon` in `no_restriction()`. The only information available to the compiler is that `'anon` is valid for the duration of the function. When calling `with_restriction()`, the compiler requires the completely unrelated type parameter `T` to outlive `'anon` because of the `T: 'a` bound in `with_restriction()`. This causes an error because `T` is not required to outlive `'anon` in `no_restriction()`.

If `no_restriction()` were to use `&T` instead of `&()` as an argument, the compiler would have added an implied bound, causing this to compile.

This error can be resolved by explicitly naming the elided lifetime for `x` and then explicitly requiring that the generic parameter `T` outlives that lifetime:

```
fn no_restriction<'a, T: 'a>(x: &'a ()) -> &'a () {
    with_restriction::<T>(x)
}

fn with_restriction<'a, T: 'a>(x: &'a ()) -> &'a () {
    x
}
```

Error code E0312

Note: this error code is no longer emitted by the compiler.

Reference's lifetime of borrowed content doesn't match the expected lifetime.

Erroneous code example:

```
pub fn opt_str<'a>(maybestr: &'a Option<String>) -> &'static str {
    if maybestr.is_none() {
        "(none)"
    } else {
        let s: &'a str = maybestr.as_ref().unwrap();
        s  // Invalid lifetime!
    }
}
```

To fix this error, either lessen the expected lifetime or find a way to not have to use this reference outside of its current scope (by running the code directly in the same block for example?):

```
// In this case, we can fix the issue by switching from "static" lifetime to
'a
pub fn opt_str<'a>(maybestr: &'a Option<String>) -> &'a str {
    if maybestr.is_none() {
        "(none)"
    } else {
        let s: &'a str = maybestr.as_ref().unwrap();
        s  // Ok!
    }
}
```

Error code E0316

A `where` clause contains a nested quantification over lifetimes.

Erroneous code example:

```
trait Tr<'a, 'b> {}

fn foo<T>(t: T)
where
    for<'a> &'a T: for<'b> Tr<'a, 'b>, // error: nested quantification
{
}
```

Rust syntax allows lifetime quantifications in two places within `where` clauses: Quantifying over the trait bound only (as in `Ty: for<'l> Trait<'l>`) and quantifying over the whole clause (as in `for<'l> &'l Ty: Trait<'l>`). Using both in the same clause leads to a nested lifetime quantification, which is not supported.

The following example compiles, because the clause with the nested quantification has been rewritten to use only one `for<>`:

```
trait Tr<'a, 'b> {}

fn foo<T>(t: T)
where
    for<'a, 'b> &'a T: Tr<'a, 'b>, // ok
{
}
```

Error code E0317

An `if` expression is missing an `else` block.

Erroneous code example:

```
let x = 5;
let a = if x == 5 {
    1
};
```

This error occurs when an `if` expression without an `else` block is used in a context where a type other than `()` is expected. In the previous code example, the `let` expression was expecting a value but since there was no `else`, no value was returned.

An `if` expression without an `else` block has the type `()`, so this is a type error. To resolve it, add an `else` block having the same type as the `if` block.

So to fix the previous code example:

```
let x = 5;
let a = if x == 5 {
    1
} else {
    2
};
```

Error code E0320

Recursion limit reached while creating drop-check rules.

Example of erroneous code:

```
enum A<T> {
    B,
    C(T, Box<A<(T, T)>>)
}

fn foo<T>() {
    A::<T>::B; // error: overflow while adding drop-check rules for A<T>
}
```

The Rust compiler must be able to reason about how a type is `Drop`ped, and by extension the types of its fields, to be able to generate the glue to properly drop a value. The code example above shows a type where this inference is impossible because it is recursive. Note that this is *not* the same as E0072, where a type has an infinite size; the type here has a finite size but any attempt to `Drop` it would recurse infinitely. For more information, read the `Drop` docs.

It is not possible to define a type with recursive drop-check rules. All such recursion must be removed.

Error code E0321

A cross-crate opt-out trait was implemented on something which wasn't a struct or enum type.

Erroneous code example:

```
#![feature(auto_traits)]

struct Foo;

impl !Sync for Foo {}

unsafe impl Send for &'static Foo {}
// error: cross-crate traits with a default impl, like `core::marker::Send`,
//        can only be implemented for a struct/enum type, not
//        `&'static Foo`
```

Only structs and enums are permitted to impl Send, Sync, and other opt-out trait, and the struct or enum must be local to the current crate. So, for example, `unsafe impl Send for Rc<Foo>` is not allowed.

Error code E0322

A built-in trait was implemented explicitly. All implementations of the trait are provided automatically by the compiler.

Erroneous code example:

```
struct Foo;

impl Sized for Foo {} // error!
```

The `Sized` trait is a special trait built-in to the compiler for types with a constant size known at compile-time. This trait is automatically implemented for types as needed by the compiler, and it is currently disallowed to explicitly implement it for a type.

Error code E0323

An associated const was implemented when another trait item was expected.

Erroneous code example:

```
trait Foo {
    type N;
}

struct Bar;

impl Foo for Bar {
    const N : u32 = 0;
    // error: item `N` is an associated const, which doesn't match its
    //        trait `<Bar as Foo>`
}
```

Please verify that the associated const wasn't misspelled and the correct trait was implemented. Example:

```
struct Bar;

trait Foo {
    type N;
}

impl Foo for Bar {
    type N = u32; // ok!
}
```

Or:

```
struct Bar;

trait Foo {
    const N : u32;
}

impl Foo for Bar {
    const N : u32 = 0; // ok!
}
```

Error code E0324

A method was implemented when another trait item was expected.

Erroneous code example:

```
struct Bar;

trait Foo {
    const N : u32;

    fn M();
}

impl Foo for Bar {
    fn N() {}
    // error: item `N` is an associated method, which doesn't match its
    //        trait `<Bar as Foo>`
}
```

To fix this error, please verify that the method name wasn't misspelled and verify that you are indeed implementing the correct trait items. Example:

```
struct Bar;

trait Foo {
    const N : u32;

    fn M();
}

impl Foo for Bar {
    const N : u32 = 0;

    fn M() {} // ok!
}
```

Error code E0325

An associated type was implemented when another trait item was expected.

Erroneous code example:

```
struct Bar;

trait Foo {
    const N : u32;
}

impl Foo for Bar {
    type N = u32;
    // error: item `N` is an associated type, which doesn't match its
    //        trait `<Bar as Foo>`
}
```

Please verify that the associated type name wasn't misspelled and your implementation corresponds to the trait definition. Example:

```
struct Bar;

trait Foo {
    type N;
}

impl Foo for Bar {
    type N = u32; // ok!
}
```

Or:

```
struct Bar;

trait Foo {
    const N : u32;
}

impl Foo for Bar {
    const N : u32 = 0; // ok!
}
```

Error code E0326

An implementation of a trait doesn't match the type constraint.

Erroneous code example:

```
trait Foo {
    const BAR: bool;
}

struct Bar;

impl Foo for Bar {
    const BAR: u32 = 5; // error, expected bool, found u32
}
```

The types of any associated constants in a trait implementation must match the types in the trait definition.

Error code E0328

The Unsize trait should not be implemented directly. All implementations of Unsize are provided automatically by the compiler.

Erroneous code example:

```
#![feature(unsize)]

use std::marker::Unsize;

pub struct MyType;

impl<T> Unsize<T> for MyType {}
```

If you are defining your own smart pointer type and would like to enable conversion from a sized to an unsized type with the DST coercion system, use `CoerceUnsized` instead.

```
#![feature(coerce_unsized)]

use std::ops::CoerceUnsized;

pub struct MyType<T: ?Sized> {
    field_with_unsized_type: T,
}

impl<T, U> CoerceUnsized<MyType<U>> for MyType<T>
    where T: CoerceUnsized<U> {}
```

Error code E0329

Note: this error code is no longer emitted by the compiler.

An attempt was made to access an associated constant through either a generic type parameter or `Self`. This is not supported yet. An example causing this error is shown below:

```
trait Foo {
    const BAR: f64;
}

struct MyStruct;

impl Foo for MyStruct {
    const BAR: f64 = 0f64;
}

fn get_bar_bad<F: Foo>(t: F) -> f64 {
    F::BAR
}
```

Currently, the value of `BAR` for a particular type can only be accessed through a concrete type, as shown below:

```
trait Foo {
    const BAR: f64;
}

struct MyStruct;

impl Foo for MyStruct {
    const BAR: f64 = 0f64;
}

fn get_bar_good() -> f64 {
    <MyStruct as Foo>::BAR
}
```

Error code E0364

Private items cannot be publicly re-exported. This error indicates that you attempted to `pub use` a type or value that was not itself public.

Erroneous code example:

```
mod a {
    fn foo() {}

    mod a {
        pub use super::foo; // error!
    }
}
```

The solution to this problem is to ensure that the items that you are re-exporting are themselves marked with `pub`:

```
mod a {
    pub fn foo() {} // ok!

    mod a {
        pub use super::foo;
    }
}
```

See the Use Declarations section of the reference for more information on this topic.

Error code E0365

Private modules cannot be publicly re-exported. This error indicates that you attempted to `pub use` a module that was not itself public.

Erroneous code example:

```
mod foo {
    pub const X: u32 = 1;
}

pub use foo as foo2;

fn main() {}
```

The solution to this problem is to ensure that the module that you are re-exporting is itself marked with `pub`:

```
pub mod foo {
    pub const X: u32 = 1;
}

pub use foo as foo2;

fn main() {}
```

See the Use Declarations section of the reference for more information on this topic.

Error code E0366

An attempt was made to implement `Drop` on a concrete specialization of a generic type. An example is shown below:

```
struct Foo<T> {
    t: T
}

impl Drop for Foo<u32> {
    fn drop(&mut self) {}
}
```

This code is not legal: it is not possible to specialize `Drop` to a subset of implementations of a generic type. One workaround for this is to wrap the generic type, as shown below:

```
struct Foo<T> {
    t: T
}

struct Bar {
    t: Foo<u32>
}

impl Drop for Bar {
    fn drop(&mut self) {}
}
```

Error code E0367

An attempt was made to implement `Drop` on a specialization of a generic type.

Erroneous code example:

```
trait Foo {}

struct MyStruct<T> {
    t: T
}

impl<T: Foo> Drop for MyStruct<T> {
    fn drop(&mut self) {}
}
```

This code is not legal: it is not possible to specialize `Drop` to a subset of implementations of a generic type. In order for this code to work, `MyStruct` must also require that `T` implements `Foo`. Alternatively, another option is to wrap the generic type in another that specializes appropriately:

```
trait Foo{}

struct MyStruct<T> {
    t: T
}

struct MyStructWrapper<T: Foo> {
    t: MyStruct<T>
}

impl <T: Foo> Drop for MyStructWrapper<T> {
    fn drop(&mut self) {}
}
```

Error code E0368

A binary assignment operator like `+=` or `^=` was applied to a type that doesn't support it.

Erroneous code example:

```
let mut x = 12f32; // error: binary operation `<<` cannot be applied to
                   //        type `f32`

x <<= 2;
```

To fix this error, please check that this type implements this binary operation. Example:

```
let mut x = 12u32; // the `u32` type does implement the `ShlAssign` trait

x <<= 2; // ok!
```

It is also possible to overload most operators for your own type by implementing the `[OP]Assign` traits from `std::ops`.

Another problem you might be facing is this: suppose you've overloaded the `+` operator for some type `Foo` by implementing the `std::ops::Add` trait for `Foo`, but you find that using `+=` does not work, as in this example:

```
use std::ops::Add;

struct Foo(u32);

impl Add for Foo {
    type Output = Foo;

    fn add(self, rhs: Foo) -> Foo {
        Foo(self.0 + rhs.0)
    }
}

fn main() {
    let mut x: Foo = Foo(5);
    x += Foo(7); // error, `+=` cannot be applied to the type `Foo`
}
```

This is because `AddAssign` is not automatically implemented, so you need to manually implement it for your type.

Error code E0369

A binary operation was attempted on a type which doesn't support it.

Erroneous code example:

```
let x = 12f32; // error: binary operation `<<` cannot be applied to
               //        type `f32`

x << 2;
```

To fix this error, please check that this type implements this binary operation. Example:

```
let x = 12u32; // the `u32` type does implement it:
               // https://doc.rust-lang.org/stable/std/ops/trait.Shl.html

x << 2; // ok!
```

It is also possible to overload most operators for your own type by implementing traits from `std::ops`.

String concatenation appends the string on the right to the string on the left and may require reallocation. This requires ownership of the string on the left. If something should be added to a string literal, move the literal to the heap by allocating it with `to_owned()` like in `"Your text".to_owned()`.

Error code E0370

The maximum value of an enum was reached, so it cannot be automatically set in the next enum value.

Erroneous code example:

```
#[repr(i64)]
enum Foo {
    X = 0x7fffffffffffffff,
    Y, // error: enum discriminant overflowed on value after
       //        9223372036854775807: i64; set explicitly via
       //        Y = -9223372036854775808 if that is desired outcome
}
```

To fix this, please set manually the next enum value or put the enum variant with the maximum value at the end of the enum. Examples:

```
#[repr(i64)]
enum Foo {
    X = 0x7fffffffffffffff,
    Y = 0, // ok!
}
```

Or:

```
#[repr(i64)]
enum Foo {
    Y = 0, // ok!
    X = 0x7fffffffffffffff,
}
```

Error code E0371

A trait was implemented on another which already automatically implemented it.

Erroneous code examples:

```
trait Foo { fn foo(&self) { } }
trait Bar: Foo { }
trait Baz: Bar { }

impl Bar for Baz { } // error, `Baz` implements `Bar` by definition
impl Foo for Baz { } // error, `Baz` implements `Bar` which implements `Foo`
impl Baz for Baz { } // error, `Baz` (trivially) implements `Baz`
impl Baz for Bar { } // Note: This is OK
```

When `Trait2` is a subtrait of `Trait1` (for example, when `Trait2` has a definition like `trait Trait2: Trait1 { ... }`), it is not allowed to implement `Trait1` for `Trait2`. This is because `Trait2` already implements `Trait1` by definition, so it is not useful to do this.

Error code E0373

A captured variable in a closure may not live long enough.

Erroneous code example:

```
fn foo() -> Box<Fn(u32) -> u32> {
    let x = 0u32;
    Box::new(|y| x + y)
}
```

This error occurs when an attempt is made to use data captured by a closure, when that data may no longer exist. It's most commonly seen when attempting to return a closure as shown in the previous code example.

Notice that `x` is stack-allocated by `foo()`. By default, Rust captures closed-over data by reference. This means that once `foo()` returns, `x` no longer exists. An attempt to access `x` within the closure would thus be unsafe.

Another situation where this might be encountered is when spawning threads:

```
fn foo() {
    let x = 0u32;
    let y = 1u32;

    let thr = std::thread::spawn(|| {
        x + y
    });
}
```

Since our new thread runs in parallel, the stack frame containing `x` and `y` may well have disappeared by the time we try to use them. Even if we call `thr.join()` within foo (which blocks until `thr` has completed, ensuring the stack frame won't disappear), we will not succeed: the compiler cannot prove that this behavior is safe, and so won't let us do it.

The solution to this problem is usually to switch to using a `move` closure. This approach moves (or copies, where possible) data into the closure, rather than taking references to it. For example:

```
fn foo() -> Box<Fn(u32) -> u32> {
    let x = 0u32;
    Box::new(move |y| x + y)
}
```

Now that the closure has its own copy of the data, there's no need to worry about safety.

This error may also be encountered while using `async` blocks:

```
use std::future::Future;

async fn f() {
    let v = vec![1, 2, 3i32];
    spawn(async { //~ ERROR E0373
        println!("{:?}", v)
    });
}

fn spawn<F: Future + Send + 'static>(future: F) {
    unimplemented!()
}
```

Similarly to closures, `async` blocks are not executed immediately and may capture closed-over data by reference. For more information, see https://rust-lang.github.io/async-book/03_async_await/01_chapter.html.

Error code E0374

`CoerceUnsized` was implemented on a struct which does not contain a field with an unsized type.

Example of erroneous code:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

struct Foo<T: ?Sized> {
    a: i32,
}

// error: Struct `Foo` has no unsized fields that need `CoerceUnsized`.
impl<T, U> CoerceUnsized<Foo<U>> for Foo<T>
    where T: CoerceUnsized<U> {}
```

An unsized type is any type where the compiler does not know the length or alignment of at compile time. Any struct containing an unsized type is also unsized.

`CoerceUnsized` is used to coerce one struct containing an unsized type into another struct containing a different unsized type. If the struct doesn't have any fields of unsized types then you don't need explicit coercion to get the types you want. To fix this you can either not try to implement `CoerceUnsized` or you can add a field that is unsized to the struct.

Example:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

// We don't need to impl `CoerceUnsized` here.
struct Foo {
    a: i32,
}

// We add the unsized type field to the struct.
struct Bar<T: ?Sized> {
    a: i32,
    b: T,
}

// The struct has an unsized field so we can implement
// `CoerceUnsized` for it.
impl<T, U> CoerceUnsized<Bar<U>> for Bar<T>
    where T: CoerceUnsized<U> {}
```

Note that `CoerceUnsized` is mainly used by smart pointers like `Box`, `Rc` and `Arc` to be able to mark that they can coerce unsized types that they are pointing at.

Error code E0375

`CoerceUnsized` was implemented on a struct which contains more than one field with an unsized type.

Erroneous code example:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

struct Foo<T: ?Sized, U: ?Sized> {
    a: i32,
    b: T,
    c: U,
}

// error: Struct `Foo` has more than one unsized field.
impl<T, U> CoerceUnsized<Foo<U, T>> for Foo<T, U> {}
```

A struct with more than one field containing an unsized type cannot implement `CoerceUnsized`. This only occurs when you are trying to coerce one of the types in your struct to another type in the struct. In this case we try to impl `CoerceUnsized` from `T` to `U` which are both types that the struct takes. An unsized type is any type that the compiler doesn't know the length or alignment of at compile time. Any struct containing an unsized type is also unsized.

`CoerceUnsized` only allows for coercion from a structure with a single unsized type field to another struct with a single unsized type field. In fact Rust only allows for a struct to have one unsized type in a struct and that unsized type must be the last field in the struct. So having two unsized types in a single struct is not allowed by the compiler. To fix this use only one field containing an unsized type in the struct and then use multiple structs to manage each unsized type field you need.

Example:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

struct Foo<T: ?Sized> {
    a: i32,
    b: T,
}

impl <T, U> CoerceUnsized<Foo<U>> for Foo<T>
    where T: CoerceUnsized<U> {}

fn coerce_foo<T: CoerceUnsized<U>, U>(t: T) -> Foo<U> {
    Foo { a: 12i32, b: t } // we use coercion to get the `Foo<U>` type we
need
}
```

Error code E0376

`CoerceUnsized` was implemented on something that isn't a struct.

Erroneous code example:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

struct Foo<T: ?Sized> {
    a: T,
}

// error: The type `U` is not a struct
impl<T, U> CoerceUnsized<U> for Foo<T> {}
```

`CoerceUnsized` can only be implemented for a struct. Unsized types are already able to be coerced without an implementation of `CoerceUnsized` whereas a struct containing an unsized type needs to know the unsized type field it's containing is able to be coerced. An unsized type is any type that the compiler doesn't know the length or alignment of at compile time. Any struct containing an unsized type is also unsized.

The `CoerceUnsized` trait takes a struct type. Make sure the type you are providing to `CoerceUnsized` is a struct with only the last field containing an unsized type.

Example:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

struct Foo<T> {
    a: T,
}

// The `Foo<U>` is a struct so `CoerceUnsized` can be implemented
impl<T, U> CoerceUnsized<Foo<U>> for Foo<T> where T: CoerceUnsized<U> {}
```

Note that in Rust, structs can only contain an unsized type if the field containing the unsized type is the last and only unsized type field in the struct.

Error code E0377

The trait `CoerceUnsized` may only be implemented for a coercion between structures with the same definition.

Example of erroneous code:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

pub struct Foo<T: ?Sized> {
    field_with_unsized_type: T,
}

pub struct Bar<T: ?Sized> {
    field_with_unsized_type: T,
}

// error: the trait `CoerceUnsized` may only be implemented for a coercion
//        between structures with the same definition
impl<T, U> CoerceUnsized<Bar<U>> for Foo<T> where T: CoerceUnsized<U> {}
```

When attempting to implement `CoerceUnsized`, the `impl` signature must look like: `impl CoerceUnsized<Type<U>> for Type<T> where T: CoerceUnsized<U>`; the *implementer* and *`CoerceUnsized` type parameter* must be the same type. In this example, `Bar` and `Foo` (even though structurally identical) are *not* the same type and are rejected. Learn more about the `CoerceUnsized` trait and DST coercion in the `CoerceUnsized` docs.

Error code E0378

The `DispatchFromDyn` trait was implemented on something which is not a pointer or a newtype wrapper around a pointer.

Erroneous code example:

```
#![feature(dispatch_from_dyn)]
use std::ops::DispatchFromDyn;

struct WrapperExtraField<T> {
    ptr: T,
    extra_stuff: i32,
}

impl<T, U> DispatchFromDyn<WrapperExtraField<U>> for WrapperExtraField<T>
where
    T: DispatchFromDyn<U>,
{}
```

The `DispatchFromDyn` trait currently can only be implemented for builtin pointer types and structs that are newtype wrappers around them — that is, the struct must have only one field (except for `PhantomData`), and that field must itself implement `DispatchFromDyn`.

```
#![feature(dispatch_from_dyn, unsize)]
use std::{
    marker::Unsize,
    ops::DispatchFromDyn,
};

struct Ptr<T: ?Sized>(*const T);

impl<T: ?Sized, U: ?Sized> DispatchFromDyn<Ptr<U>> for Ptr<T>
where
    T: Unsize<U>,
{}
```

Another example:

```
#![feature(dispatch_from_dyn)]
use std::{
    ops::DispatchFromDyn,
    marker::PhantomData,
};

struct Wrapper<T> {
    ptr: T,
    _phantom: PhantomData<()>,
}

impl<T, U> DispatchFromDyn<Wrapper<U>> for Wrapper<T>
where
    T: DispatchFromDyn<U>,
{}
```

Error code E0379

A trait method was declared const.

Erroneous code example:

```
trait Foo {
    const fn bar() -> u32; // error!
}
```

Trait methods cannot be declared `const` by design. For more information, see RFC 911.

Error code E0380

An auto trait was declared with a method or an associated item.

Erroneous code example:

```
unsafe auto trait Trait {
    type Output; // error!
}
```

Auto traits cannot have methods or associated items. For more information see the opt-in builtin traits RFC.

Error code E0381

It is not allowed to use or capture an uninitialized variable.

Erroneous code example:

```
fn main() {
    let x: i32;
    let y = x; // error, use of possibly-uninitialized variable
}
```

To fix this, ensure that any declared variables are initialized before being used. Example:

```
fn main() {
    let x: i32 = 0;
    let y = x; // ok!
}
```

Error code E0382

A variable was used after its contents have been moved elsewhere.

Erroneous code example:

```
struct MyStruct { s: u32 }

fn main() {
    let mut x = MyStruct{ s: 5u32 };
    let y = x;
    x.s = 6;
    println!("{}", x.s);
}
```

Since `MyStruct` is a type that is not marked `Copy`, the data gets moved out of `x` when we set `y`. This is fundamental to Rust's ownership system: outside of workarounds like `Rc`, a value cannot be owned by more than one variable.

Sometimes we don't need to move the value. Using a reference, we can let another function borrow the value without changing its ownership. In the example below, we don't actually have to move our string to `calculate_length`, we can give it a reference to it with `&` instead.

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

A mutable reference can be created with `&mut`.

Sometimes we don't want a reference, but a duplicate. All types marked `Clone` can be duplicated by calling `.clone()`. Subsequent changes to a clone do not affect the original variable.

Most types in the standard library are marked `Clone`. The example below demonstrates using `clone()` on a string. `s1` is first set to "many", and then copied to `s2`. Then the first character of `s1` is removed, without affecting `s2`. "any many" is printed to the console.

```
fn main() {
    let mut s1 = String::from("many");
    let s2 = s1.clone();
    s1.remove(0);
    println!("{} {}", s1, s2);
}
```

If we control the definition of a type, we can implement `Clone` on it ourselves with `#[derive(Clone)]`.

Some types have no ownership semantics at all and are trivial to duplicate. An example is `i32` and the other number types. We don't have to call `.clone()` to clone them, because they are marked `Copy` in addition to `Clone`. Implicit cloning is more convenient in this case. We can mark our own types `Copy` if all their members also are marked `Copy`.

In the example below, we implement a `Point` type. Because it only stores two integers, we opt-out of ownership semantics with `Copy`. Then we can `let p2 = p1` without `p1` being moved.

```
#[derive(Copy, Clone)]
struct Point { x: i32, y: i32 }

fn main() {
    let mut p1 = Point{ x: -1, y: 2 };
    let p2 = p1;
    p1.x = 1;
    println!("p1: {}, {}", p1.x, p1.y);
    println!("p2: {}, {}", p2.x, p2.y);
}
```

Alternatively, if we don't control the struct's definition, or mutable shared ownership is truly required, we can use `Rc` and `RefCell`:

```
use std::cell::RefCell;
use std::rc::Rc;

struct MyStruct { s: u32 }

fn main() {
    let mut x = Rc::new(RefCell::new(MyStruct{ s: 5u32 }));
    let y = x.clone();
    x.borrow_mut().s = 6;
    println!("{}", x.borrow().s);
}
```

With this approach, x and y share ownership of the data via the `Rc` (reference count type). `RefCell` essentially performs runtime borrow checking: ensuring that at most one writer or multiple readers can access the data at any one time.

If you wish to learn more about ownership in Rust, start with the Understanding Ownership chapter in the Book.

Error code E0383

Note: this error code is no longer emitted by the compiler.

This error occurs when an attempt is made to partially reinitialize a structure that is currently uninitialized.

For example, this can happen when a drop has taken place:

```
struct Foo {
    a: u32,
}
impl Drop for Foo {
    fn drop(&mut self) { /* ... */ }
}

let mut x = Foo { a: 1 };
drop(x); // `x` is now uninitialized
x.a = 2; // error, partial reinitialization of uninitialized structure `t`
```

This error can be fixed by fully reinitializing the structure in question:

```
struct Foo {
    a: u32,
}
impl Drop for Foo {
    fn drop(&mut self) { /* ... */ }
}

let mut x = Foo { a: 1 };
drop(x);
x = Foo { a: 2 };
```

Error code E0384

An immutable variable was reassigned.

Erroneous code example:

```
fn main() {
    let x = 3;
    x = 5; // error, reassignment of immutable variable
}
```

By default, variables in Rust are immutable. To fix this error, add the keyword `mut` after the keyword `let` when declaring the variable. For example:

```
fn main() {
    let mut x = 3;
    x = 5;
}
```

Error code E0386

Note: this error code is no longer emitted by the compiler.

This error occurs when an attempt is made to mutate the target of a mutable reference stored inside an immutable container.

For example, this can happen when storing a `&mut` inside an immutable `Box`:

```
let mut x: i64 = 1;
let y: Box<_> = Box::new(&mut x);
**y = 2; // error, cannot assign to data in an immutable container
```

This error can be fixed by making the container mutable:

```
let mut x: i64 = 1;
let mut y: Box<_> = Box::new(&mut x);
**y = 2;
```

It can also be fixed by using a type with interior mutability, such as `Cell` or `RefCell`:

```
use std::cell::Cell;

let x: i64 = 1;
let y: Box<Cell<_>> = Box::new(Cell::new(x));
y.set(2);
```

Error code E0387

Note: this error code is no longer emitted by the compiler.

This error occurs when an attempt is made to mutate or mutably reference data that a closure has captured immutably.

Erroneous code example:

```
// Accepts a function or a closure that captures its environment immutably.
// Closures passed to foo will not be able to mutate their closed-over
state.
fn foo<F: Fn()>(f: F) { }

// Attempts to mutate closed-over data. Error message reads:
// `cannot assign to data in a captured outer variable...`
fn mutable() {
    let mut x = 0u32;
    foo(|| x = 2);
}

// Attempts to take a mutable reference to closed-over data. Error message
// reads: `cannot borrow data mutably in a captured outer variable...`
fn mut_addr() {
    let mut x = 0u32;
    foo(|| { let y = &mut x; });
}
```

The problem here is that foo is defined as accepting a parameter of type `Fn`. Closures passed into foo will thus be inferred to be of type `Fn`, meaning that they capture their context immutably.

If the definition of `foo` is under your control, the simplest solution is to capture the data mutably. This can be done by defining `foo` to take FnMut rather than Fn:

```
fn foo<F: FnMut()>(f: F) { }
```

Alternatively, we can consider using the `Cell` and `RefCell` types to achieve interior mutability through a shared reference. Our example's `mutable` function could be redefined as below:

```
use std::cell::Cell;

fn foo<F: Fn()>(f: F) { }

fn mutable() {
    let x = Cell::new(0u32);
    foo(|| x.set(2));
}
```

You can read more in the API documentation for Cell.

Error code E0388

Note: this error code is no longer emitted by the compiler.

Error code E0389

Note: this error code is no longer emitted by the compiler.

An attempt was made to mutate data using a non-mutable reference. This commonly occurs when attempting to assign to a non-mutable reference of a mutable reference (`&(&mut T)`).

Erroneous code example:

```rust
struct FancyNum {
    num: u8,
}

fn main() {
    let mut fancy = FancyNum{ num: 5 };
    let fancy_ref = &(&mut fancy);
    fancy_ref.num = 6; // error: cannot assign to data in a `&` reference
    println!("{}", fancy_ref.num);
}
```

Here, `&mut fancy` is mutable, but `&(&mut fancy)` is not. Creating an immutable reference to a value borrows it immutably. There can be multiple references of type `&(&mut T)` that point to the same value, so they must be immutable to prevent multiple mutable references to the same value.

To fix this, either remove the outer reference:

```rust
struct FancyNum {
    num: u8,
}

fn main() {
    let mut fancy = FancyNum{ num: 5 };

    let fancy_ref = &mut fancy;
    // `fancy_ref` is now &mut FancyNum, rather than &(&mut FancyNum)

    fancy_ref.num = 6; // No error!

    println!("{}", fancy_ref.num);
}
```

Or make the outer reference mutable:

```
struct FancyNum {
    num: u8
}

fn main() {
    let mut fancy = FancyNum{ num: 5 };

    let fancy_ref = &mut (&mut fancy);
    // `fancy_ref` is now &mut(&mut FancyNum), rather than &(&mut FancyNum)

    fancy_ref.num = 6; // No error!

    println!("{}", fancy_ref.num);
}
```

Error code E0390

A method or constant was implemented on a primitive type.

Erroneous code example:

```
struct Foo {
    x: i32
}

impl *mut Foo {}
// error: cannot define inherent `impl` for primitive types
```

This isn't allowed, but using a trait to implement a method or constant is a good solution. Example:

```
struct Foo {
    x: i32
}

trait Bar {
    fn bar();
}

impl Bar for *mut Foo {
    fn bar() {} // ok!
}
```

Instead of defining an inherent implementation on a reference, you could also move the reference inside the implementation:

```
struct Foo;

impl &Foo { // error: no nominal type found for inherent implementation
    fn bar(self, other: Self) {}
}
```

becomes

```
struct Foo;

impl Foo {
    fn bar(&self, other: &Self) {}
}
```

Error code E0391

A type dependency cycle has been encountered.

Erroneous code example:

```
trait FirstTrait : SecondTrait {

}

trait SecondTrait : FirstTrait {

}
```

The previous example contains a circular dependency between two traits: `FirstTrait` depends on `SecondTrait` which itself depends on `FirstTrait`.

See https://rustc-dev-guide.rust-lang.org/overview.html#queries and https://rustc-dev-guide.rust-lang.org/query.html for more information.

Error code E0392

A type or lifetime parameter has been declared but is not actually used.

Erroneous code example:

```
enum Foo<T> {
    Bar,
}
```

If the type parameter was included by mistake, this error can be fixed by simply removing the type parameter, as shown below:

```
enum Foo {
    Bar,
}
```

Alternatively, if the type parameter was intentionally inserted, it must be used. A simple fix is shown below:

```
enum Foo<T> {
    Bar(T),
}
```

This error may also commonly be found when working with unsafe code. For example, when using raw pointers one may wish to specify the lifetime for which the pointed-at data is valid. An initial attempt (below) causes this error:

```
struct Foo<'a, T> {
    x: *const T,
}
```

We want to express the constraint that Foo should not outlive `'a`, because the data pointed to by `T` is only valid for that lifetime. The problem is that there are no actual uses of `'a`. It's possible to work around this by adding a PhantomData type to the struct, using it to tell the compiler to act as if the struct contained a borrowed reference `&'a T`:

```
use std::marker::PhantomData;

struct Foo<'a, T: 'a> {
    x: *const T,
    phantom: PhantomData<&'a T>
}
```

PhantomData can also be used to express information about unused type parameters.

Error code E0393

A type parameter which references `Self` in its default value was not specified.

Erroneous code example:

```
trait A<T=Self> {}

fn together_we_will_rule_the_galaxy(son: &A) {}
// error: the type parameter `T` must be explicitly specified in an
//        object type because its default value `Self` references the
//        type `Self`
```

A trait object is defined over a single, fully-defined trait. With a regular default parameter, this parameter can just be substituted in. However, if the default parameter is `Self`, the trait changes for each concrete type; i.e. `i32` will be expected to implement `A<i32>`, `bool` will be expected to implement `A<bool>`, etc... These types will not share an implementation of a fully-defined trait; instead they share implementations of a trait with different parameters substituted in for each implementation. This is irreconcilable with what we need to make a trait object work, and is thus disallowed. Making the trait concrete by explicitly specifying the value of the defaulted parameter will fix this issue. Fixed example:

```
trait A<T=Self> {}

fn together_we_will_rule_the_galaxy(son: &A<i32>) {} // Ok!
```

Error code E0398

Note: this error code is no longer emitted by the compiler.

In Rust 1.3, the default object lifetime bounds are expected to change, as described in RFC 1156. You are getting a warning because the compiler thinks it is possible that this change will cause a compilation error in your code. It is possible, though unlikely, that this is a false alarm.

The heart of the change is that where `&'a Box<SomeTrait>` used to default to `&'a Box<SomeTrait+'a>` , it now defaults to `&'a Box<SomeTrait+'static>` (here, `SomeTrait` is the name of some trait type). Note that the only types which are affected are references to boxes, like `&Box<SomeTrait>` or `&[Box<SomeTrait>]` . More common types like `&SomeTrait` or `Box<SomeTrait>` are unaffected.

To silence this warning, edit your code to use an explicit bound. Most of the time, this means that you will want to change the signature of a function that you are calling. For example, if the error is reported on a call like `foo(x)` , and `foo` is defined as follows:

```
fn foo(arg: &Box<SomeTrait>) { /* ... */ }
```

You might change it to:

```
fn foo<'a>(arg: &'a Box<SomeTrait+'a>) { /* ... */ }
```

This explicitly states that you expect the trait object `SomeTrait` to contain references (with a maximum lifetime of `'a`).

Error code E0399

Note: this error code is no longer emitted by the compiler

You implemented a trait, overriding one or more of its associated types but did not reimplement its default methods.

Example of erroneous code:

```
#![feature(associated_type_defaults)]

pub trait Foo {
    type Assoc = u8;
    fn bar(&self) {}
}

impl Foo for i32 {
    // error - the following trait items need to be reimplemented as
    //         `Assoc` was overridden: `bar`
    type Assoc = i32;
}
```

To fix this, add an implementation for each default method from the trait:

```
#![feature(associated_type_defaults)]

pub trait Foo {
    type Assoc = u8;
    fn bar(&self) {}
}

impl Foo for i32 {
    type Assoc = i32;
    fn bar(&self) {} // ok!
}
```

Error code E0401

Inner items do not inherit the generic parameters from the items they are embedded in.

Erroneous code example:

```
fn foo<T>(x: T) {
    fn bar(y: T) { // T is defined in the "outer" function
        // ..
    }
    bar(x);
}
```

Nor will this:

```
fn foo<T>(x: T) {
    type MaybeT = Option<T>;
    // ...
}
```

Or this:

```
fn foo<T>(x: T) {
    struct Foo {
        x: T,
    }
    // ...
}
```

Items nested inside other items are basically just like top-level items, except that they can only be used from the item they are in.

There are a couple of solutions for this.

If the item is a function, you may use a closure:

```
fn foo<T>(x: T) {
    let bar = |y: T| { // explicit type annotation may not be necessary
        // ..
    };
    bar(x);
}
```

For a generic item, you can copy over the parameters:

```
fn foo<T>(x: T) {
    fn bar<T>(y: T) {
        // ..
    }
    bar(x);
}
```

```
fn foo<T>(x: T) {
    type MaybeT<T> = Option<T>;
}
```

Be sure to copy over any bounds as well:

```
fn foo<T: Copy>(x: T) {
    fn bar<T: Copy>(y: T) {
        // ..
    }
    bar(x);
}
```

```
fn foo<T: Copy>(x: T) {
    struct Foo<T: Copy> {
        x: T,
    }
}
```

This may require additional type hints in the function body.

In case the item is a function inside an `impl`, defining a private helper function might be easier:

```
impl<T> Foo<T> {
    pub fn foo(&self, x: T) {
        self.bar(x);
    }

    fn bar(&self, y: T) {
        // ..
    }
}
```

For default impls in traits, the private helper solution won't work, however closures or copying the parameters should still work.

Error code E0403

Some type parameters have the same name.

Erroneous code example:

```
fn f<T, T>(s: T, u: T) {} // error: the name `T` is already used for a
generic
                          //        parameter in this item's generic
parameters
```

Please verify that none of the type parameters are misspelled, and rename any clashing parameters. Example:

```
fn f<T, Y>(s: T, u: Y) {} // ok!
```

Type parameters in an associated item also cannot shadow parameters from the containing item:

```
trait Foo<T> {
    fn do_something(&self) -> T;
    fn do_something_else<T: Clone>(&self, bar: T);
}
```

Error code E0404

A type that is not a trait was used in a trait position, such as a bound or `impl`.

Erroneous code example:

```
struct Foo;
struct Bar;

impl Foo for Bar {} // error: `Foo` is not a trait
fn baz<T: Foo>(t: T) {} // error: `Foo` is not a trait
```

Another erroneous code example:

```
type Foo = Iterator<Item=String>;

fn bar<T: Foo>(t: T) {} // error: `Foo` is a type alias
```

Please verify that the trait's name was not misspelled or that the right identifier was used. Example:

```
trait Foo {
    // some functions
}
struct Bar;

impl Foo for Bar { // ok!
    // functions implementation
}

fn baz<T: Foo>(t: T) {} // ok!
```

Alternatively, you could introduce a new trait with your desired restrictions as a super trait:

```
trait Qux: Foo {} // Anything that implements Qux also needs to implement Foo
fn baz<T: Qux>(t: T) {} // also ok!
```

Finally, if you are on nightly and want to use a trait alias instead of a type alias, you should use `#![feature(trait_alias)]`:

```
#![feature(trait_alias)]
trait Foo = Iterator<Item=String>;

fn bar<T: Foo>(t: T) {} // ok!
```

Error code E0405

The code refers to a trait that is not in scope.

Erroneous code example:

```
struct Foo;

impl SomeTrait for Foo {} // error: trait `SomeTrait` is not in scope
```

Please verify that the name of the trait wasn't misspelled and ensure that it was imported. Example:

```
// solution 1:
use some_file::SomeTrait;

// solution 2:
trait SomeTrait {
    // some functions
}

struct Foo;

impl SomeTrait for Foo { // ok!
    // implements functions
}
```

Error code E0407

A definition of a method not in the implemented trait was given in a trait implementation.

Erroneous code example:

```
trait Foo {
    fn a();
}

struct Bar;

impl Foo for Bar {
    fn a() {}
    fn b() {} // error: method `b` is not a member of trait `Foo`
}
```

Please verify you didn't misspell the method name and you used the correct trait. First example:

```
trait Foo {
    fn a();
    fn b();
}

struct Bar;

impl Foo for Bar {
    fn a() {}
    fn b() {} // ok!
}
```

Second example:

```
trait Foo {
    fn a();
}

struct Bar;

impl Foo for Bar {
    fn a() {}
}

impl Bar {
    fn b() {}
}
```

Error code E0408

An "or" pattern was used where the variable bindings are not consistently bound across patterns.

Erroneous code example:

```
match x {
    Some(y) | None => { /* use y */ } // error: variable `y` from pattern #1
is
                                      //        not bound in pattern #2
    _ => ()
}
```

Here, `y` is bound to the contents of the `Some` and can be used within the block corresponding to the match arm. However, in case `x` is `None`, we have not specified what `y` is, and the block will use a nonexistent variable.

To fix this error, either split into multiple match arms:

```
let x = Some(1);
match x {
    Some(y) => { /* use y */ }
    None => { /* ... */ }
}
```

or, bind the variable to a field of the same type in all sub-patterns of the or pattern:

```
let x = (0, 2);
match x {
    (0, y) | (y, 0) => { /* use y */}
    _ => {}
}
```

In this example, if `x` matches the pattern `(0, _)`, the second field is set to `y`. If it matches `(_, 0)`, the first field is set to `y`; so in all cases `y` is set to some value.

Error code E0409

An "or" pattern was used where the variable bindings are not consistently bound across patterns.

Erroneous code example:

```
let x = (0, 2);
match x {
    (0, ref y) | (y, 0) => { /* use y */} // error: variable `y` is bound
with
                                          //        different mode in
pattern #2
                                          //        than in pattern #1
    _ => ()
}
```

Here, `y` is bound by-value in one case and by-reference in the other.

To fix this error, just use the same mode in both cases. Generally using `ref` or `ref mut` where not already used will fix this:

```
let x = (0, 2);
match x {
    (0, ref y) | (ref y, 0) => { /* use y */}
    _ => ()
}
```

Alternatively, split the pattern:

```
let x = (0, 2);
match x {
    (y, 0) => { /* use y */ }
    (0, ref y) => { /* use y */}
    _ => ()
}
```

Error code E0411

The `Self` keyword was used outside an impl, trait, or type definition.

Erroneous code example:

```
<Self>::foo; // error: use of `Self` outside of an impl, trait, or type
             // definition
```

The `Self` keyword represents the current type, which explains why it can only be used inside an impl, trait, or type definition. It gives access to the associated items of a type:

```
trait Foo {
    type Bar;
}

trait Baz : Foo {
    fn bar() -> Self::Bar; // like this
}
```

However, be careful when two types have a common associated type:

```
trait Foo {
    type Bar;
}

trait Foo2 {
    type Bar;
}

trait Baz : Foo + Foo2 {
    fn bar() -> Self::Bar;
    // error: ambiguous associated type `Bar` in bounds of `Self`
}
```

This problem can be solved by specifying from which trait we want to use the `Bar` type:

```
trait Foo {
    type Bar;
}

trait Foo2 {
    type Bar;
}

trait Baz : Foo + Foo2 {
    fn bar() -> <Self as Foo>::Bar; // ok!
}
```

Error code E0412

A used type name is not in scope.

Erroneous code examples:

```
impl Something {} // error: type name `Something` is not in scope

// or:

trait Foo {
    fn bar(N); // error: type name `N` is not in scope
}

// or:

fn foo(x: T) {} // type name `T` is not in scope
```

To fix this error, please verify you didn't misspell the type name, you did declare it or imported it into the scope. Examples:

```
struct Something;

impl Something {} // ok!

// or:

trait Foo {
    type N;

    fn bar(_: Self::N); // ok!
}

// or:

fn foo<T>(x: T) {} // ok!
```

Another case that causes this error is when a type is imported into a parent module. To fix this, you can follow the suggestion and use File directly or `use super::File;` which will import the types from the parent namespace. An example that causes this error is below:

```
use std::fs::File;

mod foo {
    fn some_function(f: File) {}
}
```

```
use std::fs::File;

mod foo {
    // either
    use super::File;
    // or
    // use std::fs::File;
    fn foo(f: File) {}
}
```

Error code E0415

More than one function parameter have the same name.

Erroneous code example:

```
fn foo(f: i32, f: i32) {} // error: identifier `f` is bound more than
                          //        once in this parameter list
```

Please verify you didn't misspell parameters' name. Example:

```
fn foo(f: i32, g: i32) {} // ok!
```

Error code E0416

An identifier is bound more than once in a pattern.

Erroneous code example:

```
match (1, 2) {
    (x, x) => {} // error: identifier `x` is bound more than once in the
                 //        same pattern
}
```

Please verify you didn't misspell identifiers' name. Example:

```
match (1, 2) {
    (x, y) => {} // ok!
}
```

Or maybe did you mean to unify? Consider using a guard:

```
match (A, B, C) {
    (x, x2, see) if x == x2 => { /* A and B are equal, do one thing */ }
    (y, z, see) => { /* A and B not equal; do another thing */ }
}
```

Error code E0422

An identifier that is neither defined nor a struct was used.

Erroneous code example:

```
fn main () {
    let x = Foo { x: 1, y: 2 };
}
```

In this case, `Foo` is undefined, so it inherently isn't anything, and definitely not a struct.

```
fn main () {
    let foo = 1;
    let x = foo { x: 1, y: 2 };
}
```

In this case, `foo` is defined, but is not a struct, so Rust can't use it as one.

Error code E0423

An identifier was used like a function name or a value was expected and the identifier exists but it belongs to a different namespace.

Erroneous code example:

```
struct Foo { a: bool };

let f = Foo();
// error: expected function, tuple struct or tuple variant, found `Foo`
// `Foo` is a struct name, but this expression uses it like a function name
```

Please verify you didn't misspell the name of what you actually wanted to use here. Example:

```
fn Foo() -> u32 { 0 }

let f = Foo(); // ok!
```

It is common to forget the trailing `!` on macro invocations, which would also yield this error:

```
println("");
// error: expected function, tuple struct or tuple variant,
// found macro `println`
// did you mean `println!(...)`? (notice the trailing `!`)
```

Another case where this error is emitted is when a value is expected, but something else is found:

```
pub mod a {
    pub const I: i32 = 1;
}

fn h1() -> i32 {
    a.I
    //~^ ERROR expected value, found module `a`
    // did you mean `a::I`?
}
```

Error code E0424

The `self` keyword was used inside of an associated function without a "`self` receiver" parameter.

Erroneous code example:

```
struct Foo;

impl Foo {
    // `bar` is a method, because it has a receiver parameter.
    fn bar(&self) {}

    // `foo` is not a method, because it has no receiver parameter.
    fn foo() {
        self.bar(); // error: `self` value is a keyword only available in
                    //        methods with a `self` parameter
    }
}
```

The `self` keyword can only be used inside methods, which are associated functions (functions defined inside of a `trait` or `impl` block) that have a `self` receiver as its first parameter, like `self`, `&self`, `&mut self` or `self: &mut Pin<Self>` (this last one is an example of an "arbitrary `self` type").

Check if the associated function's parameter list should have contained a `self` receiver for it to be a method, and add it if so. Example:

```
struct Foo;

impl Foo {
    fn bar(&self) {}

    fn foo(self) { // `foo` is now a method.
        self.bar(); // ok!
    }
}
```

Error code E0425

An unresolved name was used.

Erroneous code examples:

```
something_that_doesnt_exist::foo;
// error: unresolved name `something_that_doesnt_exist::foo`

// or:

trait Foo {
    fn bar() {
        Self; // error: unresolved name `Self`
    }
}

// or:

let x = unknown_variable;  // error: unresolved name `unknown_variable`
```

Please verify that the name wasn't misspelled and ensure that the identifier being referred to is valid for the given situation. Example:

```
enum something_that_does_exist {
    Foo,
}
```

Or:

```
mod something_that_does_exist {
    pub static foo : i32 = 0i32;
}

something_that_does_exist::foo; // ok!
```

Or:

```
let unknown_variable = 12u32;
let x = unknown_variable; // ok!
```

If the item is not defined in the current module, it must be imported using a `use` statement, like so:

```
use foo::bar;
bar();
```

If the item you are importing is not defined in some super-module of the current module, then it must also be declared as public (e.g., `pub fn`).

Error code E0426

An undeclared label was used.

Erroneous code example:

```
loop {
    break 'a; // error: use of undeclared label `'a`
}
```

Please verify you spelled or declared the label correctly. Example:

```
'a: loop {
    break 'a; // ok!
}
```

Error code E0428

A type or module has been defined more than once.

Erroneous code example:

```
struct Bar;
struct Bar; // error: duplicate definition of value `Bar`
```

Please verify you didn't misspell the type/module's name or remove/rename the duplicated one. Example:

```
struct Bar;
struct Bar2; // ok!
```

Error code E0429

The `self` keyword cannot appear alone as the last segment in a `use` declaration.

Erroneous code example:

```
use std::fmt::self; // error: `self` imports are only allowed within a { }
list
```

To use a namespace itself in addition to some of its members, `self` may appear as part of a brace-enclosed list of imports:

```
use std::fmt::{self, Debug};
```

If you only want to import the namespace, do so directly:

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

Error code E0430

The `self` import appears more than once in the list.

Erroneous code example:

```
use something::{self, self}; // error: `self` import can only appear once in
                             //        the list
```

Please verify you didn't misspell the import name or remove the duplicated `self` import. Example:

```
use something::{self}; // ok!
```

Error code E0431

An invalid `self` import was made.

Erroneous code example:

```
use {self}; // error: `self` import can only appear in an import list with a
            //        non-empty prefix
```

You cannot import the current module into itself, please remove this import or verify you didn't misspell it.

Error code E0432

An import was unresolved.

Erroneous code example:

```
use something::Foo; // error: unresolved import `something::Foo`.
```

In Rust 2015, paths in `use` statements are relative to the crate root. To import items relative to the current and parent modules, use the `self::` and `super::` prefixes, respectively.

In Rust 2018 or later, paths in `use` statements are relative to the current module unless they begin with the name of a crate or a literal `crate::`, in which case they start from the crate root. As in Rust 2015 code, the `self::` and `super::` prefixes refer to the current and parent modules respectively.

Also verify that you didn't misspell the import name and that the import exists in the module from where you tried to import it. Example:

```
use self::something::Foo; // Ok.

mod something {
    pub struct Foo;
}
```

If you tried to use a module from an external crate and are using Rust 2015, you may have missed the `extern crate` declaration (which is usually placed in the crate root):

```
extern crate core; // Required to use the `core` crate in Rust 2015.

use core::any;
```

Since Rust 2018 the `extern crate` declaration is not required and you can instead just `use` it:

```
use core::any; // No extern crate required in Rust 2018.
```

Error code E0433

An undeclared crate, module, or type was used.

Erroneous code example:

```
let map = HashMap::new();
// error: failed to resolve: use of undeclared type `HashMap`
```

Please verify you didn't misspell the type/module's name or that you didn't forget to import it:

```
use std::collections::HashMap; // HashMap has been imported.
let map: HashMap<u32, u32> = HashMap::new(); // So it can be used!
```

If you've expected to use a crate name:

```
use ferris_wheel::BigO;
// error: failed to resolve: use of undeclared crate or module
`ferris_wheel`
```

Make sure the crate has been added as a dependency in `Cargo.toml` .

To use a module from your current crate, add the `crate::` prefix to the path.

Error code E0434

A variable used inside an inner function comes from a dynamic environment.

Erroneous code example:

```
fn foo() {
    let y = 5;
    fn bar() -> u32 {
        y // error: can't capture dynamic environment in a fn item; use the
          //        || { ... } closure form instead.
    }
}
```

Inner functions do not have access to their containing environment. To fix this error, you can replace the function with a closure:

```
fn foo() {
    let y = 5;
    let bar = || {
        y
    };
}
```

Or replace the captured variable with a constant or a static item:

```
fn foo() {
    static mut X: u32 = 4;
    const Y: u32 = 5;
    fn bar() -> u32 {
        unsafe {
            X = 3;
        }
        Y
    }
}
```

Error code E0435

A non-constant value was used in a constant expression.

Erroneous code example:

```
let foo = 42;
let a: [u8; foo]; // error: attempt to use a non-constant value in a
constant
```

'constant' means 'a compile-time value'.

More details can be found in the Variables and Mutability section of the book.

To fix this error, please replace the value with a constant. Example:

```
let a: [u8; 42]; // ok!
```

Or:

```
const FOO: usize = 42;
let a: [u8; FOO]; // ok!
```

Error code E0436

The functional record update syntax was used on something other than a struct.

Erroneous code example:

```
enum PublicationFrequency {
    Weekly,
    SemiMonthly { days: (u8, u8), annual_special: bool },
}

fn one_up_competitor(competitor_frequency: PublicationFrequency)
                     -> PublicationFrequency {
    match competitor_frequency {
        PublicationFrequency::Weekly => PublicationFrequency::SemiMonthly {
            days: (1, 15), annual_special: false
        },
        c @ PublicationFrequency::SemiMonthly{ .. } =>
            PublicationFrequency::SemiMonthly {
                annual_special: true, ..c // error: functional record update
                                          //        syntax requires a struct
        }
    }
}
```

The functional record update syntax is only allowed for structs (struct-like enum variants don't qualify, for example). To fix the previous code, rewrite the expression without functional record update syntax:

```
enum PublicationFrequency {
    Weekly,
    SemiMonthly { days: (u8, u8), annual_special: bool },
}

fn one_up_competitor(competitor_frequency: PublicationFrequency)
                     -> PublicationFrequency {
    match competitor_frequency {
        PublicationFrequency::Weekly => PublicationFrequency::SemiMonthly {
            days: (1, 15), annual_special: false
        },
        PublicationFrequency::SemiMonthly{ days, .. } =>
            PublicationFrequency::SemiMonthly {
                days, annual_special: true // ok!
        }
    }
}
```

Error code E0437

An associated type whose name does not match any of the associated types in the trait was used when implementing the trait.

Erroneous code example:

```
trait Foo {}

impl Foo for i32 {
    type Bar = bool;
}
```

Trait implementations can only implement associated types that are members of the trait in question.

The solution to this problem is to remove the extraneous associated type:

```
trait Foo {}

impl Foo for i32 {}
```

Error code E0438

An associated constant whose name does not match any of the associated constants in the trait was used when implementing the trait.

Erroneous code example:

```
trait Foo {}

impl Foo for i32 {
    const BAR: bool = true;
}
```

Trait implementations can only implement associated constants that are members of the trait in question.

The solution to this problem is to remove the extraneous associated constant:

```
trait Foo {}

impl Foo for i32 {}
```

Error code E0439

Note: this error code is no longer emitted by the compiler.

The length of the platform-intrinsic function `simd_shuffle` wasn't specified.

Erroneous code example:

```
#![feature(platform_intrinsics)]

extern "platform-intrinsic" {
    fn simd_shuffle<A,B>(a: A, b: A, c: [u32; 8]) -> B;
    // error: invalid `simd_shuffle`, needs length: `simd_shuffle`
}
```

The `simd_shuffle` function needs the length of the array passed as last parameter in its name. Example:

```
#![feature(platform_intrinsics)]

extern "platform-intrinsic" {
    fn simd_shuffle8<A,B>(a: A, b: A, c: [u32; 8]) -> B;
}
```

Error code E0445

Note: this error code is no longer emitted by the compiler.

A private trait was used on a public type parameter bound.

Previously erroneous code examples:

```
trait Foo {
    fn dummy(&self) { }
}

pub trait Bar : Foo {} // error: private trait in public interface
pub struct Bar2<T: Foo>(pub T); // same error
pub fn foo<T: Foo> (t: T) {} // same error

fn main() {}
```

To solve this error, please ensure that the trait is also public. The trait can be made inaccessible if necessary by placing it into a private inner module, but it still has to be marked with `pub`. Example:

```
pub trait Foo { // we set the Foo trait public
    fn dummy(&self) { }
}

pub trait Bar : Foo {} // ok!
pub struct Bar2<T: Foo>(pub T); // ok!
pub fn foo<T: Foo> (t: T) {} // ok!

fn main() {}
```

Error code E0446

A private type or trait was used in a public associated type signature.

Erroneous code example:

```
struct Bar;

pub trait PubTr {
    type Alias;
}

impl PubTr for u8 {
    type Alias = Bar; // error private type in public interface
}

fn main() {}
```

There are two ways to solve this error. The first is to make the public type signature only public to a module that also has access to the private type. This is done by using pub(crate) or pub(in crate::my_mod::etc) Example:

```
struct Bar;

pub(crate) trait PubTr { // only public to crate root
    type Alias;
}

impl PubTr for u8 {
    type Alias = Bar;
}

fn main() {}
```

The other way to solve this error is to make the private type public. Example:

```
pub struct Bar; // we set the Bar trait public

pub trait PubTr {
    type Alias;
}

impl PubTr for u8 {
    type Alias = Bar;
}

fn main() {}
```

Error code E0447

Note: this error code is no longer emitted by the compiler.

The `pub` keyword was used inside a function.

Erroneous code example:

```
fn foo() {
    pub struct Bar; // error: visibility has no effect inside functions
}
```

Since we cannot access items defined inside a function, the visibility of its items does not impact outer code. So using the `pub` keyword in this context is invalid.

Error code E0448

Note: this error code is no longer emitted by the compiler.

The `pub` keyword was used inside a public enum.

Erroneous code example:

```
pub enum Foo {
    pub Bar, // error: unnecessary `pub` visibility
}
```

Since the enum is already public, adding `pub` on one its elements is unnecessary. Example:

```
enum Foo {
    pub Bar, // not ok!
}
```

This is the correct syntax:

```
pub enum Foo {
    Bar, // ok!
}
```

Error code E0449

A visibility qualifier was used where one is not permitted. Visibility qualifiers are not permitted on enum variants, trait items, impl blocks, and extern blocks, as they already share the visibility of the parent item.

Erroneous code examples:

```
struct Bar;

trait Foo {
    fn foo();
}

enum Baz {
    pub Qux, // error: visibility qualifiers are not permitted here
}

pub impl Bar {} // error: visibility qualifiers are not permitted here

pub impl Foo for Bar { // error: visibility qualifiers are not permitted
here
    pub fn foo() {} // error: visibility qualifiers are not permitted here
}
```

To fix this error, simply remove the visibility qualifier. Example:

```
struct Bar;

trait Foo {
    fn foo();
}

enum Baz {
    // Enum variants share the visibility of the enum they are in, so
    // `pub` is not allowed here
    Qux,
}

// Directly implemented methods share the visibility of the type itself,
// so `pub` is not allowed here
impl Bar {}

// Trait methods share the visibility of the trait, so `pub` is not
// allowed in either case
impl Foo for Bar {
    fn foo() {}
}
```

Error code E0451

A struct constructor with private fields was invoked.

Erroneous code example:

```
mod bar {
    pub struct Foo {
        pub a: isize,
        b: isize,
    }
}

let f = bar::Foo{ a: 0, b: 0 }; // error: field `b` of struct `bar::Foo`
                                //        is private
```

To fix this error, please ensure that all the fields of the struct are public, or implement a function for easy instantiation. Examples:

```
mod bar {
    pub struct Foo {
        pub a: isize,
        pub b: isize, // we set `b` field public
    }
}

let f = bar::Foo{ a: 0, b: 0 }; // ok!
```

Or:

```
mod bar {
    pub struct Foo {
        pub a: isize,
        b: isize, // still private
    }

    impl Foo {
        pub fn new() -> Foo { // we create a method to instantiate `Foo`
            Foo { a: 0, b: 0 }
        }
    }
}

let f = bar::Foo::new(); // ok!
```

Error code E0452

An invalid lint attribute has been given.

Erroneous code example:

```
#![allow(foo = "")] // error: malformed lint attribute
```

Lint attributes only accept a list of identifiers (where each identifier is a lint name). Ensure the attribute is of this form:

```
#![allow(foo)] // ok!
// or:
#![allow(foo, foo2)] // ok!
```

Error code E0453

A lint check attribute was overruled by a `forbid` directive set as an attribute on an enclosing scope, or on the command line with the `-F` option.

Example of erroneous code:

```
#![forbid(non_snake_case)]

#[allow(non_snake_case)]
fn main() {
    let MyNumber = 2; // error: allow(non_snake_case) overruled by outer
                      //        forbid(non_snake_case)
}
```

The `forbid` lint setting, like `deny`, turns the corresponding compiler warning into a hard error. Unlike `deny`, `forbid` prevents itself from being overridden by inner attributes.

If you're sure you want to override the lint check, you can change `forbid` to `deny` (or use `-D` instead of `-F` if the `forbid` setting was given as a command-line option) to allow the inner lint check attribute:

```
#![deny(non_snake_case)]

#[allow(non_snake_case)]
fn main() {
    let MyNumber = 2; // ok!
}
```

Otherwise, edit the code to pass the lint check, and remove the overruled attribute:

```
#![forbid(non_snake_case)]

fn main() {
    let my_number = 2;
}
```

Error code E0454

A link name was given with an empty name.

Erroneous code example:

```
#[link(name = "")] extern "C" {}
// error: `#[link(name = "")]` given with empty name
```

The rust compiler cannot link to an external library if you don't give it its name. Example:

```
#[link(name = "some_lib")] extern "C" {} // ok!
```

Error code E0455

Some linking kinds are target-specific and not supported on all platforms.

Linking with `kind=framework` is only supported when targeting macOS, as frameworks are specific to that operating system.

Similarly, `kind=raw-dylib` is only supported when targeting Windows-like platforms.

Erroneous code example:

```
#[link(name = "FooCoreServices", kind = "framework")] extern "C" {}
// OS used to compile is Linux for example
```

To solve this error you can use conditional compilation:

```
#[cfg_attr(target="macos", link(name = "FooCoreServices", kind =
"framework"))]
extern "C" {}
```

Learn more in the Conditional Compilation section of the Reference.

Error code E0457

Note: this error code is no longer emitted by the compiler`

Plugin `..` only found in rlib format, but must be available in dylib format.

Erroneous code example:

rlib-plugin.rs

```
#![crate_type = "rlib"]
#![feature(rustc_private)]

extern crate rustc_middle;
extern crate rustc_driver;

use rustc_driver::plugin::Registry;

#[no_mangle]
fn __rustc_plugin_registrar(_: &mut Registry) {}
```

main.rs

```
#![feature(plugin)]
#![plugin(rlib_plugin)] // error: plugin `rlib_plugin` only found in rlib
                        //        format, but must be available in dylib

fn main() {}
```

The compiler exposes a plugin interface to allow altering the compile process (adding lints, etc). Plugins must be defined in their own crates (similar to proc-macro isolation) and then compiled and linked to another crate. Plugin crates *must* be compiled to the dynamically-linked dylib format, and not the statically-linked rlib format. Learn more about different output types in this section of the Rust reference.

This error is easily fixed by recompiling the plugin crate in the dylib format.

Error code E0458

An unknown "kind" was specified for a link attribute.

Erroneous code example:

```
#[link(kind = "wonderful_unicorn")] extern "C" {}
// error: unknown kind: `wonderful_unicorn`
```

Please specify a valid "kind" value, from one of the following:

- static
- dylib
- framework
- raw-dylib

Error code E0459

A link was used without a name parameter.

Erroneous code example:

```
#[link(kind = "dylib")] extern "C" {}
// error: `#[link(...)]` specified without `name = "foo"`
```

Please add the name parameter to allow the rust compiler to find the library you want. Example:

```
#[link(kind = "dylib", name = "some_lib")] extern "C" {} // ok!
```

Error code E0460

Found possibly newer version of crate `..` which `..` depends on.

Consider these erroneous files:

`a1.rs`

```
#![crate_name = "a"]

pub fn foo<T>() {}
```

`a2.rs`

```
#![crate_name = "a"]

pub fn foo<T>() {
    println!("foo<T>()");
}
```

`b.rs`

```
#![crate_name = "b"]

extern crate a; // linked with `a1.rs`

pub fn foo() {
    a::foo::<isize>();
}
```

`main.rs`

```
extern crate a; // linked with `a2.rs`
extern crate b; // error: found possibly newer version of crate `a` which `b`
                //        depends on

fn main() {}
```

The dependency graph of this program can be represented as follows:

```
    crate `main`
         |
         +-------------+
         |             |
         |             v
depends: |         crate `b`
 `a` v1  |             |
         |             | depends:
         |             |  `a` v2
         v             |
      crate `a` <------+
```

Crate `main` depends on crate `a` (version 1) and crate `b` which in turn depends on crate `a` (version 2); this discrepancy in versions cannot be reconciled. This difference in versions typically occurs when one crate is compiled and linked, then updated and linked to another crate. The crate "version" is a SVH (Strict Version Hash) of the crate in an implementation-specific way. Note that this error can *only* occur when directly compiling and linking with `rustc`; Cargo automatically resolves dependencies, without using the compiler's own dependency management that causes this issue.

This error can be fixed by:

- Using Cargo, the Rust package manager, automatically fixing this issue.
- Recompiling crate `a` so that both crate `b` and `main` have a uniform version to depend on.

Error code E0461

Couldn't find crate `..` with expected target triple `...`.

Example of erroneous code:

`a.rs`

```
#![crate_type = "lib"]

fn foo() {}
```

`main.rs`

```
extern crate a;

fn main() {
    a::foo();
}
```

`a.rs` is then compiled with `--target powerpc-unknown-linux-gnu` and `b.rs` with `--target x86_64-unknown-linux-gnu`. `a.rs` is compiled into a binary format incompatible with `b.rs`; PowerPC and x86 are totally different architectures. This issue also extends to any difference in target triples, as `std` is operating-system specific.

This error can be fixed by:

- Using Cargo, the Rust package manager, automatically fixing this issue.
- Recompiling either crate so that they target a consistent target triple.

Error code E0462

Found `staticlib ..` instead of `rlib` or `dylib`.

Consider the following two files:

`a.rs`

```
#![crate_type = "staticlib"]

fn foo() {}
```

`main.rs`

```
extern crate a;

fn main() {
    a::foo();
}
```

Crate `a` is compiled as a `staticlib`. A `staticlib` is a system-dependant library only intended for linking with non-Rust applications (C programs). Note that `staticlib`s include all upstream dependencies (`core`, `std`, other user dependencies, etc) which makes them significantly larger than `dylib`s: prefer `staticlib` for linking with C programs. Learn more about different `crate_type`s in this section of the Reference.

This error can be fixed by:

- Using Cargo, the Rust package manager, automatically fixing this issue.
- Recompiling the crate as a `rlib` or `dylib`; formats suitable for Rust linking.

Error code E0463

A crate was declared but cannot be found.

Erroneous code example:

```
extern crate foo; // error: can't find crate
```

You need to link your code to the relevant crate in order to be able to use it (through Cargo or the `-L` option of rustc, for example).

Common causes

- The crate is not present at all. If using Cargo, add it to `[dependencies]` in Cargo.toml.
- The crate is present, but under a different name. If using Cargo, look for `package =` under `[dependencies]` in Cargo.toml.

Common causes for missing `std` or `core`

- You are cross-compiling for a target which doesn't have `std` prepackaged. Consider one of the following:
  - Adding a pre-compiled version of std with `rustup target add`
  - Building std from source with `cargo build -Z build-std`
  - Using `#![no_std]` at the crate root, so you won't need `std` in the first place.
- You are developing the compiler itself and haven't built libstd from source. You can usually build it with `x.py build library/std`. More information about x.py is available in the rustc-dev-guide.

Error code E0464

The compiler found multiple library files with the requested crate name.

```
// aux-build:crateresolve-1.rs
// aux-build:crateresolve-2.rs
// aux-build:crateresolve-3.rs

extern crate crateresolve;
//~^ ERROR multiple candidates for `rlib` dependency `crateresolve` found

fn main() {}
```

This error can occur in several different cases -- for example, when using `extern crate` or passing `--extern` options without crate paths. It can also be caused by caching issues with the build directory, in which case `cargo clean` may help.

In the above example, there are three different library files, all of which define the same crate name. Without providing a full path, there is no way for the compiler to know which crate it should use.

Error code E0466

Macro import declaration was malformed.

Erroneous code examples:

```
#[macro_use(a_macro(another_macro))] // error: invalid import declaration
extern crate core as some_crate;

#[macro_use(i_want = "some_macros")] // error: invalid import declaration
extern crate core as another_crate;
```

This is a syntax error at the level of attribute declarations. The proper syntax for macro imports is the following:

```
// In some_crate:
#[macro_export]
macro_rules! get_tacos {
    ...
}

#[macro_export]
macro_rules! get_pimientos {
    ...
}

// In your crate:
#[macro_use(get_tacos, get_pimientos)] // It imports `get_tacos` and
extern crate some_crate;               // `get_pimientos` macros from
some_crate
```

If you would like to import all exported macros, write `macro_use` with no arguments.

Error code E0468

A non-root module tried to import macros from another crate.

Example of erroneous code:

```
mod foo {
    #[macro_use(debug_assert)]  // error: must be at crate root to import
    extern crate core;          //        macros from another crate
    fn run_macro() { debug_assert!(true); }
}
```

Only `extern crate` imports at the crate root level are allowed to import macros.

Either move the macro import to crate root or do without the foreign macros. This will work:

```
#[macro_use(debug_assert)] // ok!
extern crate core;

mod foo {
    fn run_macro() { debug_assert!(true); }
}
```

Error code E0469

A macro listed for import was not found.

Erroneous code example:

```
#[macro_use(drink, be_merry)] // error: imported macro not found
extern crate alloc;

fn main() {
    // ...
}
```

Either the listed macro is not contained in the imported crate, or it is not exported from the given crate.

This could be caused by a typo. Did you misspell the macro's name?

Double-check the names of the macros listed for import, and that the crate in question exports them.

A working version would be:

```
// In some_crate crate:
#[macro_export]
macro_rules! eat {
    ...
}

#[macro_export]
macro_rules! drink {
    ...
}

// In your crate:
#[macro_use(eat, drink)]
extern crate some_crate; //ok!
```

Error code E0472

Inline assembly (`asm!`) is not supported on this target.

Example of erroneous code:

```
// compile-flags: --target sparc64-unknown-linux-gnu
#![no_std]

use core::arch::asm;

fn main() {
    unsafe {
        asm!(""); // error: inline assembly is not supported on this target
    }
}
```

The Rust compiler does not support inline assembly, with the `asm!` macro (previously `llvm_asm!`), for all targets. All Tier 1 targets do support this macro but support among Tier 2 and 3 targets is not guaranteed (even when they have `std` support). Note that this error is related to `error[E0658]: inline assembly is not stable yet on this architecture` , but distinct in that with `E0472` support is not planned or in progress.

There is no way to easily fix this issue, however:

- Consider if you really need inline assembly, is there some other way to achieve your goal (intrinsics, etc)?
- Consider writing your assembly externally, linking with it and calling it from Rust.
- Consider contributing to https://github.com/rust-lang/rust and help integrate support for your target!

Error code E0476

The coerced type does not outlive the value being coerced to.

Example of erroneous code:

```
#![feature(coerce_unsized)]
#![feature(unsize)]

use std::marker::Unsize;
use std::ops::CoerceUnsized;

// error: lifetime of the source pointer does not outlive lifetime bound of
the
//        object type
impl<'a, 'b, T, S> CoerceUnsized<&'a T> for &'b S where S: Unsize<T> {}
```

During a coercion, the "source pointer" (the coerced type) did not outlive the "object type" (value being coerced to). In the above example, `'b` is not a subtype of `'a`. This error can currently only be encountered with the unstable `CoerceUnsized` trait which allows custom coercions of unsized types behind a smart pointer to be implemented.

Error code E0477

Note: this error code is no longer emitted by the compiler.

The type does not fulfill the required lifetime.

Erroneous code example:

```
use std::sync::Mutex;

struct MyString<'a> {
    data: &'a str,
}

fn i_want_static_closure<F>(a: F)
    where F: Fn() + 'static {}

fn print_string<'a>(s: Mutex<MyString<'a>>) {

    i_want_static_closure(move || {     // error: this closure has lifetime
'a
                                        //        rather than 'static
        println!("{}", s.lock().unwrap().data);
    });
}
```

In this example, the closure does not satisfy the `'static` lifetime constraint. To fix this error, you need to double check the lifetime of the type. Here, we can fix this problem by giving `s` a static lifetime:

```
use std::sync::Mutex;

struct MyString<'a> {
    data: &'a str,
}

fn i_want_static_closure<F>(a: F)
    where F: Fn() + 'static {}

fn print_string(s: Mutex<MyString<'static>>) {

    i_want_static_closure(move || {     // ok!
        println!("{}", s.lock().unwrap().data);
    });
}
```

Error code E0478

A lifetime bound was not satisfied.

Erroneous code example:

```
// Check that the explicit lifetime bound (`'SnowWhite`, in this example)
must
// outlive all the superbounds from the trait (`'kiss`, in this example).

trait Wedding<'t>: 't { }

struct Prince<'kiss, 'SnowWhite> {
    child: Box<Wedding<'kiss> + 'SnowWhite>,
    // error: lifetime bound not satisfied
}
```

In this example, the `'SnowWhite` lifetime is supposed to outlive the `'kiss` lifetime but the declaration of the `Prince` struct doesn't enforce it. To fix this issue, you need to specify it:

```
trait Wedding<'t>: 't { }

struct Prince<'kiss, 'SnowWhite: 'kiss> { // You say here that 'SnowWhite
                                          // must live longer than 'kiss.
    child: Box<Wedding<'kiss> + 'SnowWhite>, // And now it's all good!
}
```

Error code E0482

Note: this error code is no longer emitted by the compiler.

A lifetime of a returned value does not outlive the function call.

Erroneous code example:

```
fn prefix<'a>(
    words: impl Iterator<Item = &'a str>
) -> impl Iterator<Item = String> { // error!
    words.map(|v| format!("foo-{}", v))
}
```

To fix this error, make the lifetime of the returned value explicit:

```
fn prefix<'a>(
    words: impl Iterator<Item = &'a str> + 'a
) -> impl Iterator<Item = String> + 'a { // ok!
    words.map(|v| format!("foo-{}", v))
}
```

The `impl Trait` feature in this example uses an implicit `'static` lifetime restriction in the returned type. However the type implementing the `Iterator` passed to the function lives just as long as `'a`, which is not long enough.

The solution involves adding lifetime bound to both function argument and the return value to make sure that the values inside the iterator are not dropped when the function goes out of the scope.

An alternative solution would be to guarantee that the `Item` references in the iterator are alive for the whole lifetime of the program.

```
fn prefix(
    words: impl Iterator<Item = &'static str>
) -> impl Iterator<Item = String> {  // ok!
    words.map(|v| format!("foo-{}", v))
}
```

A similar lifetime problem might arise when returning closures:

```
fn foo(
    x: &mut Vec<i32>
) -> impl FnMut(&mut Vec<i32>) -> &[i32] { // error!
    |y| {
        y.append(x);
        y
    }
}
```

Analogically, a solution here is to use explicit return lifetime and move the ownership of the variable to the closure.

```
fn foo<'a>(
    x: &'a mut Vec<i32>
) -> impl FnMut(&mut Vec<i32>) -> &[i32] + 'a { // ok!
    move |y| {
        y.append(x);
        y
    }
}
```

To better understand the lifetime treatment in the `impl Trait`, please see the RFC 1951.

Error code E0491

A reference has a longer lifetime than the data it references.

Erroneous code example:

```
struct Foo<'a> {
    x: fn(&'a i32),
}

trait Trait<'a, 'b> {
    type Out;
}

impl<'a, 'b> Trait<'a, 'b> for usize {
    type Out = &'a Foo<'b>; // error!
}
```

Here, the problem is that the compiler cannot be sure that the `'b` lifetime will live longer than `'a`, which should be mandatory in order to be sure that `Trait::Out` will always have a reference pointing to an existing type. So in this case, we just need to tell the compiler than `'b` must outlive `'a`:

```
struct Foo<'a> {
    x: fn(&'a i32),
}

trait Trait<'a, 'b> {
    type Out;
}

impl<'a, 'b: 'a> Trait<'a, 'b> for usize { // we added the lifetime
enforcement
    type Out = &'a Foo<'b>; // it now works!
}
```

Error code E0492

A borrow of a constant containing interior mutability was attempted.

Erroneous code example:

```
use std::sync::atomic::AtomicUsize;

const A: AtomicUsize = AtomicUsize::new(0);
const B: &'static AtomicUsize = &A;
// error: cannot borrow a constant which may contain interior mutability,
//        create a static instead
```

A `const` represents a constant value that should never change. If one takes a `&` reference to the constant, then one is taking a pointer to some memory location containing the value. Normally this is perfectly fine: most values can't be changed via a shared `&` pointer, but interior mutability would allow it. That is, a constant value could be mutated. On the other hand, a `static` is explicitly a single memory location, which can be mutated at will.

So, in order to solve this error, use statics which are `Sync`:

```
use std::sync::atomic::AtomicUsize;

static A: AtomicUsize = AtomicUsize::new(0);
static B: &'static AtomicUsize = &A; // ok!
```

You can also have this error while using a cell type:

```
use std::cell::Cell;

const A: Cell<usize> = Cell::new(1);
const B: &Cell<usize> = &A;
// error: cannot borrow a constant which may contain interior mutability,
//        create a static instead

// or:
struct C { a: Cell<usize> }

const D: C = C { a: Cell::new(1) };
const E: &Cell<usize> = &D.a; // error

// or:
const F: &C = &D; // error
```

This is because cell types do operations that are not thread-safe. Due to this, they don't implement Sync and thus can't be placed in statics.

However, if you still wish to use these types, you can achieve this by an unsafe wrapper:

```
use std::cell::Cell;

struct NotThreadSafe<T> {
    value: Cell<T>,
}

unsafe impl<T> Sync for NotThreadSafe<T> {}

static A: NotThreadSafe<usize> = NotThreadSafe { value : Cell::new(1) };
static B: &'static NotThreadSafe<usize> = &A; // ok!
```

Remember this solution is unsafe! You will have to ensure that accesses to the cell are synchronized.

Error code E0493

A value with a custom `Drop` implementation may be dropped during const-eval.

Erroneous code example:

```
enum DropType {
    A,
}

impl Drop for DropType {
    fn drop(&mut self) {}
}

struct Foo {
    field1: DropType,
}

static FOO: Foo = Foo { field1: (DropType::A, DropType::A).1 }; // error!
```

The problem here is that if the given type or one of its fields implements the `Drop` trait, this `Drop` implementation cannot be called within a const context since it may run arbitrary, non-const-checked code. To prevent this issue, ensure all values with a custom `Drop` implementation escape the initializer.

```
enum DropType {
    A,
}

impl Drop for DropType {
    fn drop(&mut self) {}
}

struct Foo {
    field1: DropType,
}

static FOO: Foo = Foo { field1: DropType::A }; // We initialize all fields
                                               // by hand.
```

Error code E0495

Note: this error code is no longer emitted by the compiler.

A lifetime cannot be determined in the given situation.

Erroneous code example:

```
fn transmute_lifetime<'a, 'b, T>(t: &'a (T,)) -> &'b T {
    match (&t,) { // error!
        ((u,),) => u,
    }
}

let y = Box::new((42,));
let x = transmute_lifetime(&y);
```

In this code, you have two ways to solve this issue:

1. Enforce that `'a` lives at least as long as `'b`.
2. Use the same lifetime requirement for both input and output values.

So for the first solution, you can do it by replacing `'a` with `'a: 'b`:

```
fn transmute_lifetime<'a: 'b, 'b, T>(t: &'a (T,)) -> &'b T {
    match (&t,) { // ok!
        ((u,),) => u,
    }
}
```

In the second you can do it by simply removing `'b` so they both use `'a`:

```
fn transmute_lifetime<'a, T>(t: &'a (T,)) -> &'a T {
    match (&t,) { // ok!
        ((u,),) => u,
    }
}
```

Error code E0496

A lifetime name is shadowing another lifetime name.

Erroneous code example:

```
struct Foo<'a> {
    a: &'a i32,
}

impl<'a> Foo<'a> {
    fn f<'a>(x: &'a i32) { // error: lifetime name `'a` shadows a lifetime
                           //        name that is already in scope
    }
}
```

Please change the name of one of the lifetimes to remove this error. Example:

```
struct Foo<'a> {
    a: &'a i32,
}

impl<'a> Foo<'a> {
    fn f<'b>(x: &'b i32) { // ok!
    }
}

fn main() {
}
```

Error code E0497

Note: this error code is no longer emitted by the compiler.

A stability attribute was used outside of the standard library.

Erroneous code example:

```
#[stable] // error: stability attributes may not be used outside of the
          //        standard library
fn foo() {}
```

It is not possible to use stability attributes outside of the standard library. Also, for now, it is not possible to write deprecation messages either.

Error code E0498

Note: this error code is no longer emitted by the compiler.

The `plugin` attribute was malformed.

Erroneous code example:

```
#![feature(plugin)]
#![plugin(foo(args))] // error: invalid argument
#![plugin(bar="test")] // error: invalid argument
```

The `#[plugin]` attribute should take a single argument: the name of the plugin.

For example, for the plugin `foo` :

```
#![feature(plugin)]
#![plugin(foo)] // ok!
```

See the `plugin` feature section of the Unstable book for more details.

Error code E0499

A variable was borrowed as mutable more than once.

Erroneous code example:

```rust
let mut i = 0;
let mut x = &mut i;
let mut a = &mut i;
x;
// error: cannot borrow `i` as mutable more than once at a time
```

Please note that in Rust, you can either have many immutable references, or one mutable reference. For more details you may want to read the References & Borrowing section of the Book.

Example:

```rust
let mut i = 0;
let mut x = &mut i; // ok!

// or:
let mut i = 0;
let a = &i; // ok!
let b = &i; // still ok!
let c = &i; // still ok!
b;
a;
```

Error code E0500

A borrowed variable was used by a closure.

Erroneous code example:

```
fn you_know_nothing(jon_snow: &mut i32) {
    let nights_watch = &jon_snow;
    let starks = || {
        *jon_snow = 3; // error: closure requires unique access to
`jon_snow`
                       //        but it is already borrowed
    };
    println!("{}", nights_watch);
}
```

In here, `jon_snow` is already borrowed by the `nights_watch` reference, so it cannot be borrowed by the `starks` closure at the same time. To fix this issue, you can create the closure after the borrow has ended:

```
fn you_know_nothing(jon_snow: &mut i32) {
    let nights_watch = &jon_snow;
    println!("{}", nights_watch);
    let starks = || {
        *jon_snow = 3;
    };
}
```

Or, if the type implements the `Clone` trait, you can clone it between closures:

```
fn you_know_nothing(jon_snow: &mut i32) {
    let mut jon_copy = jon_snow.clone();
    let starks = || {
        *jon_snow = 3;
    };
    println!("{}", jon_copy);
}
```

Error code E0501

A mutable variable is used but it is already captured by a closure.

Erroneous code example:

```
fn inside_closure(x: &mut i32) {
    // Actions which require unique access
}

fn outside_closure(x: &mut i32) {
    // Actions which require unique access
}

fn foo(a: &mut i32) {
    let mut bar = || {
        inside_closure(a)
    };
    outside_closure(a); // error: cannot borrow `*a` as mutable because
previous
                        //        closure requires unique access.
    bar();
}
```

This error indicates that a mutable variable is used while it is still captured by a closure. Because the closure has borrowed the variable, it is not available until the closure goes out of scope.

Note that a capture will either move or borrow a variable, but in this situation, the closure is borrowing the variable. Take a look at the chapter on Capturing in Rust By Example for more information.

To fix this error, you can finish using the closure before using the captured variable:

```
fn inside_closure(x: &mut i32) {}
fn outside_closure(x: &mut i32) {}

fn foo(a: &mut i32) {
    let mut bar = || {
        inside_closure(a)
    };
    bar();
    // borrow on `a` ends.
    outside_closure(a); // ok!
}
```

Or you can pass the variable as a parameter to the closure:

```
fn inside_closure(x: &mut i32) {}
fn outside_closure(x: &mut i32) {}

fn foo(a: &mut i32) {
    let mut bar = |s: &mut i32| {
        inside_closure(s)
    };
    outside_closure(a);
    bar(a);
}
```

It may be possible to define the closure later:

```
fn inside_closure(x: &mut i32) {}
fn outside_closure(x: &mut i32) {}

fn foo(a: &mut i32) {
    outside_closure(a);
    let mut bar = || {
        inside_closure(a)
    };
    bar();
}
```

Error code E0502

A variable already borrowed as immutable was borrowed as mutable.

Erroneous code example:

```
fn bar(x: &mut i32) {}
fn foo(a: &mut i32) {
    let y = &a; // a is borrowed as immutable.
    bar(a); // error: cannot borrow `*a` as mutable because `a` is also
borrowed
            //        as immutable
    println!("{}", y);
}
```

To fix this error, ensure that you don't have any other references to the variable before trying to access it mutably:

```
fn bar(x: &mut i32) {}
fn foo(a: &mut i32) {
    bar(a);
    let y = &a; // ok!
    println!("{}", y);
}
```

For more information on Rust's ownership system, take a look at the References & Borrowing section of the Book.

Error code E0503

A value was used after it was mutably borrowed.

Erroneous code example:

```
fn main() {
    let mut value = 3;
    // Create a mutable borrow of `value`.
    let borrow = &mut value;
    let _sum = value + 1; // error: cannot use `value` because
                          //        it was mutably borrowed
    println!("{}", borrow);
}
```

In this example, `value` is mutably borrowed by `borrow` and cannot be used to calculate `sum`. This is not possible because this would violate Rust's mutability rules.

You can fix this error by finishing using the borrow before the next use of the value:

```
fn main() {
    let mut value = 3;
    let borrow = &mut value;
    println!("{}", borrow);
    // The block has ended and with it the borrow.
    // You can now use `value` again.
    let _sum = value + 1;
}
```

Or by cloning `value` before borrowing it:

```
fn main() {
    let mut value = 3;
    // We clone `value`, creating a copy.
    let value_cloned = value.clone();
    // The mutable borrow is a reference to `value` and
    // not to `value_cloned`...
    let borrow = &mut value;
    // ... which means we can still use `value_cloned`,
    let _sum = value_cloned + 1;
    // even though the borrow only ends here.
    println!("{}", borrow);
}
```

For more information on Rust's ownership system, take a look at the References & Borrowing section of the Book.

Error code E0504

Note: this error code is no longer emitted by the compiler.

This error occurs when an attempt is made to move a borrowed variable into a closure.

Erroneous code example:

```
struct FancyNum {
    num: u8,
}

fn main() {
    let fancy_num = FancyNum { num: 5 };
    let fancy_ref = &fancy_num;

    let x = move || {
        println!("child function: {}", fancy_num.num);
        // error: cannot move `fancy_num` into closure because it is
borrowed
    };

    x();
    println!("main function: {}", fancy_ref.num);
}
```

Here, `fancy_num` is borrowed by `fancy_ref` and so cannot be moved into the closure `x`. There is no way to move a value into a closure while it is borrowed, as that would invalidate the borrow.

If the closure can't outlive the value being moved, try using a reference rather than moving:

```
struct FancyNum {
    num: u8,
}

fn main() {
    let fancy_num = FancyNum { num: 5 };
    let fancy_ref = &fancy_num;

    let x = move || {
        // fancy_ref is usable here because it doesn't move `fancy_num`
        println!("child function: {}", fancy_ref.num);
    };

    x();

    println!("main function: {}", fancy_num.num);
}
```

If the value has to be borrowed and then moved, try limiting the lifetime of the borrow using a scoped block:

```
struct FancyNum {
    num: u8,
}

fn main() {
    let fancy_num = FancyNum { num: 5 };

    {
        let fancy_ref = &fancy_num;
        println!("main function: {}", fancy_ref.num);
        // `fancy_ref` goes out of scope here
    }

    let x = move || {
        // `fancy_num` can be moved now (no more references exist)
        println!("child function: {}", fancy_num.num);
    };

    x();
}
```

If the lifetime of a reference isn't enough, such as in the case of threading, consider using an `Arc` to create a reference-counted value:

```
use std::sync::Arc;
use std::thread;

struct FancyNum {
    num: u8,
}

fn main() {
    let fancy_ref1 = Arc::new(FancyNum { num: 5 });
    let fancy_ref2 = fancy_ref1.clone();

    let x = thread::spawn(move || {
        // `fancy_ref1` can be moved and has a `'static` lifetime
        println!("child thread: {}", fancy_ref1.num);
    });

    x.join().expect("child thread should finish");
    println!("main thread: {}", fancy_ref2.num);
}
```

Error code E0505

A value was moved out while it was still borrowed.

Erroneous code example:

```rust
struct Value {}

fn borrow(val: &Value) {}

fn eat(val: Value) {}

fn main() {
    let x = Value{};
    let _ref_to_val: &Value = &x;
    eat(x);
    borrow(_ref_to_val);
}
```

Here, the function `eat` takes ownership of `x`. However, `x` cannot be moved because the borrow to `_ref_to_val` needs to last till the function `borrow`. To fix that you can do a few different things:

- Try to avoid moving the variable.
- Release borrow before move.
- Implement the `Copy` trait on the type.

Examples:

```rust
struct Value {}

fn borrow(val: &Value) {}

fn eat(val: &Value) {}

fn main() {
    let x = Value{};

    let ref_to_val: &Value = &x;
    eat(&x); // pass by reference, if it's possible
    borrow(ref_to_val);
}
```

Or:

```rust
struct Value {}

fn borrow(val: &Value) {}

fn eat(val: Value) {}

fn main() {
    let x = Value{};

    let ref_to_val: &Value = &x;
    borrow(ref_to_val);
    // ref_to_val is no longer used.
    eat(x);
}
```

Or:

```rust
#[derive(Clone, Copy)] // implement Copy trait
struct Value {}

fn borrow(val: &Value) {}

fn eat(val: Value) {}

fn main() {
    let x = Value{};
    let ref_to_val: &Value = &x;
    eat(x); // it will be copied here.
    borrow(ref_to_val);
}
```

For more information on Rust's ownership system, take a look at the References & Borrowing section of the Book.

Error code E0506

An attempt was made to assign to a borrowed value.

Erroneous code example:

```
struct FancyNum {
    num: u8,
}

let mut fancy_num = FancyNum { num: 5 };
let fancy_ref = &fancy_num;
fancy_num = FancyNum { num: 6 };
// error: cannot assign to `fancy_num` because it is borrowed

println!("Num: {}, Ref: {}", fancy_num.num, fancy_ref.num);
```

Because `fancy_ref` still holds a reference to `fancy_num`, `fancy_num` can't be assigned to a new value as it would invalidate the reference.

Alternatively, we can move out of `fancy_num` into a second `fancy_num`:

```
struct FancyNum {
    num: u8,
}

let mut fancy_num = FancyNum { num: 5 };
let moved_num = fancy_num;
fancy_num = FancyNum { num: 6 };

println!("Num: {}, Moved num: {}", fancy_num.num, moved_num.num);
```

If the value has to be borrowed, try limiting the lifetime of the borrow using a scoped block:

```
struct FancyNum {
    num: u8,
}

let mut fancy_num = FancyNum { num: 5 };

{
    let fancy_ref = &fancy_num;
    println!("Ref: {}", fancy_ref.num);
}

// Works because `fancy_ref` is no longer in scope
fancy_num = FancyNum { num: 6 };
println!("Num: {}", fancy_num.num);
```

Or by moving the reference into a function:

```
struct FancyNum {
    num: u8,
}

fn print_fancy_ref(fancy_ref: &FancyNum){
    println!("Ref: {}", fancy_ref.num);
}

let mut fancy_num = FancyNum { num: 5 };

print_fancy_ref(&fancy_num);

// Works because function borrow has ended
fancy_num = FancyNum { num: 6 };
println!("Num: {}", fancy_num.num);
```

Error code E0507

A borrowed value was moved out.

Erroneous code example:

```
use std::cell::RefCell;

struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

fn main() {
    let x = RefCell::new(TheDarkKnight);

    x.borrow().nothing_is_true(); // error: cannot move out of borrowed
content
}
```

Here, the `nothing_is_true` method takes the ownership of `self`. However, `self` cannot be moved because `.borrow()` only provides an `&TheDarkKnight`, which is a borrow of the content owned by the `RefCell`. To fix this error, you have three choices:

- Try to avoid moving the variable.
- Somehow reclaim the ownership.
- Implement the `Copy` trait on the type.

This can also happen when using a type implementing `Fn` or `FnMut`, as neither allows moving out of them (they usually represent closures which can be called more than once). Much of the text following applies equally well to non- `FnOnce` closure bodies.

Examples:

```
use std::cell::RefCell;

struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(&self) {} // First case, we don't take ownership
}

fn main() {
    let x = RefCell::new(TheDarkKnight);

    x.borrow().nothing_is_true(); // ok!
}
```

Or:

```
use std::cell::RefCell;

struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

fn main() {
    let x = RefCell::new(TheDarkKnight);
    let x = x.into_inner(); // we get back ownership

    x.nothing_is_true(); // ok!
}
```

Or:

```
use std::cell::RefCell;

#[derive(Clone, Copy)] // we implement the Copy trait
struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

fn main() {
    let x = RefCell::new(TheDarkKnight);

    x.borrow().nothing_is_true(); // ok!
}
```

Moving a member out of a mutably borrowed struct will also cause E0507 error:

```
struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

struct Batcave {
    knight: TheDarkKnight
}

fn main() {
    let mut cave = Batcave {
        knight: TheDarkKnight
    };
    let borrowed = &mut cave;

    borrowed.knight.nothing_is_true(); // E0507
}
```

It is fine only if you put something back. `mem::replace` can be used for that:

```
use std::mem;

let mut cave = Batcave {
    knight: TheDarkKnight
};
let borrowed = &mut cave;

mem::replace(&mut borrowed.knight, TheDarkKnight).nothing_is_true(); // ok!
```

For more information on Rust's ownership system, take a look at the References & Borrowing section of the Book.

Error code E0508

A value was moved out of a non-copy fixed-size array.

Erroneous code example:

```
struct NonCopy;

fn main() {
    let array = [NonCopy; 1];
    let _value = array[0]; // error: cannot move out of type `[NonCopy; 1]`,
                           //        a non-copy fixed-size array
}
```

The first element was moved out of the array, but this is not possible because `NonCopy` does not implement the `Copy` trait.

Consider borrowing the element instead of moving it:

```
struct NonCopy;

fn main() {
    let array = [NonCopy; 1];
    let _value = &array[0]; // Borrowing is allowed, unlike moving.
}
```

Alternatively, if your type implements `Clone` and you need to own the value, consider borrowing and then cloning:

```
#[derive(Clone)]
struct NonCopy;

fn main() {
    let array = [NonCopy; 1];
    // Now you can clone the array element.
    let _value = array[0].clone();
}
```

If you really want to move the value out, you can use a destructuring array pattern to move it:

```
struct NonCopy;

fn main() {
    let array = [NonCopy; 1];
    // Destructuring the array
    let [_value] = array;
}
```

Error code E0509

This error occurs when an attempt is made to move out of a value whose type implements the `Drop` trait.

Erroneous code example:

```
struct FancyNum {
    num: usize
}

struct DropStruct {
    fancy: FancyNum
}

impl Drop for DropStruct {
    fn drop(&mut self) {
        // Destruct DropStruct, possibly using FancyNum
    }
}

fn main() {
    let drop_struct = DropStruct{fancy: FancyNum{num: 5}};
    let fancy_field = drop_struct.fancy; // Error E0509
    println!("Fancy: {}", fancy_field.num);
    // implicit call to `drop_struct.drop()` as drop_struct goes out of
scope
}
```

Here, we tried to move a field out of a struct of type `DropStruct` which implements the `Drop` trait. However, a struct cannot be dropped if one or more of its fields have been moved.

Structs implementing the `Drop` trait have an implicit destructor that gets called when they go out of scope. This destructor may use the fields of the struct, so moving out of the struct could make it impossible to run the destructor. Therefore, we must think of all values whose type implements the `Drop` trait as single units whose fields cannot be moved.

This error can be fixed by creating a reference to the fields of a struct, enum, or tuple using the `ref` keyword:

```
struct FancyNum {
    num: usize
}

struct DropStruct {
    fancy: FancyNum
}

impl Drop for DropStruct {
    fn drop(&mut self) {
        // Destruct DropStruct, possibly using FancyNum
    }
}

fn main() {
    let drop_struct = DropStruct{fancy: FancyNum{num: 5}};
    let ref fancy_field = drop_struct.fancy; // No more errors!
    println!("Fancy: {}", fancy_field.num);
    // implicit call to `drop_struct.drop()` as drop_struct goes out of scope
}
```

Note that this technique can also be used in the arms of a match expression:

```
struct FancyNum {
    num: usize
}

enum DropEnum {
    Fancy(FancyNum)
}

impl Drop for DropEnum {
    fn drop(&mut self) {
        // Destruct DropEnum, possibly using FancyNum
    }
}

fn main() {
    // Creates and enum of type `DropEnum`, which implements `Drop`
    let drop_enum = DropEnum::Fancy(FancyNum{num: 10});
    match drop_enum {
        // Creates a reference to the inside of `DropEnum::Fancy`
        DropEnum::Fancy(ref fancy_field) => // No error!
            println!("It was fancy-- {}!", fancy_field.num),
    }
    // implicit call to `drop_enum.drop()` as drop_enum goes out of scope
}
```

Error code E0510

The matched value was assigned in a match guard.

Erroneous code example:

```
let mut x = Some(0);
match x {
    None => {}
    Some(_) if { x = None; false } => {} // error!
    Some(_) => {}
}
```

When matching on a variable it cannot be mutated in the match guards, as this could cause the match to be non-exhaustive.

Here executing `x = None` would modify the value being matched and require us to go "back in time" to the `None` arm. To fix it, change the value in the match arm:

```
let mut x = Some(0);
match x {
    None => {}
    Some(_) => {
        x = None; // ok!
    }
}
```

Error code E0511

Invalid monomorphization of an intrinsic function was used.

Erroneous code example:

```
#![feature(platform_intrinsics)]

extern "platform-intrinsic" {
    fn simd_add<T>(a: T, b: T) -> T;
}

fn main() {
    unsafe { simd_add(0, 1); }
    // error: invalid monomorphization of `simd_add` intrinsic
}
```

The generic type has to be a SIMD type. Example:

```
#![feature(repr_simd)]
#![feature(platform_intrinsics)]

#[repr(simd)]
#[derive(Copy, Clone)]
struct i32x2(i32, i32);

extern "platform-intrinsic" {
    fn simd_add<T>(a: T, b: T) -> T;
}

unsafe { simd_add(i32x2(0, 0), i32x2(1, 2)); } // ok!
```

Error code E0512

Transmute with two differently sized types was attempted.

Erroneous code example:

```
fn takes_u8(_: u8) {}

fn main() {
    unsafe { takes_u8(::std::mem::transmute(0u16)); }
    // error: cannot transmute between types of different sizes,
    //        or dependently-sized types
}
```

Please use types with same size or use the expected type directly. Example:

```
fn takes_u8(_: u8) {}

fn main() {
    unsafe { takes_u8(::std::mem::transmute(0i8)); } // ok!
    // or:
    unsafe { takes_u8(0u8); } // ok!
}
```

Error code E0514

Dependency compiled with different version of `rustc`.

Example of erroneous code:

`a.rs`

```
// compiled with stable `rustc`

#[crate_type = "lib"]
```

`b.rs`

```
// compiled with nightly `rustc`

#[crate_type = "lib"]

extern crate a; // error: found crate `a` compiled by an incompatible version
                //        of rustc
```

This error is caused when the version of `rustc` used to compile a crate, as stored in the binary's metadata, differs from the version of one of its dependencies. Many parts of Rust binaries are considered unstable. For instance, the Rust ABI is not stable between compiler versions. This means that the compiler cannot be sure about *how* to call a function between compiler versions, and therefore this error occurs.

This error can be fixed by:

- Using Cargo, the Rust package manager and Rustup, the Rust toolchain installer, automatically fixing this issue.
- Recompiling the crates with a uniform `rustc` version.

Error code E0515

A reference to a local variable was returned.

Erroneous code example:

```
fn get_dangling_reference() -> &'static i32 {
    let x = 0;
    &x
}
```

```
use std::slice::Iter;
fn get_dangling_iterator<'a>() -> Iter<'a, i32> {
    let v = vec![1, 2, 3];
    v.iter()
}
```

Local variables, function parameters and temporaries are all dropped before the end of the function body. So a reference to them cannot be returned.

Consider returning an owned value instead:

```
use std::vec::IntoIter;

fn get_integer() -> i32 {
    let x = 0;
    x
}

fn get_owned_iterator() -> IntoIter<i32> {
    let v = vec![1, 2, 3];
    v.into_iter()
}
```

Error code E0516

The `typeof` keyword is currently reserved but unimplemented.

Erroneous code example:

```
fn main() {
    let x: typeof(92) = 92;
}
```

Try using type inference instead. Example:

```
fn main() {
    let x = 92;
}
```

Error code E0517

A `#[repr(..)]` attribute was placed on an unsupported item.

Examples of erroneous code:

```
#[repr(C)]
type Foo = u8;

#[repr(packed)]
enum Foo {Bar, Baz}

#[repr(u8)]
struct Foo {bar: bool, baz: bool}

#[repr(C)]
impl Foo {
    // ...
}
```

- The `#[repr(C)]` attribute can only be placed on structs and enums.
- The `#[repr(packed)]` and `#[repr(simd)]` attributes only work on structs.
- The `#[repr(u8)]` , `#[repr(i16)]` , etc attributes only work on enums.

These attributes do not work on typedefs, since typedefs are just aliases.

Representations like `#[repr(u8)]` , `#[repr(i64)]` are for selecting the discriminant size for enums with no data fields on any of the variants, e.g. `enum Color {Red, Blue, Green}` , effectively setting the size of the enum to the size of the provided type. Such an enum can be cast to a value of the same type as well. In short, `#[repr(u8)]` makes the enum behave like an integer with a constrained set of allowed values.

Only field-less enums can be cast to numerical primitives, so this attribute will not apply to structs.

`#[repr(packed)]` reduces padding to make the struct size smaller. The representation of enums isn't strictly defined in Rust, and this attribute won't work on enums.

`#[repr(simd)]` will give a struct consisting of a homogeneous series of machine types (i.e., `u8` , `i32` , etc) a representation that permits vectorization via SIMD. This doesn't make much sense for enums since they don't consist of a single list of data.

Error code E0518

An `#[inline(..)]` attribute was incorrectly placed on something other than a function or method.

Example of erroneous code:

```
#[inline(always)]
struct Foo;

#[inline(never)]
impl Foo {
    // ...
}
```

`#[inline]` hints the compiler whether or not to attempt to inline a method or function. By default, the compiler does a pretty good job of figuring this out itself, but if you feel the need for annotations, `#[inline(always)]` and `#[inline(never)]` can override or force the compiler's decision.

If you wish to apply this attribute to all methods in an impl, manually annotate each method; it is not possible to annotate the entire impl with an `#[inline]` attribute.

Error code E0519

The current crate is indistinguishable from one of its dependencies, in terms of metadata.

Example of erroneous code:

a.rs

```
#![crate_name = "a"]
#![crate_type = "lib"]

pub fn foo() {}
```

b.rs

```
#![crate_name = "a"]
#![crate_type = "lib"]

// error: the current crate is indistinguishable from one of its
dependencies:
//        it has the same crate-name `a` and was compiled with the same
//        `-C metadata` arguments. This will result in symbol conflicts
between
//        the two.
extern crate a;

pub fn foo() {}

fn bar() {
    a::foo(); // is this calling the local crate or the dependency?
}
```

The above example compiles two crates with exactly the same name and `crate_type` (plus any other metadata). This causes an error because it becomes impossible for the compiler to distinguish between symbols (`pub` item names).

This error can be fixed by:

- Using Cargo, the Rust package manager, automatically fixing this issue.
- Recompiling the crate with different metadata (different name/ `crate_type`).

Error code E0520

A non-default implementation was already made on this type so it cannot be specialized further.

Erroneous code example:

```
#![feature(specialization)]

trait SpaceLlama {
    fn fly(&self);
}

// applies to all T
impl<T> SpaceLlama for T {
    default fn fly(&self) {}
}

// non-default impl
// applies to all `Clone` T and overrides the previous impl
impl<T: Clone> SpaceLlama for T {
    fn fly(&self) {}
}

// since `i32` is clone, this conflicts with the previous implementation
impl SpaceLlama for i32 {
    default fn fly(&self) {}
    // error: item `fly` is provided by an `impl` that specializes
    //        another, but the item in the parent `impl` is not marked
    //        `default` and so it cannot be specialized.
}
```

Specialization only allows you to override `default` functions in implementations.

To fix this error, you need to mark all the parent implementations as default. Example:

```
#![feature(specialization)]

trait SpaceLlama {
    fn fly(&self);
}

// applies to all T
impl<T> SpaceLlama for T {
    default fn fly(&self) {} // This is a parent implementation.
}

// applies to all `Clone` T; overrides the previous impl
impl<T: Clone> SpaceLlama for T {
    default fn fly(&self) {} // This is a parent implementation but was
                             // previously not a default one, causing the
error
}

// applies to i32, overrides the previous two impls
impl SpaceLlama for i32 {
    fn fly(&self) {} // And now that's ok!
}
```

Error code E0521

Borrowed data escapes outside of closure.

Erroneous code example:

```
let mut list: Vec<&str> = Vec::new();

let _add = |el: &str| {
    list.push(el); // error: `el` escapes the closure body here
};
```

A type annotation of a closure parameter implies a new lifetime declaration. Consider to drop it, the compiler is reliably able to infer them.

```
let mut list: Vec<&str> = Vec::new();

let _add = |el| {
    list.push(el);
};
```

See the Closure type inference and annotation and Lifetime elision sections of the Book for more details.

Error code E0522

The lang attribute was used in an invalid context.

Erroneous code example:

```
#![feature(lang_items)]

#[lang = "cookie"]
fn cookie() -> ! { // error: definition of an unknown language item:
`cookie`
    loop {}
}
```

The lang attribute is intended for marking special items that are built-in to Rust itself. This includes special traits (like `Copy` and `Sized`) that affect how the compiler behaves, as well as special functions that may be automatically invoked (such as the handler for out-of-bounds accesses when indexing a slice).

Error code E0523

Note: this error code is no longer emitted by the compiler.

The compiler found multiple library files with the requested crate name.

```
// aux-build:crateresolve-1.rs
// aux-build:crateresolve-2.rs
// aux-build:crateresolve-3.rs

extern crate crateresolve;
//~^ ERROR multiple candidates for `rlib` dependency `crateresolve` found

fn main() {}
```

This error can occur in several different cases -- for example, when using `extern crate` or passing `--extern` options without crate paths. It can also be caused by caching issues with the build directory, in which case `cargo clean` may help.

In the above example, there are three different library files, all of which define the same crate name. Without providing a full path, there is no way for the compiler to know which crate it should use.

Note that E0523 has been merged into E0464.

Error code E0524

A variable which requires unique access is being used in more than one closure at the same time.

Erroneous code example:

```
fn set(x: &mut isize) {
    *x += 4;
}

fn dragoooon(x: &mut isize) {
    let mut c1 = || set(x);
    let mut c2 = || set(x); // error!

    c2();
    c1();
}
```

To solve this issue, multiple solutions are available. First, is it required for this variable to be used in more than one closure at a time? If it is the case, use reference counted types such as `Rc` (or `Arc` if it runs concurrently):

```
use std::rc::Rc;
use std::cell::RefCell;

fn set(x: &mut isize) {
    *x += 4;
}

fn dragoooon(x: &mut isize) {
    let x = Rc::new(RefCell::new(x));
    let y = Rc::clone(&x);
    let mut c1 = || { let mut x2 = x.borrow_mut(); set(&mut x2); };
    let mut c2 = || { let mut x2 = y.borrow_mut(); set(&mut x2); }; // ok!

    c2();
    c1();
}
```

If not, just run closures one at a time:

```
fn set(x: &mut isize) {
    *x += 4;
}

fn dragoooon(x: &mut isize) {
    { // This block isn't necessary since non-lexical lifetimes, it's just to
      // make it more clear.
        let mut c1 = || set(&mut *x);
        c1();
    } // `c1` has been dropped here so we're free to use `x` again!
    let mut c2 = || set(&mut *x);
    c2();
}
```

Error code E0525

A closure was used but didn't implement the expected trait.

Erroneous code example:

```
struct X;

fn foo<T>(_: T) {}
fn bar<T: Fn(u32)>(_: T) {}

fn main() {
    let x = X;
    let closure = |_| foo(x); // error: expected a closure that implements
                              //        the `Fn` trait, but this closure
only
                              //        implements `FnOnce`
    bar(closure);
}
```

In the example above, `closure` is an `FnOnce` closure whereas the `bar` function expected an `Fn` closure. In this case, it's simple to fix the issue, you just have to implement `Copy` and `Clone` traits on `struct X` and it'll be ok:

```
#[derive(Clone, Copy)] // We implement `Clone` and `Copy` traits.
struct X;

fn foo<T>(_: T) {}
fn bar<T: Fn(u32)>(_: T) {}

fn main() {
    let x = X;
    let closure = |_| foo(x);
    bar(closure); // ok!
}
```

To better understand how these work in Rust, read the Closures chapter of the Book.

Error code E0527

The number of elements in an array or slice pattern differed from the number of elements in the array being matched.

Example of erroneous code:

```
let r = &[1, 2, 3, 4];
match r {
    &[a, b] => { // error: pattern requires 2 elements but array
                 //        has 4
        println!("a={}, b={}", a, b);
    }
}
```

Ensure that the pattern is consistent with the size of the matched array. Additional elements can be matched with `..`:

```
let r = &[1, 2, 3, 4];
match r {
    &[a, b, ..] => { // ok!
        println!("a={}, b={}", a, b);
    }
}
```

Error code E0528

An array or slice pattern required more elements than were present in the matched array.

Example of erroneous code:

```
let r = &[1, 2];
match r {
    &[a, b, c, rest @ ..] => { // error: pattern requires at least 3
                               //        elements but array has 2
        println!("a={}, b={}, c={} rest={:?}", a, b, c, rest);
    }
}
```

Ensure that the matched array has at least as many elements as the pattern requires. You can match an arbitrary number of remaining elements with `..`:

```
let r = &[1, 2, 3, 4, 5];
match r {
    &[a, b, c, rest @ ..] => { // ok!
        // prints `a=1, b=2, c=3 rest=[4, 5]`
        println!("a={}, b={}, c={} rest={:?}", a, b, c, rest);
    }
}
```

Error code E0529

An array or slice pattern was matched against some other type.

Example of erroneous code:

```
let r: f32 = 1.0;
match r {
    [a, b] => { // error: expected an array or slice, found `f32`
        println!("a={}, b={}", a, b);
    }
}
```

Ensure that the pattern and the expression being matched on are of consistent types:

```
let r = [1.0, 2.0];
match r {
    [a, b] => { // ok!
        println!("a={}, b={}", a, b);
    }
}
```

Error code E0530

A binding shadowed something it shouldn't.

A match arm or a variable has a name that is already used by something else, e.g.

- struct name
- enum variant
- static
- associated constant

This error may also happen when an enum variant *with fields* is used in a pattern, but without its fields.

```
enum Enum {
    WithField(i32)
}

use Enum::*;
match WithField(1) {
    WithField => {} // error: missing (_)
}
```

Match bindings cannot shadow statics:

```
static TEST: i32 = 0;

let r = 123;
match r {
    TEST => {} // error: name of a static
}
```

Fixed examples:

```
static TEST: i32 = 0;

let r = 123;
match r {
    some_value => {} // ok!
}
```

or

```
const TEST: i32 = 0; // const, not static

let r = 123;
match r {
    TEST => {} // const is ok!
    other_values => {}
}
```

Error code E0531

An unknown tuple struct/variant has been used.

Erroneous code example:

```
let Type(x) = Type(12); // error!
match Bar(12) {
    Bar(x) => {} // error!
    _ => {}
}
```

In most cases, it's either a forgotten import or a typo. However, let's look at how you can have such a type:

```
struct Type(u32); // this is a tuple struct

enum Foo {
    Bar(u32), // this is a tuple variant
}

use Foo::*; // To use Foo's variant directly, we need to import them in
            // the scope.
```

Either way, it should work fine with our previous code:

```
struct Type(u32);

enum Foo {
    Bar(u32),
}
use Foo::*;

let Type(x) = Type(12); // ok!
match Type(12) {
    Type(x) => {} // ok!
    _ => {}
}
```

Error code E0532

Pattern arm did not match expected kind.

Erroneous code example:

```
enum State {
    Succeeded,
    Failed(String),
}

fn print_on_failure(state: &State) {
    match *state {
        // error: expected unit struct, unit variant or constant, found
tuple
        //        variant `State::Failed`
        State::Failed => println!("Failed"),
        _ => ()
    }
}
```

To fix this error, ensure the match arm kind is the same as the expression matched.

Fixed example:

```
enum State {
    Succeeded,
    Failed(String),
}

fn print_on_failure(state: &State) {
    match *state {
        State::Failed(ref msg) => println!("Failed with {}", msg),
        _ => ()
    }
}
```

Error code E0533

An item which isn't a unit struct, a variant, nor a constant has been used as a match pattern.

Erroneous code example:

```
struct Tortoise;

impl Tortoise {
    fn turtle(&self) -> u32 { 0 }
}

match 0u32 {
    Tortoise::turtle => {} // Error!
    _ => {}
}
if let Tortoise::turtle = 0u32 {} // Same error!
```

If you want to match against a value returned by a method, you need to bind the value first:

```
struct Tortoise;

impl Tortoise {
    fn turtle(&self) -> u32 { 0 }
}

match 0u32 {
    x if x == Tortoise.turtle() => {} // Bound into `x` then we compare it!
    _ => {}
}
```

Error code E0534

The `inline` attribute was malformed.

Erroneous code example:

```
#[inline()] // error: expected one argument
pub fn something() {}

fn main() {}
```

The parenthesized `inline` attribute requires the parameter to be specified:

```
#[inline(always)]
fn something() {}
```

or:

```
#[inline(never)]
fn something() {}
```

Alternatively, a paren-less version of the attribute may be used to hint the compiler about inlining opportunity:

```
#[inline]
fn something() {}
```

For more information see the `inline` attribute section of the Reference.

Error code E0535

An unknown argument was given to the `inline` attribute.

Erroneous code example:

```
#[inline(unknown)] // error: invalid argument
pub fn something() {}

fn main() {}
```

The `inline` attribute only supports two arguments:

- always
- never

All other arguments given to the `inline` attribute will return this error. Example:

```
#[inline(never)] // ok!
pub fn something() {}

fn main() {}
```

For more information see the `inline` Attribute section of the Reference.

Error code E0536

The `not` cfg-predicate was malformed.

Erroneous code example:

```
#[cfg(not())] // error: expected 1 cfg-pattern
pub fn something() {}

pub fn main() {}
```

The `not` predicate expects one cfg-pattern. Example:

```
#[cfg(not(target_os = "linux"))] // ok!
pub fn something() {}

pub fn main() {}
```

For more information about the `cfg` attribute, read the section on Conditional Compilation in the Reference.

Error code E0537

An unknown predicate was used inside the `cfg` attribute.

Erroneous code example:

```
#[cfg(unknown())] // error: invalid predicate `unknown`
pub fn something() {}

pub fn main() {}
```

The `cfg` attribute supports only three kinds of predicates:

- any
- all
- not

Example:

```
#[cfg(not(target_os = "linux"))] // ok!
pub fn something() {}

pub fn main() {}
```

For more information about the `cfg` attribute, read the section on Conditional Compilation in the Reference.

Error code E0538

Attribute contains same meta item more than once.

Erroneous code example:

```
#[deprecated(
    since="1.0.0",
    note="First deprecation note.",
    note="Second deprecation note." // error: multiple same meta item
)]
fn deprecated_function() {}
```

Meta items are the key-value pairs inside of an attribute. Each key may only be used once in each attribute.

To fix the problem, remove all but one of the meta items with the same key.

Example:

```
#[deprecated(
    since="1.0.0",
    note="First deprecation note."
)]
fn deprecated_function() {}
```

Error code E0539

An invalid meta-item was used inside an attribute.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[deprecated(note)] // error!
#[unstable(feature = "deprecated_fn", issue = "123")]
fn deprecated() {}

#[unstable(feature = "unstable_struct", issue)] // error!
struct Unstable;

#[rustc_const_unstable(feature)] // error!
const fn unstable_fn() {}

#[stable(feature = "stable_struct", since)] // error!
struct Stable;

#[rustc_const_stable(feature)] // error!
const fn stable_fn() {}
```

Meta items are the key-value pairs inside of an attribute. To fix these issues you need to give required key-value pairs.

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[deprecated(since = "1.39.0", note = "reason")] // ok!
#[unstable(feature = "deprecated_fn", issue = "123")]
fn deprecated() {}

#[unstable(feature = "unstable_struct", issue = "123")] // ok!
struct Unstable;

#[rustc_const_unstable(feature = "unstable_fn", issue = "124")] // ok!
const fn unstable_fn() {}

#[stable(feature = "stable_struct", since = "1.39.0")] // ok!
struct Stable;

#[rustc_const_stable(feature = "stable_fn", since = "1.39.0")] // ok!
const fn stable_fn() {}
```

Error code E0541

An unknown meta item was used.

Erroneous code example:

```
#[deprecated(
    since="1.0.0",
    // error: unknown meta item
    reason="Example invalid meta item. Should be 'note'")
]
fn deprecated_function() {}
```

Meta items are the key-value pairs inside of an attribute. The keys provided must be one of the valid keys for the specified attribute.

To fix the problem, either remove the unknown meta item, or rename it if you provided the wrong name.

In the erroneous code example above, the wrong name was provided, so changing to a correct one it will fix the error. Example:

```
#[deprecated(
    since="1.0.0",
    note="This is a valid meta item for the deprecated attribute."
)]
fn deprecated_function() {}
```

Error code E0542

The `since` value is missing in a stability attribute.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[stable(feature = "_stable_fn")] // invalid
fn _stable_fn() {}

#[rustc_const_stable(feature = "_stable_const_fn")] // invalid
const fn _stable_const_fn() {}

#[stable(feature = "_deprecated_fn", since = "0.1.0")]
#[deprecated(
    note = "explanation for deprecation"
)] // invalid
fn _deprecated_fn() {}
```

To fix this issue, you need to provide the `since` field. Example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[stable(feature = "_stable_fn", since = "1.0.0")] // ok!
fn _stable_fn() {}

#[rustc_const_stable(feature = "_stable_const_fn", since = "1.0.0")] // ok!
const fn _stable_const_fn() {}

#[stable(feature = "_deprecated_fn", since = "0.1.0")]
#[deprecated(
    since = "1.0.0",
    note = "explanation for deprecation"
)] // ok!
fn _deprecated_fn() {}
```

See the How Rust is Made and “Nightly Rust” appendix of the Book and the Stability attributes section of the Rustc Dev Guide for more details.

Error code E0543

The `note` value is missing in a stability attribute.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[stable(since = "0.1.0", feature = "_deprecated_fn")]
#[deprecated(
    since = "1.0.0"
)] // invalid
fn _deprecated_fn() {}
```

To fix this issue, you need to provide the `note` field. Example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[stable(since = "0.1.0", feature = "_deprecated_fn")]
#[deprecated(
    since = "1.0.0",
    note = "explanation for deprecation"
)] // ok!
fn _deprecated_fn() {}
```

See the How Rust is Made and “Nightly Rust” appendix of the Book and the Stability attributes section of the Rustc Dev Guide for more details.

Error code E0544

Multiple stability attributes were declared on the same item.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "rust1")]

#[stable(feature = "rust1", since = "1.0.0")]
#[stable(feature = "test", since = "2.0.0")] // invalid
fn foo() {}
```

To fix this issue, ensure that each item has at most one stability attribute.

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "rust1")]

#[stable(feature = "test", since = "2.0.0")] // ok!
fn foo() {}
```

See the How Rust is Made and “Nightly Rust” appendix of the Book and the Stability attributes section of the Rustc Dev Guide for more details.

Error code E0545

The `issue` value is incorrect in a stability attribute.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[unstable(feature = "_unstable_fn", issue = "0")] // invalid
fn _unstable_fn() {}

#[rustc_const_unstable(feature = "_unstable_const_fn", issue = "0")] //
invalid
const fn _unstable_const_fn() {}
```

To fix this issue, you need to provide a correct value in the `issue` field. Example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[unstable(feature = "_unstable_fn", issue = "none")] // ok!
fn _unstable_fn() {}

#[rustc_const_unstable(feature = "_unstable_const_fn", issue = "1")] // ok!
const fn _unstable_const_fn() {}
```

See the How Rust is Made and “Nightly Rust” appendix of the Book and the Stability attributes section of the Rustc Dev Guide for more details.

Error code E0546

The `feature` value is missing in a stability attribute.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[unstable(issue = "none")] // invalid
fn unstable_fn() {}

#[stable(since = "1.0.0")] // invalid
fn stable_fn() {}
```

To fix this issue, you need to provide the `feature` field. Example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[unstable(feature = "unstable_fn", issue = "none")] // ok!
fn unstable_fn() {}

#[stable(feature = "stable_fn", since = "1.0.0")] // ok!
fn stable_fn() {}
```

See the How Rust is Made and “Nightly Rust” appendix of the Book and the Stability attributes section of the Rustc Dev Guide for more details.

Error code E0547

The `issue` value is missing in a stability attribute.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[unstable(feature = "_unstable_fn")] // invalid
fn _unstable_fn() {}

#[rustc_const_unstable(feature = "_unstable_const_fn")] // invalid
const fn _unstable_const_fn() {}
```

To fix this issue, you need to provide the `issue` field. Example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[unstable(feature = "_unstable_fn", issue = "none")] // ok!
fn _unstable_fn() {}

#[rustc_const_unstable(
    feature = "_unstable_const_fn",
    issue = "none"
)] // ok!
const fn _unstable_const_fn() {}
```

See the How Rust is Made and “Nightly Rust” appendix of the Book and the Stability attributes section of the Rustc Dev Guide for more details.

Error code E0549

A `deprecated` attribute wasn't paired with a `stable`/`unstable` attribute with `#![feature(staged_api)]` enabled.

Erroneous code example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[deprecated(
    since = "1.0.1",
    note = "explanation for deprecation"
)] // invalid
fn _deprecated_fn() {}
```

To fix this issue, you need to add also an attribute `stable` or `unstable`. Example:

```
#![feature(staged_api)]
#![allow(internal_features)]
#![stable(since = "1.0.0", feature = "test")]

#[stable(since = "1.0.0", feature = "test")]
#[deprecated(
    since = "1.0.1",
    note = "explanation for deprecation"
)] // ok!
fn _deprecated_fn() {}
```

See the How Rust is Made and “Nightly Rust” appendix of the Book and the Stability attributes section of the Rustc Dev Guide for more details.

Error code E0550

Note: this error code is no longer emitted by the compiler

More than one `deprecated` attribute has been put on an item.

Erroneous code example:

```
#[deprecated(note = "because why not?")]
#[deprecated(note = "right?")] // error!
fn the_banished() {}
```

The `deprecated` attribute can only be present **once** on an item.

```
#[deprecated(note = "because why not, right?")]
fn the_banished() {} // ok!
```

Error code E0551

Note: this error code is no longer emitted by the compiler

An invalid meta-item was used inside an attribute.

Erroneous code example:

```
#[deprecated(note)] // error!
fn i_am_deprecated() {}
```

Meta items are the key-value pairs inside of an attribute. To fix this issue, you need to give a value to the `note` key. Example:

```
#[deprecated(note = "because")] // ok!
fn i_am_deprecated() {}
```

Error code E0552

A unrecognized representation attribute was used.

Erroneous code example:

```
#[repr(D)] // error: unrecognized representation hint
struct MyStruct {
    my_field: usize
}
```

You can use a `repr` attribute to tell the compiler how you want a struct or enum to be laid out in memory.

Make sure you're using one of the supported options:

```
#[repr(C)] // ok!
struct MyStruct {
    my_field: usize
}
```

For more information about specifying representations, see the "Alternative Representations" section of the Rustonomicon.

Error code E0554

Feature attributes are only allowed on the nightly release channel. Stable or beta compilers will not comply.

Erroneous code example:

```
#![feature(lang_items)] // error: `#![feature]` may not be used on the
                        //        stable release channel
```

If you need the feature, make sure to use a nightly release of the compiler (but be warned that the feature may be removed or altered in the future).

Error code E0556

The `feature` attribute was badly formed.

Erroneous code example:

```
#![feature(foo_bar_baz, foo(bar), foo = "baz", foo)] // error!
#![feature] // error!
#![feature = "foo"] // error!
```

The `feature` attribute only accept a "feature flag" and can only be used on nightly. Example:

```
#![feature(flag)]
```

Error code E0557

A feature attribute named a feature that has been removed.

Erroneous code example:

```
#![feature(managed_boxes)] // error: feature has been removed
```

Delete the offending feature attribute.

Error code E0559

An unknown field was specified into an enum's structure variant.

Erroneous code example:

```
enum Field {
    Fool { x: u32 },
}

let s = Field::Fool { joke: 0 };
// error: struct variant `Field::Fool` has no field named `joke`
```

Verify you didn't misspell the field's name or that the field exists. Example:

```
enum Field {
    Fool { joke: u32 },
}

let s = Field::Fool { joke: 0 }; // ok!
```

Error code E0560

An unknown field was specified into a structure.

Erroneous code example:

```
struct Simba {
    mother: u32,
}

let s = Simba { mother: 1, father: 0 };
// error: structure `Simba` has no field named `father`
```

Verify you didn't misspell the field's name or that the field exists. Example:

```
struct Simba {
    mother: u32,
    father: u32,
}

let s = Simba { mother: 1, father: 0 }; // ok!
```

Error code E0561

A non-ident or non-wildcard pattern has been used as a parameter of a function pointer type.

Erroneous code example:

```
type A1 = fn(mut param: u8); // error!
type A2 = fn(&param: u32); // error!
```

When using an alias over a function type, you cannot e.g. denote a parameter as being mutable.

To fix the issue, remove patterns (`_` is allowed though). Example:

```
type A1 = fn(param: u8); // ok!
type A2 = fn(_: u32); // ok!
```

You can also omit the parameter name:

```
type A3 = fn(i16); // ok!
```

Error code E0562

Abstract return types (written `impl Trait` for some trait `Trait`) are only allowed as function and inherent impl return types.

Erroneous code example:

```
fn main() {
    let count_to_ten: impl Iterator<Item=usize> = 0..10;
    // error: `impl Trait` not allowed outside of function and inherent
method
    //        return types
    for i in count_to_ten {
        println!("{}", i);
    }
}
```

Make sure `impl Trait` only appears in return-type position.

```
fn count_to_n(n: usize) -> impl Iterator<Item=usize> {
    0..n
}

fn main() {
    for i in count_to_n(10) {  // ok!
        println!("{}", i);
    }
}
```

See RFC 1522 for more details.

Error code E0565

A literal was used in a built-in attribute that doesn't support literals.

Erroneous code example:

```
#[repr("C")] // error: meta item in `repr` must be an identifier
struct Repr {}

fn main() {}
```

Literals in attributes are new and largely unsupported in built-in attributes. Work to support literals where appropriate is ongoing. Try using an unquoted name instead:

```
#[repr(C)] // ok!
struct Repr {}

fn main() {}
```

Error code E0566

Conflicting representation hints have been used on a same item.

Erroneous code example:

```
#[repr(u32, u64)]
enum Repr { A }
```

In most cases (if not all), using just one representation hint is more than enough. If you want to have a representation hint depending on the current architecture, use `cfg_attr`. Example:

```
#[cfg_attr(linux, repr(u32))]
#[cfg_attr(not(linux), repr(u64))]
enum Repr { A }
```

Error code E0567

Generics have been used on an auto trait.

Erroneous code example:

```
#![feature(auto_traits)]

auto trait Generic<T> {} // error!
```

Since an auto trait is implemented on all existing types, the compiler would not be able to infer the types of the trait's generic parameters.

To fix this issue, just remove the generics:

```
#![feature(auto_traits)]

auto trait Generic {} // ok!
```

Error code E0568

A super trait has been added to an auto trait.

Erroneous code example:

```
#![feature(auto_traits)]

auto trait Bound : Copy {} // error!

fn main() {}
```

Since an auto trait is implemented on all existing types, adding a super trait would filter out a lot of those types. In the current example, almost none of all the existing types could implement `Bound` because very few of them have the `Copy` trait.

To fix this issue, just remove the super trait:

```
#![feature(auto_traits)]

auto trait Bound {} // ok!

fn main() {}
```

Error code E0569

If an impl has a generic parameter with the `#[may_dangle]` attribute, then that impl must be declared as an `unsafe impl`.

Erroneous code example:

```
#![feature(dropck_eyepatch)]

struct Foo<X>(X);
impl<#[may_dangle] X> Drop for Foo<X> {
    fn drop(&mut self) { }
}
```

In this example, we are asserting that the destructor for `Foo` will not access any data of type `X`, and require this assertion to be true for overall safety in our program. The compiler does not currently attempt to verify this assertion; therefore we must tag this `impl` as unsafe.

Error code E0570

The requested ABI is unsupported by the current target.

The rust compiler maintains for each target a list of unsupported ABIs on that target. If an ABI is present in such a list this usually means that the target / ABI combination is currently unsupported by llvm.

If necessary, you can circumvent this check using custom target specifications.

Error code E0571

A `break` statement with an argument appeared in a non-`loop` loop.

Example of erroneous code:

```
let result = while true {
    if satisfied(i) {
        break 2 * i; // error: `break` with value from a `while` loop
    }
    i += 1;
};
```

The `break` statement can take an argument (which will be the value of the loop expression if the `break` statement is executed) in `loop` loops, but not `for`, `while`, or `while let` loops.

Make sure `break value;` statements only occur in `loop` loops:

```
let result = loop { // This is now a "loop" loop.
    if satisfied(i) {
        break 2 * i; // ok!
    }
    i += 1;
};
```

Error code E0572

A return statement was found outside of a function body.

Erroneous code example:

```
const FOO: u32 = return 0; // error: return statement outside of function
body

fn main() {}
```

To fix this issue, just remove the return keyword or move the expression into a function. Example:

```
const FOO: u32 = 0;

fn some_fn() -> u32 {
    return FOO;
}

fn main() {
    some_fn();
}
```

Error code E0573

Something other than a type has been used when one was expected.

Erroneous code examples:

```
enum Dragon {
    Born,
}

fn oblivion() -> Dragon::Born { // error!
    Dragon::Born
}

const HOBBIT: u32 = 2;
impl HOBBIT {} // error!

enum Wizard {
    Gandalf,
    Saruman,
}

trait Isengard {
    fn wizard(_: Wizard::Saruman); // error!
}
```

In all these errors, a type was expected. For example, in the first error, if we want to return the `Born` variant from the `Dragon` enum, we must set the function to return the enum and not its variant:

```
enum Dragon {
    Born,
}

fn oblivion() -> Dragon { // ok!
    Dragon::Born
}
```

In the second error, you can't implement something on an item, only on types. We would need to create a new type if we wanted to do something similar:

```
struct Hobbit(u32); // we create a new type

const HOBBIT: Hobbit = Hobbit(2);
impl Hobbit {} // ok!
```

In the third case, we tried to only expect one variant of the `Wizard` enum, which is not possible. To make this work, we need to using pattern matching over the `Wizard` enum:

```
enum Wizard {
    Gandalf,
    Saruman,
}

trait Isengard {
    fn wizard(w: Wizard) { // ok!
        match w {
            Wizard::Saruman => {
                // do something
            }
            _ => {} // ignore everything else
        }
    }
}
```

Error code E0574

Something other than a struct, variant or union has been used when one was expected.

Erroneous code example:

```
mod mordor {}

let sauron = mordor { x: () }; // error!

enum Jak {
    Daxter { i: isize },
}

let eco = Jak::Daxter { i: 1 };
match eco {
    Jak { i } => {} // error!
}
```

In all these errors, a type was expected. For example, in the first error, we tried to instantiate the `mordor` module, which is impossible. If you want to instantiate a type inside a module, you can do it as follow:

```
mod mordor {
    pub struct TheRing {
        pub x: usize,
    }
}

let sauron = mordor::TheRing { x: 1 }; // ok!
```

In the second error, we tried to bind the `Jak` enum directly, which is not possible: you can only bind one of its variants. To do so:

```
enum Jak {
    Daxter { i: isize },
}

let eco = Jak::Daxter { i: 1 };
match eco {
    Jak::Daxter { i } => {} // ok!
}
```

Error code E0575

Something other than a type or an associated type was given.

Erroneous code example:

```
enum Rick { Morty }

let _: <u8 as Rick>::Morty; // error!

trait Age {
    type Empire;
    fn Mythology() {}
}

impl Age for u8 {
    type Empire = u16;
}

let _: <u8 as Age>::Mythology; // error!
```

In both cases, we're declaring a variable (called `_`) and we're giving it a type. However, `<u8 as Rick>::Morty` and `<u8 as Age>::Mythology` aren't types, therefore the compiler throws an error.

`<u8 as Rick>::Morty` is an enum variant, you cannot use a variant as a type, you have to use the enum directly:

```
enum Rick { Morty }

let _: Rick; // ok!
```

`<u8 as Age>::Mythology` is a trait method, which is definitely not a type. However, the `Age` trait provides an associated type `Empire` which can be used as a type:

```
trait Age {
    type Empire;
    fn Mythology() {}
}

impl Age for u8 {
    type Empire = u16;
}

let _: <u8 as Age>::Empire; // ok!
```

Error code E0576

An associated item wasn't found in the given type.

Erroneous code example:

```
trait Hello {
    type Who;

    fn hello() -> <Self as Hello>::You; // error!
}
```

In this example, we tried to use the nonexistent associated type `You` of the `Hello` trait. To fix this error, use an existing associated type:

```
trait Hello {
    type Who;

    fn hello() -> <Self as Hello>::Who; // ok!
}
```

Error code E0577

Something other than a module was found in visibility scope.

Erroneous code example:

```
pub enum Sea {}

pub (in crate::Sea) struct Shark; // error!

fn main() {}
```

`Sea` is not a module, therefore it is invalid to use it in a visibility path. To fix this error we need to ensure `sea` is a module.

Please note that the visibility scope can only be applied on ancestors!

```
pub mod sea {
    pub (in crate::sea) struct Shark; // ok!
}

fn main() {}
```

Error code E0578

A module cannot be found and therefore, the visibility cannot be determined.

Erroneous code example:

```
foo!();

pub (in ::Sea) struct Shark; // error!

fn main() {}
```

Because of the call to the `foo` macro, the compiler guesses that the missing module could be inside it and fails because the macro definition cannot be found.

To fix this error, please be sure that the module is in scope:

```
pub mod Sea {
    pub (in crate::Sea) struct Shark;
}

fn main() {}
```

Error code E0579

A lower range wasn't less than the upper range.

Erroneous code example:

```
#![feature(exclusive_range_pattern)]

fn main() {
    match 5u32 {
        // This range is ok, albeit pointless.
        1..2 => {}
        // This range is empty, and the compiler can tell.
        5..5 => {} // error!
    }
}
```

When matching against an exclusive range, the compiler verifies that the range is non-empty. Exclusive range patterns include the start point but not the end point, so this is equivalent to requiring the start of the range to be less than the end of the range.

Error code E0580

The `main` function was incorrectly declared.

Erroneous code example:

```
fn main(x: i32) { // error: main function has wrong type
    println!("{}", x);
}
```

The `main` function prototype should never take arguments. Example:

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

If you want to get command-line arguments, use `std::env::args`. To exit with a specified exit code, use `std::process::exit`.

Error code E0581

In a `fn` type, a lifetime appears only in the return type and not in the arguments types.

Erroneous code example:

```
fn main() {
    // Here, `'a` appears only in the return type:
    let x: for<'a> fn() -> &'a i32;
}
```

The problem here is that the lifetime isn't constrained by any of the arguments, making it impossible to determine how long it's supposed to live.

To fix this issue, either use the lifetime in the arguments, or use the `'static` lifetime. Example:

```
fn main() {
    // Here, `'a` appears only in the return type:
    let x: for<'a> fn(&'a i32) -> &'a i32;
    let y: fn() -> &'static i32;
}
```

Note: The examples above used to be (erroneously) accepted by the compiler, but this was since corrected. See issue #33685 for more details.

Error code E0582

A lifetime is only present in an associated-type binding, and not in the input types to the trait.

Erroneous code example:

```
fn bar<F>(t: F)
    // No type can satisfy this requirement, since `'a` does not
    // appear in any of the input types (here, `i32`):
    where F: for<'a> Fn(i32) -> Option<&'a i32>
{
}

fn main() { }
```

To fix this issue, either use the lifetime in the inputs, or use `'static`. Example:

```
fn bar<F, G>(t: F, u: G)
    where F: for<'a> Fn(&'a i32) -> Option<&'a i32>,
          G: Fn(i32) -> Option<&'static i32>,
{
}

fn main() { }
```

Note: The examples above used to be (erroneously) accepted by the compiler, but this was since corrected. See issue #33685 for more details.

Error code E0583

A file wasn't found for an out-of-line module.

Erroneous code example:

```
mod file_that_doesnt_exist; // error: file not found for module

fn main() {}
```

Please be sure that a file corresponding to the module exists. If you want to use a module named `file_that_doesnt_exist`, you need to have a file named `file_that_doesnt_exist.rs` or `file_that_doesnt_exist/mod.rs` in the same directory.

Error code E0584

A doc comment that is not attached to anything has been encountered.

Erroneous code example:

```
trait Island {
    fn lost();

    /// I'm lost!
}
```

A little reminder: a doc comment has to be placed before the item it's supposed to document. So if you want to document the `Island` trait, you need to put a doc comment before it, not inside it. Same goes for the `lost` method: the doc comment needs to be before it:

```
/// I'm THE island!
trait Island {
    /// I'm lost!
    fn lost();
}
```

Error code E0585

A documentation comment that doesn't document anything was found.

Erroneous code example:

```
fn main() {
    // The following doc comment will fail:
    /// This is a useless doc comment!
}
```

Documentation comments need to be followed by items, including functions, types, modules, etc. Examples:

```
/// I'm documenting the following struct:
struct Foo;

/// I'm documenting the following function:
fn foo() {}
```

Error code E0586

An inclusive range was used with no end.

Erroneous code example:

```
fn main() {
    let tmp = vec![0, 1, 2, 3, 4, 4, 3, 3, 2, 1];
    let x = &tmp[1..=]; // error: inclusive range was used with no end
}
```

An inclusive range needs an end in order to *include* it. If you just need a start and no end, use a non-inclusive range (with `..`):

```
fn main() {
    let tmp = vec![0, 1, 2, 3, 4, 4, 3, 3, 2, 1];
    let x = &tmp[1..]; // ok!
}
```

Or put an end to your inclusive range:

```
fn main() {
    let tmp = vec![0, 1, 2, 3, 4, 4, 3, 3, 2, 1];
    let x = &tmp[1..=3]; // ok!
}
```

Error code E0587

A type has both `packed` and `align` representation hints.

Erroneous code example:

```
#[repr(packed, align(8))] // error!
struct Umbrella(i32);
```

You cannot use `packed` and `align` hints on a same type. If you want to pack a type to a given size, you should provide a size to packed:

```
#[repr(packed(8))] // ok!
struct Umbrella(i32);
```

Error code E0588

A type with `packed` representation hint has a field with `align` representation hint.

Erroneous code example:

```
#[repr(align(16))]
struct Aligned(i32);

#[repr(packed)] // error!
struct Packed(Aligned);
```

Just like you cannot have both `align` and `packed` representation hints on the same type, a `packed` type cannot contain another type with the `align` representation hint. However, you can do the opposite:

```
#[repr(packed)]
struct Packed(i32);

#[repr(align(16))] // ok!
struct Aligned(Packed);
```

Error code E0589

The value of `N` that was specified for `repr(align(N))` was not a power of two, or was greater than 2^29.

Erroneous code example:

```
#[repr(align(15))] // error: invalid `repr(align)` attribute: not a power of
two
enum Foo {
    Bar(u64),
}
```

Error code E0590

`break` or `continue` keywords were used in a condition of a `while` loop without a label.

Erroneous code code:

```
while break {}
```

`break` or `continue` must include a label when used in the condition of a `while` loop.

To fix this, add a label specifying which loop is being broken out of:

```
'foo: while break 'foo {}
```

Error code E0591

Per RFC 401, if you have a function declaration `foo` :

```
struct S;

// For the purposes of this explanation, all of these
// different kinds of `fn` declarations are equivalent:

fn foo(x: S) { /* ... */ }
extern "C" {
    fn foo(x: S);
}
impl S {
    fn foo(self) { /* ... */ }
}
```

the type of `foo` is **not** `fn(S)` , as one might expect. Rather, it is a unique, zero-sized marker type written here as `typeof(foo)` . However, `typeof(foo)` can be *coerced* to a function pointer `fn(S)` , so you rarely notice this:

```
let x: fn(S) = foo; // OK, coerces
```

The reason that this matter is that the type `fn(S)` is not specific to any particular function: it's a function *pointer*. So calling `x()` results in a virtual call, whereas `foo()` is statically dispatched, because the type of `foo` tells us precisely what function is being called.

As noted above, coercions mean that most code doesn't have to be concerned with this distinction. However, you can tell the difference when using **transmute** to convert a fn item into a fn pointer.

This is sometimes done as part of an FFI:

```
extern "C" fn foo(userdata: Box<i32>) {
    /* ... */
}

unsafe {
    let f: extern "C" fn(*mut i32) = transmute(foo);
    callback(f);
}
```

Here, transmute is being used to convert the types of the fn arguments. This pattern is incorrect because the type of `foo` is a function **item** (`typeof(foo)`), which is zero-sized, and the target type (`fn()`) is a function pointer, which is not zero-sized. This pattern should be rewritten. There are a few possible ways to do this:

- change the original fn declaration to match the expected signature, and do the cast in the fn body (the preferred option)
- cast the fn item of a fn pointer before calling transmute, as shown here:

```
# extern "C" fn foo(_: Box<i32>) {}
# use std::mem::transmute;
# unsafe {
let f: extern "C" fn(*mut i32) = transmute(foo as extern "C" fn(_));
let f: extern "C" fn(*mut i32) = transmute(foo as usize); // works too
# }
```

The same applies to transmutes to `*mut fn()`, which were observed in practice. Note though that use of this type is generally incorrect. The intention is typically to describe a function pointer, but just `fn()` alone suffices for that. `*mut fn()` is a pointer to a fn pointer. (Since these values are typically just passed to C code, however, this rarely makes a difference in practice.)

Error code E0592

This error occurs when you defined methods or associated functions with same name.

Erroneous code example:

```
struct Foo;

impl Foo {
    fn bar() {} // previous definition here
}

impl Foo {
    fn bar() {} // duplicate definition here
}
```

A similar error is E0201. The difference is whether there is one declaration block or not. To avoid this error, you must give each `fn` a unique name.

```
struct Foo;

impl Foo {
    fn bar() {}
}

impl Foo {
    fn baz() {} // define with different name
}
```

Error code E0593

You tried to supply an `Fn`-based type with an incorrect number of arguments than what was expected.

Erroneous code example:

```
fn foo<F: Fn()>(x: F) { }

fn main() {
    // [E0593] closure takes 1 argument but 0 arguments are required
    foo(|y| { });
}
```

You have to provide the same number of arguments as expected by the `Fn`-based type. So to fix the previous example, we need to remove the `y` argument:

```
fn foo<F: Fn()>(x: F) { }

fn main() {
    foo(|| { }); // ok!
}
```

Error code E0594

A non-mutable value was assigned a value.

Erroneous code example:

```
struct SolarSystem {
    earth: i32,
}

let ss = SolarSystem { earth: 3 };
ss.earth = 2; // error!
```

To fix this error, declare `ss` as mutable by using the `mut` keyword:

```
struct SolarSystem {
    earth: i32,
}

let mut ss = SolarSystem { earth: 3 }; // declaring `ss` as mutable
ss.earth = 2; // ok!
```

Error code E0595

Note: this error code is no longer emitted by the compiler.

Closures cannot mutate immutable captured variables.

Erroneous code example:

```
let x = 3; // error: closure cannot assign to immutable local variable `x`
let mut c = || { x += 1 };
```

Make the variable binding mutable:

```
let mut x = 3; // ok!
let mut c = || { x += 1 };
```

Error code E0596

This error occurs because you tried to mutably borrow a non-mutable variable.

Erroneous code example:

```
let x = 1;
let y = &mut x; // error: cannot borrow mutably
```

In here, `x` isn't mutable, so when we try to mutably borrow it in `y`, it fails. To fix this error, you need to make `x` mutable:

```
let mut x = 1;
let y = &mut x; // ok!
```

Error code E0597

This error occurs because a value was dropped while it was still borrowed.

Erroneous code example:

```
struct Foo<'a> {
    x: Option<&'a u32>,
}

let mut x = Foo { x: None };
{
    let y = 0;
    x.x = Some(&y); // error: `y` does not live long enough
}
println!("{:?}", x.x);
```

Here, `y` is dropped at the end of the inner scope, but it is borrowed by `x` until the `println`. To fix the previous example, just remove the scope so that `y` isn't dropped until after the println

```
struct Foo<'a> {
    x: Option<&'a u32>,
}

let mut x = Foo { x: None };

let y = 0;
x.x = Some(&y);

println!("{:?}", x.x);
```

Error code E0599

This error occurs when a method is used on a type which doesn't implement it:

Erroneous code example:

```
struct Mouth;

let x = Mouth;
x.chocolate(); // error: no method named `chocolate` found for type `Mouth`
               //        in the current scope
```

In this case, you need to implement the `chocolate` method to fix the error:

```
struct Mouth;

impl Mouth {
    fn chocolate(&self) { // We implement the `chocolate` method here.
        println!("Hmmm! I love chocolate!");
    }
}

let x = Mouth;
x.chocolate(); // ok!
```

Error code E0600

An unary operator was used on a type which doesn't implement it.

Erroneous code example:

```
enum Question {
    Yes,
    No,
}

!Question::Yes; // error: cannot apply unary operator `!` to type `Question`
```

In this case, `Question` would need to implement the `std::ops::Not` trait in order to be able to use `!` on it. Let's implement it:

```
use std::ops::Not;

enum Question {
    Yes,
    No,
}

// We implement the `Not` trait on the enum.
impl Not for Question {
    type Output = bool;

    fn not(self) -> bool {
        match self {
            Question::Yes => false, // If the `Answer` is `Yes`, then it
                                    // returns false.
            Question::No => true, // And here we do the opposite.
        }
    }
}

assert_eq!(!Question::Yes, false);
assert_eq!(!Question::No, true);
```

Error code E0601

No `main` function was found in a binary crate.

To fix this error, add a `main` function:

```
fn main() {
    // Your program will start here.
    println!("Hello world!");
}
```

If you don't know the basics of Rust, you can look at the Rust Book to get started.

Error code E0602

An unknown or invalid lint was used on the command line.

Erroneous code example:

```
rustc -D bogus rust_file.rs
```

Maybe you just misspelled the lint name or the lint doesn't exist anymore. Either way, try to update/remove it in order to fix the error.

Error code E0603

A private item was used outside its scope.

Erroneous code example:

```
mod foo {
    const PRIVATE: u32 = 0x_a_bad_1dea_u32; // This const is private, so we
                                            // can't use it outside of the
                                            // `foo` module.
}

println!("const value: {}", foo::PRIVATE); // error: constant `PRIVATE`
                                                  //        is private
```

In order to fix this error, you need to make the item public by using the `pub` keyword. Example:

```
mod foo {
    pub const PRIVATE: u32 = 0x_a_bad_1dea_u32; // We set it public by using
the
                                                // `pub` keyword.
}

println!("const value: {}", foo::PRIVATE); // ok!
```

Error code E0604

A cast to `char` was attempted on a type other than `u8`.

Erroneous code example:

```
0u32 as char; // error: only `u8` can be cast as `char`, not `u32`
```

`char` is a Unicode Scalar Value, an integer value from 0 to 0xD7FF and 0xE000 to 0x10FFFF. (The gap is for surrogate pairs.) Only `u8` always fits in those ranges so only `u8` may be cast to `char`.

To allow larger values, use `char::from_u32`, which checks the value is valid.

```
assert_eq!(86u8 as char, 'V'); // ok!
assert_eq!(char::from_u32(0x3B1), Some('α')); // ok!
assert_eq!(char::from_u32(0xD800), None); // not a USV.
```

For more information about casts, take a look at the Type cast section in The Reference Book.

Error code E0605

An invalid cast was attempted.

Erroneous code examples:

```
let x = 0u8;
x as Vec<u8>; // error: non-primitive cast: `u8` as `std::vec::Vec<u8>`

// Another example

let v = core::ptr::null::<u8>(); // So here, `v` is a `*const u8`.
v as &u8; // error: non-primitive cast: `*const u8` as `&u8`
```

Only primitive types can be cast into each other. Examples:

```
let x = 0u8;
x as u32; // ok!

let v = core::ptr::null::<u8>();
v as *const i8; // ok!
```

For more information about casts, take a look at the Type cast section in The Reference Book.

Error code E0606

An incompatible cast was attempted.

Erroneous code example:

```
let x = &0u8; // Here, `x` is a `&u8`.
let y: u32 = x as u32; // error: casting `&u8` as `u32` is invalid
```

When casting, keep in mind that only primitive types can be cast into each other. Example:

```
let x = &0u8;
let y: u32 = *x as u32; // We dereference it first and then cast it.
```

For more information about casts, take a look at the Type cast section in The Reference Book.

Error code E0607

A cast between a thin and a fat pointer was attempted.

Erroneous code example:

```
let v = core::ptr::null::<u8>();
v as *const [u8];
```

First: what are thin and fat pointers?

Thin pointers are "simple" pointers: they are purely a reference to a memory address.

Fat pointers are pointers referencing Dynamically Sized Types (also called DSTs). DSTs don't have a statically known size, therefore they can only exist behind some kind of pointer that contains additional information. For example, slices and trait objects are DSTs. In the case of slices, the additional information the fat pointer holds is their size.

To fix this error, don't try to cast directly between thin and fat pointers.

For more information about type casts, take a look at the section of the The Rust Reference on type cast expressions.

Error code E0608

An attempt to use index on a type which doesn't implement the `std::ops::Index` trait was performed.

Erroneous code example:

```
0u8[2]; // error: cannot index into a value of type `u8`
```

To be able to index into a type it needs to implement the `std::ops::Index` trait. Example:

```
let v: Vec<u8> = vec![0, 1, 2, 3];

// The `Vec` type implements the `Index` trait so you can do:
println!("{}", v[2]);
```

Error code E0609

Attempted to access a nonexistent field in a struct.

Erroneous code example:

```
struct StructWithFields {
    x: u32,
}

let s = StructWithFields { x: 0 };
println!("{}", s.foo); // error: no field `foo` on type `StructWithFields`
```

To fix this error, check that you didn't misspell the field's name or that the field actually exists. Example:

```
struct StructWithFields {
    x: u32,
}

let s = StructWithFields { x: 0 };
println!("{}", s.x); // ok!
```

Error code E0610

Attempted to access a field on a primitive type.

Erroneous code example:

```
let x: u32 = 0;
println!("{}", x.foo); // error: `{integer}` is a primitive type, therefore
                       //        doesn't have fields
```

Primitive types are the most basic types available in Rust and don't have fields. To access data via named fields, struct types are used. Example:

```
// We declare struct called `Foo` containing two fields:
struct Foo {
    x: u32,
    y: i64,
}

// We create an instance of this struct:
let variable = Foo { x: 0, y: -12 };
// And we can now access its fields:
println!("x: {}, y: {}", variable.x, variable.y);
```

For more information about primitives and structs, take a look at the Book.

Error code E0614

Attempted to dereference a variable which cannot be dereferenced.

Erroneous code example:

```
let y = 0u32;
*y; // error: type `u32` cannot be dereferenced
```

Only types implementing `std::ops::Deref` can be dereferenced (such as `&T`). Example:

```
let y = 0u32;
let x = &y;
// So here, `x` is a `&u32`, so we can dereference it:
*x; // ok!
```

Error code E0615

Attempted to access a method like a field.

Erroneous code example:

```
struct Foo {
    x: u32,
}

impl Foo {
    fn method(&self) {}
}

let f = Foo { x: 0 };
f.method; // error: attempted to take value of method `method` on type `Foo`
```

If you want to use a method, add `()` after it:

```
f.method();
```

However, if you wanted to access a field of a struct check that the field name is spelled correctly. Example:

```
println!("{}", f.x);
```

Error code E0616

Attempted to access a private field on a struct.

Erroneous code example:

```
mod some_module {
    pub struct Foo {
        x: u32, // So `x` is private in here.
    }

    impl Foo {
        pub fn new() -> Foo { Foo { x: 0 } }
    }
}

let f = some_module::Foo::new();
println!("{}", f.x); // error: field `x` of struct `some_module::Foo` is
private
```

If you want to access this field, you have two options:

1. Set the field public:

```
mod some_module {
    pub struct Foo {
        pub x: u32, // `x` is now public.
    }

    impl Foo {
        pub fn new() -> Foo { Foo { x: 0 } }
    }
}

let f = some_module::Foo::new();
println!("{}", f.x); // ok!
```

2. Add a getter function:

```
mod some_module {
    pub struct Foo {
        x: u32, // So `x` is still private in here.
    }

    impl Foo {
        pub fn new() -> Foo { Foo { x: 0 } }

        // We create the getter function here:
        pub fn get_x(&self) -> &u32 { &self.x }
    }
}

let f = some_module::Foo::new();
println!("{}", f.get_x()); // ok!
```

Error code E0617

Attempted to pass an invalid type of variable into a variadic function.

Erroneous code example:

```
extern "C" {
    fn printf(format: *const c_char, ...) -> c_int;
}

unsafe {
    printf("%f\n\0".as_ptr() as _, 0f32);
    // error: cannot pass an `f32` to variadic function, cast to `c_double`
}
```

Certain Rust types must be cast before passing them to a variadic function, because of arcane ABI rules dictated by the C standard. To fix the error, cast the value to the type specified by the error message (which you may need to import from `std::os::raw`).

In this case, `c_double` has the same size as `f64` so we can use it directly:

```
# use std::os::raw::{c_char, c_int};
# extern "C" {
#     fn printf(format: *const c_char, ...) -> c_int;
# }

unsafe {
    printf("%f\n\0".as_ptr() as _, 0f64); // ok!
}
```

Error code E0618

Attempted to call something which isn't a function nor a method.

Erroneous code examples:

```
enum X {
    Entry,
}

X::Entry(); // error: expected function, tuple struct or tuple variant,
            // found `X::Entry`

// Or even simpler:
let x = 0i32;
x(); // error: expected function, tuple struct or tuple variant, found `i32`
```

Only functions and methods can be called using `()`. Example:

```
// We declare a function:
fn i_am_a_function() {}

// And we call it:
i_am_a_function();
```

Error code E0619

Note: this error code is no longer emitted by the compiler.

The type-checker needed to know the type of an expression, but that type had not yet been inferred.

Erroneous code example:

```
let mut x = vec![];
match x.pop() {
    Some(v) => {
        // Here, the type of `v` is not (yet) known, so we
        // cannot resolve this method call:
        v.to_uppercase(); // error: the type of this value must be known in
                          //        this context
    }
    None => {}
}
```

Type inference typically proceeds from the top of the function to the bottom, figuring out types as it goes. In some cases -- notably method calls and overloadable operators like `*` -- the type checker may not have enough information *yet* to make progress. This can be true even if the rest of the function provides enough context (because the type-checker hasn't looked that far ahead yet). In this case, type annotations can be used to help it along.

To fix this error, just specify the type of the variable. Example:

```
let mut x: Vec<String> = vec![]; // We precise the type of the vec elements.
match x.pop() {
    Some(v) => {
        v.to_uppercase(); // Since rustc now knows the type of the vec
elements,
                          // we can use `v`'s methods.
    }
    None => {}
}
```

Error code E0620

A cast to an unsized type was attempted.

Erroneous code example:

```
let x = &[1_usize, 2] as [usize]; // error: cast to unsized type: `&[usize;
2]`
                                  //        as `[usize]`
```

In Rust, some types don't have a known size at compile-time. For example, in a slice type like `[u32]` , the number of elements is not known at compile-time and hence the overall size cannot be computed. As a result, such types can only be manipulated through a reference (e.g., `&T` or `&mut T`) or other pointer-type (e.g., `Box` or `Rc`). Try casting to a reference instead:

```
let x = &[1_usize, 2] as &[usize]; // ok!
```

Error code E0621

This error code indicates a mismatch between the lifetimes appearing in the function signature (i.e., the parameter types and the return type) and the data-flow found in the function body.

Erroneous code example:

```
fn foo<'a>(x: &'a i32, y: &i32) -> &'a i32 { // error: explicit lifetime
                                             //        required in the type
of
                                             //        `y`
    if x > y { x } else { y }
}
```

In the code above, the function is returning data borrowed from either `x` or `y`, but the `'a` annotation indicates that it is returning data only from `x`. To fix the error, the signature and the body must be made to match. Typically, this is done by updating the function signature. So, in this case, we change the type of `y` to `&'a i32`, like so:

```
fn foo<'a>(x: &'a i32, y: &'a i32) -> &'a i32 {
    if x > y { x } else { y }
}
```

Now the signature indicates that the function data borrowed from either `x` or `y`. Alternatively, you could change the body to not return data from `y`:

```
fn foo<'a>(x: &'a i32, y: &i32) -> &'a i32 {
    x
}
```

Error code E0622

An intrinsic was declared without being a function.

Erroneous code example:

```
#![feature(intrinsics)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    pub static breakpoint: fn(); // error: intrinsic must be a function
}

fn main() { unsafe { breakpoint(); } }
```

An intrinsic is a function available for use in a given programming language whose implementation is handled specially by the compiler. In order to fix this error, just declare a function. Example:

```
#![feature(intrinsics)]
#![allow(internal_features)]

extern "rust-intrinsic" {
    pub fn breakpoint(); // ok!
}

fn main() { unsafe { breakpoint(); } }
```

Error code E0623

A lifetime didn't match what was expected.

Erroneous code example:

```
struct Foo<'a, 'b, T>(std::marker::PhantomData<(&'a (), &'b (), T)>)
where
    T: Convert<'a, 'b>;

trait Convert<'a, 'b>: Sized {
    fn cast(&'a self) -> &'b Self;
}
impl<'long: 'short, 'short, T> Convert<'long, 'short> for T {
    fn cast(&'long self) -> &'short T {
        self
    }
}
// error
fn badboi<'in_, 'out, T>(
    x: Foo<'in_, 'out, T>,
    sadness: &'in_ T
) -> &'out T {
    sadness.cast()
}
```

In this example, we tried to set a value with an incompatible lifetime to another one (`'in_` is unrelated to `'out`). We can solve this issue in two different ways:

Either we make `'in_` live at least as long as `'out` :

```
struct Foo<'a, 'b, T>(std::marker::PhantomData<(&'a (), &'b (), T)>)
where
    T: Convert<'a, 'b>;

trait Convert<'a, 'b>: Sized {
    fn cast(&'a self) -> &'b Self;
}
impl<'long: 'short, 'short, T> Convert<'long, 'short> for T {
    fn cast(&'long self) -> &'short T {
        self
    }
}
fn badboi<'in_: 'out, 'out, T>(
    x: Foo<'in_, 'out, T>,
    sadness: &'in_ T
) -> &'out T {
    sadness.cast()
}
```

Or we use only one lifetime:

```
struct Foo<'a, 'b, T>(std::marker::PhantomData<(&'a (), &'b (), T)>)
where
    T: Convert<'a, 'b>;

trait Convert<'a, 'b>: Sized {
    fn cast(&'a self) -> &'b Self;
}
impl<'long: 'short, 'short, T> Convert<'long, 'short> for T {
    fn cast(&'long self) -> &'short T {
        self
    }
}
fn badboi<'out, T>(x: Foo<'out, 'out, T>, sadness: &'out T) -> &'out T {
    sadness.cast()
}
```

Error code E0624

A private item was used outside of its scope.

Erroneous code example:

```
mod inner {
    pub struct Foo;

    impl Foo {
        fn method(&self) {}
    }
}

let foo = inner::Foo;
foo.method(); // error: method `method` is private
```

Two possibilities are available to solve this issue:

1. Only use the item in the scope it has been defined:

```
mod inner {
    pub struct Foo;

    impl Foo {
        fn method(&self) {}
    }

    pub fn call_method(foo: &Foo) { // We create a public function.
        foo.method(); // Which calls the item.
    }
}

let foo = inner::Foo;
inner::call_method(&foo); // And since the function is public, we can call
the
                          // method through it.
```

2. Make the item public:

```
mod inner {
    pub struct Foo;

    impl Foo {
        pub fn method(&self) {} // It's now public.
    }
}

let foo = inner::Foo;
foo.method(); // Ok!
```

Error code E0625

A compile-time const variable is referring to a thread-local static variable.

Erroneous code example:

```
#![feature(thread_local)]

#[thread_local]
static X: usize = 12;

const Y: usize = 2 * X;
```

Static and const variables can refer to other const variables but a const variable cannot refer to a thread-local static variable. In this example, `Y` cannot refer to `X`. To fix this, the value can be extracted as a const and then used:

```
#![feature(thread_local)]

const C: usize = 12;

#[thread_local]
static X: usize = C;

const Y: usize = 2 * C;
```

Error code E0626

This error occurs because a borrow in a coroutine persists across a yield point.

Erroneous code example:

```
let mut b = || {
    let a = &String::new(); // <-- This borrow...
    yield (); // ...is still in scope here, when the yield occurs.
    println!("{}", a);
};
Pin::new(&mut b).resume(());
```

At present, it is not permitted to have a yield that occurs while a borrow is still in scope. To resolve this error, the borrow must either be "contained" to a smaller scope that does not overlap the yield or else eliminated in another way. So, for example, we might resolve the previous example by removing the borrow and just storing the integer by value:

```
let mut b = || {
    let a = 3;
    yield ();
    println!("{}", a);
};
Pin::new(&mut b).resume(());
```

This is a very simple case, of course. In more complex cases, we may wish to have more than one reference to the value that was borrowed -- in those cases, something like the `Rc` or `Arc` types may be useful.

This error also frequently arises with iteration:

```
let mut b = || {
  let v = vec![1,2,3];
  for &x in &v { // <-- borrow of `v` is still in scope...
    yield x; // ...when this yield occurs.
  }
};
Pin::new(&mut b).resume(());
```

Such cases can sometimes be resolved by iterating "by value" (or using `into_iter()`) to avoid borrowing:

```
let mut b = || {
  let v = vec![1,2,3];
  for x in v { // <-- Take ownership of the values instead!
    yield x; // <-- Now yield is OK.
  }
};
Pin::new(&mut b).resume(());
```

If taking ownership is not an option, using indices can work too:

```
let mut b = || {
  let v = vec![1,2,3];
  let len = v.len(); // (*)
  for i in 0..len {
    let x = v[i]; // (*)
    yield x; // <-- Now yield is OK.
  }
};
Pin::new(&mut b).resume(());

// (*) -- Unfortunately, these temporaries are currently required.
// See <https://github.com/rust-lang/rust/issues/43122>.
```

Error code E0627

A yield expression was used outside of the coroutine literal.

Erroneous code example:

```
#![feature(coroutines, coroutine_trait)]

fn fake_coroutine() -> &'static str {
    yield 1;
    return "foo"
}

fn main() {
    let mut coroutine = fake_coroutine;
}
```

The error occurs because keyword `yield` can only be used inside the coroutine literal. This can be fixed by constructing the coroutine correctly.

```
#![feature(coroutines, coroutine_trait)]

fn main() {
    let mut coroutine = || {
        yield 1;
        return "foo"
    };
}
```

Error code E0628

More than one parameter was used for a coroutine.

Erroneous code example:

```
#![feature(coroutines, coroutine_trait)]

fn main() {
    let coroutine = |a: i32, b: i32| {
        // error: too many parameters for a coroutine
        // Allowed only 0 or 1 parameter
        yield a;
    };
}
```

At present, it is not permitted to pass more than one explicit parameter for a coroutine.This can be fixed by using at most 1 parameter for the coroutine. For example, we might resolve the previous example by passing only one parameter.

```
#![feature(coroutines, coroutine_trait)]

fn main() {
    let coroutine = |a: i32| {
        yield a;
    };
}
```

Error code E0631

This error indicates a type mismatch in closure arguments.

Erroneous code example:

```
fn foo<F: Fn(i32)>(f: F) {
}

fn main() {
    foo(|x: &str| {});
}
```

The error occurs because `foo` accepts a closure that takes an `i32` argument, but in `main`, it is passed a closure with a `&str` argument.

This can be resolved by changing the type annotation or removing it entirely if it can be inferred.

```
fn foo<F: Fn(i32)>(f: F) {
}

fn main() {
    foo(|x: i32| {});
}
```

Error code E0632

Note: this error code is no longer emitted by the compiler.

An explicit generic argument was provided when calling a function that uses `impl Trait` in argument position.

Erroneous code example:

```
fn foo<T: Copy>(a: T, b: impl Clone) {}

foo::<i32>(0i32, "abc".to_string());
```

Either all generic arguments should be inferred at the call site, or the function definition should use an explicit generic type parameter instead of `impl Trait`. Example:

```
fn foo<T: Copy>(a: T, b: impl Clone) {}
fn bar<T: Copy, U: Clone>(a: T, b: U) {}

foo(0i32, "abc".to_string());

bar::<i32, String>(0i32, "abc".to_string());
bar::<_, _>(0i32, "abc".to_string());
bar(0i32, "abc".to_string());
```

Error code E0633

Note: this error code is no longer emitted by the compiler.

The `unwind` attribute was malformed.

Erroneous code example:

```
#![feature(unwind_attributes)]

#[unwind()] // error: expected one argument
pub extern "C" fn something() {}

fn main() {}
```

The `#[unwind]` attribute should be used as follows:

- `#[unwind(aborts)]` -- specifies that if a non-Rust ABI function should abort the process if it attempts to unwind. This is the safer and preferred option.
- `#[unwind(allowed)]` -- specifies that a non-Rust ABI function should be allowed to unwind. This can easily result in Undefined Behavior (UB), so be careful.

NB. The default behavior here is "allowed", but this is unspecified and likely to change in the future.

Error code E0634

A type has conflicting `packed` representation hints.

Erroneous code examples:

```
#[repr(packed, packed(2))] // error!
struct Company(i32);

#[repr(packed(2))] // error!
#[repr(packed)]
struct Company(i32);
```

You cannot use conflicting `packed` hints on a same type. If you want to pack a type to a given size, you should provide a size to packed:

```
#[repr(packed)] // ok!
struct Company(i32);
```

Error code E0635

The `#![feature]` attribute specified an unknown feature.

Erroneous code example:

```
#![feature(nonexistent_rust_feature)] // error: unknown feature
```

Error code E0636

A `#![feature]` attribute was declared multiple times.

Erroneous code example:

```
#![allow(stable_features)]
#![feature(rust1)]
#![feature(rust1)] // error: the feature `rust1` has already been declared
```

Error code E0637

`'_` lifetime name or `&T` without an explicit lifetime name has been used on illegal place.

Erroneous code example:

```
fn underscore_lifetime<'_>(str1: &'_ str, str2: &'_ str) -> &'_ str {
                     //^^ `'_` is a reserved lifetime name
    if str1.len() > str2.len() {
        str1
    } else {
        str2
    }
}

fn and_without_explicit_lifetime<T>()
where
    T: Into<&u32>,
          //^ `&` without an explicit lifetime name
{
}
```

First, `'_` cannot be used as a lifetime identifier in some places because it is a reserved for the anonymous lifetime. Second, `&T` without an explicit lifetime name cannot also be used in some places. To fix them, use a lowercase letter such as `'a`, or a series of lowercase letters such as `'foo`. For more information about lifetime identifier, see the book. For more information on using the anonymous lifetime in Rust 2018, see the Rust 2018 blog post.

Corrected example:

```
fn underscore_lifetime<'a>(str1: &'a str, str2: &'a str) -> &'a str {
    if str1.len() > str2.len() {
        str1
    } else {
        str2
    }
}

fn and_without_explicit_lifetime<'foo, T>()
where
    T: Into<&'foo u32>,
{
}
```

Error code E0638

This error indicates that the struct, enum or enum variant must be matched non-exhaustively as it has been marked as `non_exhaustive`.

When applied within a crate, downstream users of the crate will need to use the `_` pattern when matching enums and use the `..` pattern when matching structs. Downstream crates cannot match against non-exhaustive enum variants.

For example, in the below example, since the enum is marked as `non_exhaustive`, it is required that downstream crates match non-exhaustively on it.

```
#[non_exhaustive]
pub enum Error {
    Message(String),
    Other,
}

impl Display for Error {
    fn fmt(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        // This will not error, despite being marked as non_exhaustive, as
this
        // enum is defined within the current crate, it can be matched
        // exhaustively.
        let display = match self {
            Message(s) => s,
            Other => "other or unknown error",
        };
        formatter.write_str(display)
    }
}
```

An example of matching non-exhaustively on the above enum is provided below:

```
use mycrate::Error;

// This will not error as the non_exhaustive Error enum has been matched with
a
// wildcard.
match error {
    Message(s) => ...,
    Other => ...,
    _ => ...,
}
```

Similarly, for structs, match with `..` to avoid this error.

Error code E0639

This error indicates that the struct, enum or enum variant cannot be instantiated from outside of the defining crate as it has been marked as `non_exhaustive` and as such more fields/variants may be added in future that could cause adverse side effects for this code.

Erroneous code example:

```
#[non_exhaustive]
pub struct NormalStruct {
    pub first_field: u16,
    pub second_field: u16,
}

let ns = NormalStruct { first_field: 640, second_field: 480 }; // error!
```

It is recommended that you look for a `new` function or equivalent in the crate's documentation.

Error code E0640

This error code is internal to the compiler and will not be emitted with normal Rust code.

Error code E0641

Attempted to cast to/from a pointer with an unknown kind.

Erroneous code example:

```
let b = 0 as *const _; // error
```

Type information must be provided if a pointer type being cast from/into another type which cannot be inferred:

```
// Creating a pointer from reference: type can be inferred
let a = &(String::from("Hello world!")) as *const _; // ok!

let b = 0 as *const i32; // ok!

let c: *const i32 = 0 as *const _; // ok!
```

Error code E0642

Trait methods currently cannot take patterns as arguments.

Erroneous code example:

```
trait Foo {
    fn foo((x, y): (i32, i32)); // error: patterns aren't allowed
                                //        in trait methods
}
```

You can instead use a single name for the argument:

```
trait Foo {
    fn foo(x_and_y: (i32, i32)); // ok!
}
```

Error code E0643

This error indicates that there is a mismatch between generic parameters and impl Trait parameters in a trait declaration versus its impl.

```
trait Foo {
    fn foo(&self, _: &impl Iterator);
}
impl Foo for () {
    fn foo<U: Iterator>(&self, _: &U) { } // error method `foo` has
incompatible
                                          // signature for trait
}
```

Error code E0644

A closure or generator was constructed that references its own type.

Erroneous code example:

```
fn fix<F>(f: &F)
  where F: Fn(&F)
{
    f(&f);
}

fn main() {
    fix(&|y| {
        // Here, when `x` is called, the parameter `y` is equal to `x`.
    });
}
```

Rust does not permit a closure to directly reference its own type, either through an argument (as in the example above) or by capturing itself through its environment. This restriction helps keep closure inference tractable.

The easiest fix is to rewrite your closure into a top-level function, or into a method. In some cases, you may also be able to have your closure call itself by capturing a `&Fn()` object or `fn()` pointer that refers to itself. That is permitting, since the closure would be invoking itself via a virtual call, and hence does not directly reference its own *type*.

Error code E0646

It is not possible to define `main` with a where clause.

Erroneous code example:

```
fn main() where i32: Copy { // error: main function is not allowed to have
                            // a where clause
}
```

Error code E0647

The `start` function was defined with a where clause.

Erroneous code example:

```
#![feature(start)]

#[start]
fn start(_: isize, _: *const *const u8) -> isize where (): Copy {
    //^ error: `#[start]` function is not allowed to have a where clause
    0
}
```

Error code E0648

An `export_name` attribute contains null characters (`\0`).

Erroneous code example:

```
#[export_name="\0foo"] // error: `export_name` may not contain null
characters
pub fn bar() {}
```

To fix this error, remove the null characters:

```
#[export_name="foo"] // ok!
pub fn bar() {}
```

Error code E0657

A lifetime bound on a trait implementation was captured at an incorrect place.

Erroneous code example:

```
trait Id<T> {}
trait Lt<'a> {}

impl<'a> Lt<'a> for () {}
impl<T> Id<T> for T {}

fn free_fn_capture_hrtb_in_impl_trait()
    -> Box<for<'a> Id<impl Lt<'a>>> // error!
{
    Box::new(())
}

struct Foo;
impl Foo {
    fn impl_fn_capture_hrtb_in_impl_trait()
        -> Box<for<'a> Id<impl Lt<'a>>> // error!
    {
        Box::new(())
    }
}
```

Here, you have used the inappropriate lifetime in the `impl Trait`, The `impl Trait` can only capture lifetimes bound at the fn or impl level.

To fix this we have to define the lifetime at the function or impl level and use that lifetime in the `impl Trait`. For example you can define the lifetime at the function:

```
trait Id<T> {}
trait Lt<'a> {}

impl<'a> Lt<'a> for () {}
impl<T> Id<T> for T {}

fn free_fn_capture_hrtb_in_impl_trait<'b>()
    -> Box<for<'a> Id<impl Lt<'b>>> // ok!
{
    Box::new(())
}

struct Foo;
impl Foo {
    fn impl_fn_capture_hrtb_in_impl_trait<'b>()
        -> Box<for<'a> Id<impl Lt<'b>>> // ok!
    {
        Box::new(())
    }
}
```

Error code E0658

An unstable feature was used.

Erroneous code example:

```
#[repr(u128)] // error: use of unstable library feature 'repr128'
enum Foo {
    Bar(u64),
}
```

If you're using a stable or a beta version of rustc, you won't be able to use any unstable features. In order to do so, please switch to a nightly version of rustc (by using rustup).

If you're using a nightly version of rustc, just add the corresponding feature to be able to use it:

```
#![feature(repr128)]

#[repr(u128)] // ok!
enum Foo {
    Bar(u64),
}
```

Error code E0659

An item usage is ambiguous.

Erroneous code example:

```
pub mod moon {
    pub fn foo() {}
}

pub mod earth {
    pub fn foo() {}
}

mod collider {
    pub use crate::moon::*;
    pub use crate::earth::*;
}

fn main() {
    crate::collider::foo(); // ERROR: `foo` is ambiguous
}
```

This error generally appears when two items with the same name are imported into a module. Here, the `foo` functions are imported and reexported from the `collider` module and therefore, when we're using `collider::foo()`, both functions collide.

To solve this error, the best solution is generally to keep the path before the item when using it. Example:

```
pub mod moon {
    pub fn foo() {}
}

pub mod earth {
    pub fn foo() {}
}

mod collider {
    pub use crate::moon;
    pub use crate::earth;
}

fn main() {
    crate::collider::moon::foo(); // ok!
    crate::collider::earth::foo(); // ok!
}
```

Error code E0660

Note: this error code is no longer emitted by the compiler.

The argument to the `llvm_asm` macro is not well-formed.

Erroneous code example:

```
llvm_asm!("nop" "nop");
```

Error code E0661

Note: this error code is no longer emitted by the compiler.

An invalid syntax was passed to the second argument of an `llvm_asm` macro line.

Erroneous code example:

```
let a;
llvm_asm!("nop" : "r"(a));
```

Error code E0662

Note: this error code is no longer emitted by the compiler.

An invalid input operand constraint was passed to the `llvm_asm` macro (third line).

Erroneous code example:

```
llvm_asm!("xor %eax, %eax"
          :
          : "=test"("a")
         );
```

Error code E0663

Note: this error code is no longer emitted by the compiler.

An invalid input operand constraint was passed to the `llvm_asm` macro (third line).

Erroneous code example:

```
llvm_asm!("xor %eax, %eax"
          :
          : "+test"("a")
         );
```

Error code E0664

Note: this error code is no longer emitted by the compiler.

A clobber was surrounded by braces in the `llvm_asm` macro.

Erroneous code example:

```
llvm_asm!("mov $$0x200, %eax"
          :
          :
          : "{eax}"
         );
```

Error code E0665

Note: this error code is no longer emitted by the compiler.

The `Default` trait was derived on an enum.

Erroneous code example:

```
#[derive(Default)]
enum Food {
    Sweet,
    Salty,
}
```

The `Default` cannot be derived on an enum for the simple reason that the compiler doesn't know which value to pick by default whereas it can for a struct as long as all its fields implement the `Default` trait as well.

If you still want to implement `Default` on your enum, you'll have to do it "by hand":

```
enum Food {
    Sweet,
    Salty,
}

impl Default for Food {
    fn default() -> Food {
        Food::Sweet
    }
}
```

Error code E0666

`impl Trait` types cannot appear nested in the generic arguments of other `impl Trait` types.

Erroneous code example:

```
trait MyGenericTrait<T> {}
trait MyInnerTrait {}

fn foo(
    bar: impl MyGenericTrait<impl MyInnerTrait>, // error!
) {}
```

Type parameters for `impl Trait` types must be explicitly defined as named generic parameters:

```
trait MyGenericTrait<T> {}
trait MyInnerTrait {}

fn foo<T: MyInnerTrait>(
    bar: impl MyGenericTrait<T>, // ok!
) {}
```

Error code E0667

`impl Trait` is not allowed in path parameters.

Erroneous code example:

```
fn some_fn(mut x: impl Iterator) -> <impl Iterator>::Item { // error!
    x.next().unwrap()
}
```

You cannot use `impl Trait` in path parameters. If you want something equivalent, you can do this instead:

```
fn some_fn<T: Iterator>(mut x: T) -> T::Item { // ok!
    x.next().unwrap()
}
```

Error code E0668

Note: this error code is no longer emitted by the compiler.

Malformed inline assembly rejected by LLVM.

Erroneous code example:

```
#![feature(llvm_asm)]

fn main() {
    let rax: u64;
    unsafe {
        llvm_asm!("" :"={rax"(rax));
        println!("Accumulator is: {}", rax);
    }
}
```

LLVM checks the validity of the constraints and the assembly string passed to it. This error implies that LLVM seems something wrong with the inline assembly call.

In particular, it can happen if you forgot the closing bracket of a register constraint (see issue #51430), like in the previous code example.

Error code E0669

Note: this error code is no longer emitted by the compiler.

Cannot convert inline assembly operand to a single LLVM value.

Erroneous code example:

```
#![feature(llvm_asm)]

fn main() {
    unsafe {
        llvm_asm!("" :: "r"("")); // error!
    }
}
```

This error usually happens when trying to pass in a value to an input inline assembly operand that is actually a pair of values. In particular, this can happen when trying to pass in a slice, for instance a `&str`. In Rust, these values are represented internally as a pair of values, the pointer and its length. When passed as an input operand, this pair of values can not be coerced into a register and thus we must fail with an error.

Error code E0670

Rust 2015 does not permit the use of `async fn`.

Erroneous code example:

```
async fn foo() {}
```

Switch to the Rust 2018 edition to use `async fn`.

Error code E0671

Note: this error code is no longer emitted by the compiler.

Const parameters cannot depend on type parameters. The following is therefore invalid:

```
fn const_id<T, const N: T>() -> T { // error
    N
}
```

Error code E0687

Note: this error code is no longer emitted by the compiler.

In-band lifetimes cannot be used in `fn` / `Fn` syntax.

Erroneous code examples:

```
#![feature(in_band_lifetimes)]

fn foo(x: fn(&'a u32)) {} // error!

fn bar(x: &Fn(&'a u32)) {} // error!

fn baz(x: fn(&'a u32), y: &'a u32) {} // error!

struct Foo<'a> { x: &'a u32 }

impl Foo<'a> {
    fn bar(&self, x: fn(&'a u32)) {} // error!
}
```

Lifetimes used in `fn` or `Fn` syntax must be explicitly declared using `<...>` binders. For example:

```
fn foo<'a>(x: fn(&'a u32)) {} // ok!

fn bar<'a>(x: &Fn(&'a u32)) {} // ok!

fn baz<'a>(x: fn(&'a u32), y: &'a u32) {} // ok!

struct Foo<'a> { x: &'a u32 }

impl<'a> Foo<'a> {
    fn bar(&self, x: fn(&'a u32)) {} // ok!
}
```

Error code E0688

Note: this error code is no longer emitted by the compiler.

In-band lifetimes were mixed with explicit lifetime binders.

Erroneous code example:

```
#![feature(in_band_lifetimes)]

fn foo<'a>(x: &'a u32, y: &'b u32) {}   // error!

struct Foo<'a> { x: &'a u32 }

impl Foo<'a> {
    fn bar<'b>(x: &'a u32, y: &'b u32, z: &'c u32) {}   // error!
}

impl<'b> Foo<'a> {  // error!
    fn baz() {}
}
```

In-band lifetimes cannot be mixed with explicit lifetime binders. For example:

```
fn foo<'a, 'b>(x: &'a u32, y: &'b u32) {}   // ok!

struct Foo<'a> { x: &'a u32 }

impl<'a> Foo<'a> {
    fn bar<'b,'c>(x: &'a u32, y: &'b u32, z: &'c u32) {}    // ok!
}

impl<'a> Foo<'a> {  // ok!
    fn baz() {}
}
```

Error code E0689

A method was called on an ambiguous numeric type.

Erroneous code example:

```
2.0.neg(); // error!
```

This error indicates that the numeric value for the method being passed exists but the type of the numeric value or binding could not be identified.

The error happens on numeric literals and on numeric bindings without an identified concrete type:

```
let x = 2.0;
x.neg();  // same error as above
```

Because of this, you must give the numeric literal or binding a type:

```
use std::ops::Neg;

let _ = 2.0_f32.neg(); // ok!
let x: f32 = 2.0;
let _ = x.neg(); // ok!
let _ = (2.0 as f32).neg(); // ok!
```

Error code E0690

A struct with the representation hint `repr(transparent)` had two or more fields that were not guaranteed to be zero-sized.

Erroneous code example:

```
#[repr(transparent)]
struct LengthWithUnit<U> { // error: transparent struct needs at most one
    value: f32,            //        non-zero-sized field, but has 2
    unit: U,
}
```

Because transparent structs are represented exactly like one of their fields at run time, said field must be uniquely determined. If there are multiple fields, it is not clear how the struct should be represented. Note that fields of zero-sized types (e.g., `PhantomData`) can also exist alongside the field that contains the actual data, they do not count for this error. When generic types are involved (as in the above example), an error is reported because the type parameter could be non-zero-sized.

To combine `repr(transparent)` with type parameters, `PhantomData` may be useful:

```
use std::marker::PhantomData;

#[repr(transparent)]
struct LengthWithUnit<U> {
    value: f32,
    unit: PhantomData<U>,
}
```

Error code E0691

Note: this error code is no longer emitted by the compiler.

A struct, enum, or union with the `repr(transparent)` representation hint contains a zero-sized field that requires non-trivial alignment.

Erroneous code example:

```
#![feature(repr_align)]

#[repr(align(32))]
struct ForceAlign32;

#[repr(transparent)]
struct Wrapper(f32, ForceAlign32); // error: zero-sized field in transparent
                                   //        struct has alignment of 32,
which
                                   //        is larger than 1
```

A transparent struct, enum, or union is supposed to be represented exactly like the piece of data it contains. Zero-sized fields with different alignment requirements potentially conflict with this property. In the example above, `Wrapper` would have to be aligned to 32 bytes even though `f32` has a smaller alignment requirement.

Consider removing the over-aligned zero-sized field:

```
#[repr(transparent)]
struct Wrapper(f32);
```

Alternatively, `PhantomData<T>` has alignment 1 for all `T`, so you can use it if you need to keep the field for some reason:

```
#![feature(repr_align)]

use std::marker::PhantomData;

#[repr(align(32))]
struct ForceAlign32;

#[repr(transparent)]
struct Wrapper(f32, PhantomData<ForceAlign32>);
```

Note that empty arrays `[T; 0]` have the same alignment requirement as the element type `T`. Also note that the error is conservatively reported even when the alignment of the zero-sized type is less than or equal to the data field's alignment.

Error code E0692

A `repr(transparent)` type was also annotated with other, incompatible representation hints.

Erroneous code example:

```
#[repr(transparent, C)] // error: incompatible representation hints
struct Grams(f32);
```

A type annotated as `repr(transparent)` delegates all representation concerns to another type, so adding more representation hints is contradictory. Remove either the `transparent` hint or the other hints, like this:

```
#[repr(transparent)]
struct Grams(f32);
```

Alternatively, move the other attributes to the contained type:

```
#[repr(C)]
struct Foo {
    x: i32,
    // ...
}

#[repr(transparent)]
struct FooWrapper(Foo);
```

Note that introducing another `struct` just to have a place for the other attributes may have unintended side effects on the representation:

```
#[repr(transparent)]
struct Grams(f32);

#[repr(C)]
struct Float(f32);

#[repr(transparent)]
struct Grams2(Float); // this is not equivalent to `Grams` above
```

Here, `Grams2` is a not equivalent to `Grams` -- the former transparently wraps a (non-transparent) struct containing a single float, while `Grams` is a transparent wrapper around a float. This can make a difference for the ABI.

Error code E0693

`align` representation hint was incorrectly declared.

Erroneous code examples:

```
#[repr(align=8)] // error!
struct Align8(i8);

#[repr(align="8")] // error!
struct Align8(i8);
```

This is a syntax error at the level of attribute declarations. The proper syntax for `align` representation hint is the following:

```
#[repr(align(8))] // ok!
struct Align8(i8);
```

Error code E0695

A `break` statement without a label appeared inside a labeled block.

Erroneous code example:

```
loop {
    'a: {
        break;
    }
}
```

Make sure to always label the `break`:

```
'l: loop {
    'a: {
        break 'l;
    }
}
```

Or if you want to `break` the labeled block:

```
loop {
    'a: {
        break 'a;
    }
    break;
}
```

Error code E0696

A function is using `continue` keyword incorrectly.

Erroneous code example:

```
fn continue_simple() {
    'b: {
        continue; // error!
    }
}
fn continue_labeled() {
    'b: {
        continue 'b; // error!
    }
}
fn continue_crossing() {
    loop {
        'b: {
            continue; // error!
        }
    }
}
```

Here we have used the `continue` keyword incorrectly. As we have seen above that `continue` pointing to a labeled block.

To fix this we have to use the labeled block properly. For example:

```
fn continue_simple() {
    'b: loop {
        continue ; // ok!
    }
}
fn continue_labeled() {
    'b: loop {
        continue 'b; // ok!
    }
}
fn continue_crossing() {
    loop {
        'b: loop {
            continue; // ok!
        }
    }
}
```

Error code E0697

A closure has been used as `static`.

Erroneous code example:

```
fn main() {
    static || {}; // used as `static`
}
```

Closures cannot be used as `static`. They "save" the environment, and as such a static closure would save only a static environment which would consist only of variables with a static lifetime. Given this it would be better to use a proper function. The easiest fix is to remove the `static` keyword.

Error code E0698

Note: this error code is no longer emitted by the compiler.

When using coroutines (or async) all type variables must be bound so a coroutine can be constructed.

Erroneous code example:

```
async fn bar<T>() -> () {}

async fn foo() {
    bar().await; // error: cannot infer type for `T`
}
```

In the above example `T` is unknowable by the compiler. To fix this you must bind `T` to a concrete type such as `String` so that a coroutine can then be constructed:

```
async fn bar<T>() -> () {}

async fn foo() {
    bar::<String>().await;
    //   ^^^^^^^^ specify type explicitly
}
```

Error code E0699

A method was called on a raw pointer whose inner type wasn't completely known.

Erroneous code example:

```
let foo = &1;
let bar = foo as *const _;
if bar.is_null() {
    // ...
}
```

Here, the type of `bar` isn't known; it could be a pointer to anything. Instead, specify a type for the pointer (preferably something that makes sense for the thing you're pointing to):

```
let foo = &1;
let bar = foo as *const i32;
if bar.is_null() {
    // ...
}
```

Even though `is_null()` exists as a method on any raw pointer, Rust shows this error because Rust allows for `self` to have arbitrary types (behind the arbitrary_self_types feature flag).

This means that someone can specify such a function:

```
impl Foo {
    fn is_null(self: *const Self) -> bool {
        // do something else
    }
}
```

and now when you call `.is_null()` on a raw pointer to `Foo`, there's ambiguity.

Given that we don't know what type the pointer is, and there's potential ambiguity for some types, we disallow calling methods on raw pointers when the type is unknown.

Error code E0700

The `impl Trait` return type captures lifetime parameters that do not appear within the `impl Trait` itself.

Erroneous code example:

```
use std::cell::Cell;

trait Trait<'a> { }

impl<'a, 'b> Trait<'b> for Cell<&'a u32> { }

fn foo<'x, 'y>(x: Cell<&'x u32>) -> impl Trait<'y>
where 'x: 'y
{
    x
}
```

Here, the function `foo` returns a value of type `Cell<&'x u32>`, which references the lifetime `'x`. However, the return type is declared as `impl Trait<'y>` -- this indicates that `foo` returns "some type that implements `Trait<'y>`", but it also indicates that the return type **only captures data referencing the lifetime** `'y`. In this case, though, we are referencing data with lifetime `'x`, so this function is in error.

To fix this, you must reference the lifetime `'x` from the return type. For example, changing the return type to `impl Trait<'y> + 'x` would work:

```
use std::cell::Cell;

trait Trait<'a> { }

impl<'a,'b> Trait<'b> for Cell<&'a u32> { }

fn foo<'x, 'y>(x: Cell<&'x u32>) -> impl Trait<'y> + 'x
where 'x: 'y
{
    x
}
```

Error code E0701

This error indicates that a `#[non_exhaustive]` attribute was incorrectly placed on something other than a struct or enum.

Erroneous code example:

```
#[non_exhaustive]
trait Foo { }
```

Error code E0703

Invalid ABI (Application Binary Interface) used in the code.

Erroneous code example:

```
extern "invalid" fn foo() {} // error!
```

At present few predefined ABI's (like Rust, C, system, etc.) can be used in Rust. Verify that the ABI is predefined. For example you can replace the given ABI from 'Rust'.

```
extern "Rust" fn foo() {} // ok!
```

Error code E0704

An incorrect visibility restriction was specified.

Erroneous code example:

```
mod foo {
    pub(foo) struct Bar {
        x: i32
    }
}
```

To make struct `Bar` only visible in module `foo` the `in` keyword should be used:

```
mod foo {
    pub(in crate::foo) struct Bar {
        x: i32
    }
}
```

For more information see the Rust Reference on Visibility.

Error code E0705

Note: this error code is no longer emitted by the compiler.

A `#![feature]` attribute was declared for a feature that is stable in the current edition, but not in all editions.

Erroneous code example:

```
#![feature(rust_2018_preview)]
#![feature(test_2018_feature)] // error: the feature
                               // `test_2018_feature` is
                               // included in the Rust 2018 edition
```

Error code E0706

Note: this error code is no longer emitted by the compiler.

`async fn`s are not yet supported in traits in Rust.

Erroneous code example:

```
trait T {
    // Neither case is currently supported.
    async fn foo() {}
    async fn bar(&self) {}
}
```

`async fn`s return an `impl Future`, making the following two examples equivalent:

```
async fn foo() -> User {
    unimplemented!()
}
// The async fn above gets desugared as follows:
fn foo(&self) -> impl Future<Output = User> + '_ {
    unimplemented!()
}
```

But when it comes to supporting this in traits, there are a few implementation issues. One of them is returning `impl Trait` in traits is not supported, as it would require Generic Associated Types to be supported:

```
impl MyDatabase {
    async fn get_user(&self) -> User {
        unimplemented!()
    }
}

impl MyDatabase {
    fn get_user(&self) -> impl Future<Output = User> + '_ {
        unimplemented!()
    }
}
```

Until these issues are resolved, you can use the `async-trait` crate, allowing you to use `async fn` in traits by desugaring to "boxed futures" (`Pin<Box<dyn Future + Send + 'async>>`).

Note that using these trait methods will result in a heap allocation per-function-call. This is not a significant cost for the vast majority of applications, but should be considered when deciding whether to use this functionality in the public API of a low-level function that is expected to be called millions of times a second.

You might be interested in visiting the async book for further information.

Error code E0708

`async` non-`move` closures with parameters are currently not supported.

Erroneous code example:

```
#![feature(async_closure)]

fn main() {
    let add_one = async |num: u8| { // error!
        num + 1
    };
}
```

`async` with non-move is currently not supported with the current version, you can use successfully by using move:

```
#![feature(async_closure)]

fn main() {
    let add_one = async move |num: u8| { // ok!
        num + 1
    };
}
```

Error code E0710

An unknown tool name was found in a scoped lint.

Erroneous code examples:

```
#[allow(clipp::filter_map)] // error!
fn main() {
    // business logic
}
```

```
#[warn(clipp::filter_map)] // error!
fn main() {
    // business logic
}
```

Please verify you didn't misspell the tool's name or that you didn't forget to import it in you project:

```
#[allow(clippy::filter_map)] // ok!
fn main() {
    // business logic
}
```

```
#[warn(clippy::filter_map)] // ok!
fn main() {
    // business logic
}
```

Error code E0712

A borrow of a thread-local variable was made inside a function which outlived the lifetime of the function.

Erroneous code example:

```
#![feature(thread_local)]

#[thread_local]
static FOO: u8 = 3;

fn main() {
    let a = &FOO; // error: thread-local variable borrowed past end of
function

    std::thread::spawn(move || {
        println!("{}", a);
    });
}
```

Error code E0713

This error occurs when an attempt is made to borrow state past the end of the lifetime of a type that implements the `Drop` trait.

Erroneous code example:

```
pub struct S<'a> { data: &'a mut String }

impl<'a> Drop for S<'a> {
    fn drop(&mut self) { self.data.push_str("being dropped"); }
}

fn demo<'a>(s: S<'a>) -> &'a mut String { let p = &mut *s.data; p }
```

Here, `demo` tries to borrow the string data held within its argument `s` and then return that borrow. However, `S` is declared as implementing `Drop`.

Structs implementing the `Drop` trait have an implicit destructor that gets called when they go out of scope. This destructor gets exclusive access to the fields of the struct when it runs.

This means that when `s` reaches the end of `demo`, its destructor gets exclusive access to its `&mut`-borrowed string data. allowing another borrow of that string data (`p`), to exist across the drop of `s` would be a violation of the principle that `&mut`-borrows have exclusive, unaliased access to their referenced data.

This error can be fixed by changing `demo` so that the destructor does not run while the string-data is borrowed; for example by taking `S` by reference:

```
pub struct S<'a> { data: &'a mut String }

impl<'a> Drop for S<'a> {
    fn drop(&mut self) { self.data.push_str("being dropped"); }
}

fn demo<'a>(s: &'a mut S<'a>) -> &'a mut String { let p = &mut *(*s).data; p
}
```

Note that this approach needs a reference to S with lifetime `'a`. Nothing shorter than `'a` will suffice: a shorter lifetime would imply that after `demo` finishes executing, something else (such as the destructor!) could access `s.data` after the end of that shorter lifetime, which would again violate the `&mut`-borrow's exclusive access.

Error code E0714

A `#[marker]` trait contained an associated item.

Erroneous code example:

```
#![feature(marker_trait_attr)]
#![feature(associated_type_defaults)]

#[marker]
trait MarkerConst {
    const N: usize; // error!
}

fn main() {}
```

The items of marker traits cannot be overridden, so there's no need to have them when they cannot be changed per-type anyway. If you wanted them for ergonomic reasons, consider making an extension trait instead.

Error code E0715

An `impl` for a `#[marker]` trait tried to override an associated item.

Erroneous code example:

```
#![feature(marker_trait_attr)]

#[marker]
trait Marker {
    const N: usize = 0;
    fn do_something() {}
}

struct OverrideConst;
impl Marker for OverrideConst { // error!
    const N: usize = 1;
}
```

Because marker traits are allowed to have multiple implementations for the same type, it's not allowed to override anything in those implementations, as it would be ambiguous which override should actually be used.

Error code E0716

A temporary value is being dropped while a borrow is still in active use.

Erroneous code example:

```
fn foo() -> i32 { 22 }
fn bar(x: &i32) -> &i32 { x }
let p = bar(&foo());
         // ------ creates a temporary
let q = *p;
```

Here, the expression `&foo()` is borrowing the expression `foo()`. As `foo()` is a call to a function, and not the name of a variable, this creates a **temporary** -- that temporary stores the return value from `foo()` so that it can be borrowed. You could imagine that `let p = bar(&foo());` is equivalent to the following, which uses an explicit temporary variable.

Erroneous code example:

```
let p = {
  let tmp = foo(); // the temporary
  bar(&tmp) // error: `tmp` does not live long enough
}; // <-- tmp is freed as we exit this block
let q = p;
```

Whenever a temporary is created, it is automatically dropped (freed) according to fixed rules. Ordinarily, the temporary is dropped at the end of the enclosing statement -- in this case, after the `let`. This is illustrated in the example above by showing that `tmp` would be freed as we exit the block.

To fix this problem, you need to create a local variable to store the value in rather than relying on a temporary. For example, you might change the original program to the following:

```
fn foo() -> i32 { 22 }
fn bar(x: &i32) -> &i32 { x }
let value = foo(); // dropped at the end of the enclosing block
let p = bar(&value);
let q = *p;
```

By introducing the explicit `let value`, we allocate storage that will last until the end of the enclosing block (when `value` goes out of scope). When we borrow `&value`, we are borrowing a local variable that already exists, and hence no temporary is created.

Temporaries are not always dropped at the end of the enclosing statement. In simple cases where the `&` expression is immediately stored into a variable, the compiler will automatically extend the lifetime of the temporary until the end of the enclosing block. Therefore, an alternative way to fix the original program is to write `let tmp = &foo()` and not `let tmp = foo()` :

```
fn foo() -> i32 { 22 }
fn bar(x: &i32) -> &i32 { x }
let value = &foo();
let p = bar(value);
let q = *p;
```

Here, we are still borrowing `foo()` , but as the borrow is assigned directly into a variable, the temporary will not be dropped until the end of the enclosing block. Similar rules apply when temporaries are stored into aggregate structures like a tuple or struct:

```
// Here, two temporaries are created, but
// as they are stored directly into `value`,
// they are not dropped until the end of the
// enclosing block.
fn foo() -> i32 { 22 }
let value = (&foo(), &foo());
```

Error code E0711

This error code is internal to the compiler and will not be emitted with normal Rust code.

Feature declared with conflicting stability requirements.

```
// NOTE: this attribute is perma-unstable and should *never* be used outside
of
//       stdlib and the compiler.
#![feature(staged_api)]

#![stable(feature = "...", since = "1.0.0")]

#[stable(feature = "foo", since = "1.0.0")]
fn foo_stable_1_0_0() {}

// error: feature `foo` is declared stable since 1.29.0
#[stable(feature = "foo", since = "1.29.0")]
fn foo_stable_1_29_0() {}

// error: feature `foo` is declared unstable
#[unstable(feature = "foo", issue = "none")]
fn foo_unstable() {}
```

In the above example, the `foo` feature is first defined to be stable since 1.0.0, but is then re-declared stable since 1.29.0. This discrepancy in versions causes an error. Furthermore, `foo` is then re-declared as unstable, again the conflict causes an error.

This error can be fixed by splitting the feature, this allows any stability requirements and removes any possibility of conflict.

Error code E0717

This error code is internal to the compiler and will not be emitted with normal Rust code.

Error code E0718

A `#[lang = ".."]` attribute was placed on the wrong item type.

Erroneous code example:

```
#![feature(lang_items)]

#[lang = "owned_box"]
static X: u32 = 42;
```

Error code E0719

An associated type value was specified more than once.

Erroneous code example:

```
#![feature(associated_type_bounds)]

trait FooTrait {}
trait BarTrait {}

// error: associated type `Item` in trait `Iterator` is specified twice
struct Foo<T: Iterator<Item: FooTrait, Item: BarTrait>> { f: T }
```

`Item` in trait `Iterator` cannot be specified multiple times for struct `Foo`. To fix this, create a new trait that is a combination of the desired traits and specify the associated type with the new trait.

Corrected example:

```
#![feature(associated_type_bounds)]

trait FooTrait {}
trait BarTrait {}
trait FooBarTrait: FooTrait + BarTrait {}

struct Foo<T: Iterator<Item: FooBarTrait>> { f: T } // ok!
```

For more information about associated types, see the book. For more information on associated type bounds, see RFC 2289.

Error code E0720

An `impl Trait` type expands to a recursive type.

Erroneous code example:

```
fn make_recursive_type() -> impl Sized {
    [make_recursive_type(), make_recursive_type()]
}
```

An `impl Trait` type must be expandable to a concrete type that contains no `impl Trait` types. For example the previous example tries to create an `impl Trait` type `T` that is equal to `[T, T]` .

Error code E0722

The `optimize` attribute was malformed.

Erroneous code example:

```
#![feature(optimize_attribute)]

#[optimize(something)] // error: invalid argument
pub fn something() {}
```

The `#[optimize]` attribute should be used as follows:

- `#[optimize(size)]` -- instructs the optimization pipeline to generate code that's smaller rather than faster
- `#[optimize(speed)]` -- instructs the optimization pipeline to generate code that's faster rather than smaller

For example:

```
#![feature(optimize_attribute)]

#[optimize(size)]
pub fn something() {}
```

See RFC 2412 for more details.

Error code E0724

`#[ffi_returns_twice]` was used on something other than a foreign function declaration.

Erroneous code example:

```
#![feature(ffi_returns_twice)]
#![crate_type = "lib"]

#[ffi_returns_twice] // error!
pub fn foo() {}
```

`#[ffi_returns_twice]` can only be used on foreign function declarations. For example, we might correct the previous example by declaring the function inside of an `extern` block.

```
#![feature(ffi_returns_twice)]

extern "C" {
   #[ffi_returns_twice] // ok!
   pub fn foo();
}
```

Error code E0725

A feature attribute named a feature that was disallowed in the compiler command line flags.

Erroneous code example:

```
#![feature(never_type)] // error: the feature `never_type` is not in
                        // the list of allowed features
```

Delete the offending feature attribute, or add it to the list of allowed features in the `-Z allow_features` flag.

Error code E0726

An argument lifetime was elided in an async function.

Erroneous code example:

When a struct or a type is bound/declared with a lifetime it is important for the Rust compiler to know, on usage, the lifespan of the type. When the lifetime is not explicitly mentioned and the Rust Compiler cannot determine the lifetime of your type, the following error occurs.

```
use futures::executor::block_on;
struct Content<'a> {
    title: &'a str,
    body: &'a str,
}
async fn create(content: Content) { // error: implicit elided
                                    // lifetime not allowed here
    println!("title: {}", content.title);
    println!("body: {}", content.body);
}
let content = Content { title: "Rust", body: "is great!" };
let future = create(content);
block_on(future);
```

Specify desired lifetime of parameter `content` or indicate the anonymous lifetime like `content: Content<'_>`. The anonymous lifetime tells the Rust compiler that `content` is only needed until the `create` function is done with its execution.

The `implicit elision` meaning the omission of suggested lifetime that is `pub async fn create<'a>(content: Content<'a>) {}` is not allowed here as lifetime of the `content` can differ from current context:

```
async fn create(content: Content<'_>) { // ok!
    println!("title: {}", content.title);
    println!("body: {}", content.body);
}
```

Know more about lifetime elision in this chapter and a chapter on lifetimes can be found here.

Error code E0727

A `yield` clause was used in an `async` context.

Erroneous code example:

```
#![feature(coroutines)]

fn main() {
    let coroutine = || {
        async {
            yield;
        }
    };
}
```

Here, the `yield` keyword is used in an `async` block, which is not yet supported.

To fix this error, you have to move `yield` out of the `async` block:

```
#![feature(coroutines)]

fn main() {
    let coroutine = || {
        yield;
    };
}
```

Error code E0728

`await` has been used outside `async` function or `async` block.

Erroneous code example:

```
fn foo() {
    wake_and_yield_once().await // `await` is used outside `async` context
}
```

`await` is used to suspend the current computation until the given future is ready to produce a value. So it is legal only within an `async` context, like an `async` function or an `async` block.

```
async fn foo() {
    wake_and_yield_once().await // `await` is used within `async` function
}

fn bar(x: u8) -> impl Future<Output = u8> {
    async move {
        wake_and_yield_once().await; // `await` is used within `async` block
        x
    }
}
```

Error code E0729

Note: this error code is no longer emitted by the compiler

Support for Non-Lexical Lifetimes (NLL) has been included in the Rust compiler since 1.31, and has been enabled on the 2015 edition since 1.36. The new borrow checker for NLL uncovered some bugs in the old borrow checker, which in some cases allowed unsound code to compile, resulting in memory safety issues.

What do I do?

Change your code so the warning does no longer trigger. For backwards compatibility, this unsound code may still compile (with a warning) right now. However, at some point in the future, the compiler will no longer accept this code and will throw a hard error.

Shouldn't you fix the old borrow checker?

The old borrow checker has known soundness issues that are basically impossible to fix. The new NLL-based borrow checker is the fix.

Can I turn these warnings into errors by denying a lint?

No.

When are these warnings going to turn into errors?

No formal timeline for turning the warnings into errors has been set. See GitHub issue 58781 for more information.

Why do I get this message with code that doesn't involve borrowing?

There are some known bugs that trigger this message.

Error code E0730

An array without a fixed length was pattern-matched.

Erroneous code example:

```
fn is_123<const N: usize>(x: [u32; N]) -> bool {
    match x {
        [1, 2, ..] => true, // error: cannot pattern-match on an
                            //        array without a fixed length
        _ => false
    }
}
```

To fix this error, you have two solutions:

1. Use an array with a fixed length.
2. Use a slice.

Example with an array with a fixed length:

```
fn is_123(x: [u32; 3]) -> bool { // We use an array with a fixed size
    match x {
        [1, 2, ..] => true, // ok!
        _ => false
    }
}
```

Example with a slice:

```
fn is_123(x: &[u32]) -> bool { // We use a slice
    match x {
        [1, 2, ..] => true, // ok!
        _ => false
    }
}
```

Error code E0731

An enum with the representation hint `repr(transparent)` had zero or more than one variants.

Erroneous code example:

```
#[repr(transparent)]
enum Status { // error: transparent enum needs exactly one variant, but has
2
    Errno(u32),
    Ok,
}
```

Because transparent enums are represented exactly like one of their variants at run time, said variant must be uniquely determined. If there is no variant, or if there are multiple variants, it is not clear how the enum should be represented.

Error code E0732

An `enum` with a discriminant must specify a `#[repr(inttype)]`.

Erroneous code example:

```
enum Enum { // error!
    Unit = 1,
    Tuple() = 2,
    Struct{} = 3,
}
```

A `#[repr(inttype)]` must be provided on an `enum` if it has a non-unit variant with a discriminant, or where there are both unit variants with discriminants and non-unit variants. This restriction ensures that there is a well-defined way to extract a variant's discriminant from a value; for instance:

```
#[repr(u8)]
enum Enum {
    Unit = 3,
    Tuple(u16) = 2,
    Struct {
        a: u8,
        b: u16,
    } = 1,
}

fn discriminant(v : &Enum) -> u8 {
    unsafe { *(v as *const Enum as *const u8) }
}

fn main() {
    assert_eq!(3, discriminant(&Enum::Unit));
    assert_eq!(2, discriminant(&Enum::Tuple(5)));
    assert_eq!(1, discriminant(&Enum::Struct{a: 7, b: 11}));
}
```

Error code E0733

An `async` function used recursion without boxing.

Erroneous code example:

```
async fn foo(n: usize) {
    if n > 0 {
        foo(n - 1).await;
    }
}
```

To perform async recursion, the `async fn` needs to be desugared such that the `Future` is explicit in the return type:

```
use std::future::Future;
fn foo_desugared(n: usize) -> impl Future<Output = ()> {
    async move {
        if n > 0 {
            foo_desugared(n - 1).await;
        }
    }
}
```

Finally, the future is wrapped in a pinned box:

```
use std::future::Future;
use std::pin::Pin;
fn foo_recursive(n: usize) -> Pin<Box<dyn Future<Output = ()>>> {
    Box::pin(async move {
        if n > 0 {
            foo_recursive(n - 1).await;
        }
    })
}
```

The `Box<...>` ensures that the result is of known size, and the pin is required to keep it in the same place in memory.

Error code E0734

A stability attribute has been used outside of the standard library.

Erroneous code example:

```
#[stable(feature = "a", since = "b")] // invalid
#[unstable(feature = "b", issue = "none")] // invalid
fn foo(){}
```

These attributes are meant to only be used by the standard library and are rejected in your own crates.

Error code E0735

Type parameter defaults cannot use `Self` on structs, enums, or unions.

Erroneous code example:

```
struct Foo<X = Box<Self>> {
    field1: Option<X>,
    field2: Option<X>,
}
// error: type parameters cannot use `Self` in their defaults.
```

Error code E0736

`#[track_caller]` and `#[naked]` cannot both be applied to the same function.

Erroneous code example:

```
#[naked]
#[track_caller]
fn foo() {}
```

This is primarily due to ABI incompatibilities between the two attributes. See RFC 2091 for details on this and other limitations.

Error code E0737

`#[track_caller]` requires functions to have the `"Rust"` ABI for implicitly receiving caller location. See RFC 2091 for details on this and other restrictions.

Erroneous code example:

```
#[track_caller]
extern "C" fn foo() {}
```

Error code E0739

`#[track_caller]` can not be applied on struct.

Erroneous code example:

```
#[track_caller]
struct Bar {
    a: u8,
}
```

Error code E0740

A `union` was declared with fields with destructors.

Erroneous code example:

```
union Test {
    a: A, // error!
}

#[derive(Debug)]
struct A(i32);

impl Drop for A {
    fn drop(&mut self) { println!("A"); }
}
```

A `union` cannot have fields with destructors.

Error code E0741

A non-structural-match type was used as the type of a const generic parameter.

Erroneous code example:

```
#![feature(adt_const_params)]

struct A;

struct B<const X: A>; // error!
```

Only structural-match types, which are types that derive `PartialEq` and `Eq` and implement `ConstParamTy`, may be used as the types of const generic parameters.

To fix the previous code example, we derive `PartialEq`, `Eq`, and `ConstParamTy`:

```
#![feature(adt_const_params)]

use std::marker::ConstParamTy;

#[derive(PartialEq, Eq, ConstParamTy)] // We derive both traits here.
struct A;

struct B<const X: A>; // ok!
```

Error code E0742

Visibility is restricted to a module which isn't an ancestor of the current item.

Erroneous code example:

```
pub mod sea {}

pub (in crate::sea) struct Shark; // error!

fn main() {}
```

To fix this error, we need to move the `Shark` struct inside the `sea` module:

```
pub mod sea {
    pub (in crate::sea) struct Shark; // ok!
}

fn main() {}
```

Of course, you can do it as long as the module you're referring to is an ancestor:

```
pub mod earth {
    pub mod sea {
        pub (in crate::earth) struct Shark; // ok!
    }
}

fn main() {}
```

Error code E0743

The C-variadic type `...` has been nested inside another type.

Erroneous code example:

```
fn foo2(x: u8, y: &...) {} // error!
```

Only foreign functions can use the C-variadic type (`...`). In such functions, `...` may only occur non-nested. That is, `y: &'a ...` is not allowed.

A C-variadic type is used to give an undefined number of parameters to a given function (like `printf` in C). The equivalent in Rust would be to use macros directly (like `println!` for example).

Error code E0744

Note: this error code is no longer emitted by the compiler.

An unsupported expression was used inside a const context.

Erroneous code example:

```
const _: i32 = {
    async { 0 }.await
};
```

At the moment, `.await` is forbidden inside a `const`, `static`, or `const fn`.

This may be allowed at some point in the future, but the implementation is not yet complete. See the tracking issue for `async` in `const fn`.

Error code E0745

The address of temporary value was taken.

Erroneous code example:

```
fn temp_address() {
    let ptr = &raw const 2; // error!
}
```

In this example, `2` is destroyed right after the assignment, which means that `ptr` now points to an unavailable location.

To avoid this error, first bind the temporary to a named local variable:

```
fn temp_address() {
    let val = 2;
    let ptr = &raw const val; // ok!
}
```

Error code E0746

An unboxed trait object was used as a return value.

Erroneous code example:

```
trait T {
    fn bar(&self);
}
struct S(usize);
impl T for S {
    fn bar(&self) {}
}

// Having the trait `T` as return type is invalid because
// unboxed trait objects do not have a statically known size:
fn foo() -> dyn T { // error!
    S(42)
}
```

Return types cannot be `dyn Trait` s as they must be `Sized` .

To avoid the error there are a couple of options.

If there is a single type involved, you can use `impl Trait` :

```
// The compiler will select `S(usize)` as the materialized return type of
this
// function, but callers will only know that the return type implements `T`.
fn foo() -> impl T { // ok!
    S(42)
}
```

If there are multiple types involved, the only way you care to interact with them is through the trait's interface, and having to rely on dynamic dispatch is acceptable, then you can use trait objects with `Box` , or other container types like `Rc` or `Arc` :

```
struct O(&'static str);
impl T for O {
    fn bar(&self) {}
}

// This now returns a "trait object" and callers are only be able to access
// associated items from `T`.
fn foo(x: bool) -> Box<dyn T> { // ok!
    if x {
        Box::new(S(42))
    } else {
        Box::new(O("val"))
    }
}
```

Finally, if you wish to still be able to access the original type, you can create a new `enum` with a variant for each type:

```
enum E {
    S(S),
    O(O),
}

// The caller can access the original types directly, but it needs to match
on
// the returned `enum E`.
fn foo(x: bool) -> E {
    if x {
        E::S(S(42))
    } else {
        E::O(O("val"))
    }
}
```

You can even implement the `trait` on the returned `enum` so the callers *don't* have to match on the returned value to invoke the associated items:

```
impl T for E {
    fn bar(&self) {
        match self {
            E::S(s) => s.bar(),
            E::O(o) => o.bar(),
        }
    }
}
```

If you decide to use trait objects, be aware that these rely on dynamic dispatch, which has performance implications, as the compiler needs to emit code that will figure out which method to call *at runtime* instead of during compilation. Using trait objects we are trading flexibility for performance.

Error code E0747

Generic arguments were not provided in the same order as the corresponding generic parameters are declared.

Erroneous code example:

```
struct S<'a, T>(&'a T);

type X = S<(), 'static>; // error: the type argument is provided before the
                         // lifetime argument
```

The argument order should be changed to match the parameter declaration order, as in the following:

```
struct S<'a, T>(&'a T);

type X = S<'static, ()>; // ok
```

Error code E0748

A raw string isn't correctly terminated because the trailing `#` count doesn't match its leading `#` count.

Erroneous code example:

```
let dolphins = r##"Dolphins!"#; // error!
```

To terminate a raw string, you have to have the same number of `#` at the end as at the beginning. Example:

```
let dolphins = r#"Dolphins!"#; // One `#` at the beginning, one at the end so
                               // all good!
```

Error code E0749

An item was added on a negative impl.

Erroneous code example:

```
trait MyTrait {
    type Foo;
}

impl !MyTrait for u32 {
    type Foo = i32; // error!
}
```

Negative impls are not allowed to have any items. Negative impls declare that a trait is **not** implemented (and never will be) and hence there is no need to specify the values for trait methods or other items.

One way to fix this is to remove the items in negative impls:

```
trait MyTrait {
    type Foo;
}

impl !MyTrait for u32 {}
```

Error code E0750

A negative impl was made default impl.

Erroneous code example:

```
trait MyTrait {
    type Foo;
}

default impl !MyTrait for u32 {} // error!
```

Negative impls cannot be default impls. A default impl supplies default values for the items within to be used by other impls, whereas a negative impl declares that there are no other impls. Combining it does not make sense.

Error code E0751

There are both a positive and negative trait implementation for the same type.

Erroneous code example:

```
trait MyTrait {}
impl MyTrait for i32 { }
impl !MyTrait for i32 { } // error!
```

Negative implementations are a promise that the trait will never be implemented for the given types. Therefore, both cannot exists at the same time.

Error code E0752

The entry point of the program was marked as `async`.

Erroneous code example:

```
async fn main() -> Result<(), ()> { // error!
    Ok(())
}
```

`fn main()` or the specified start function is not allowed to be `async`. Not having a correct async runtime library setup may cause this error. To fix it, declare the entry point without `async`:

```
fn main() -> Result<(), ()> { // ok!
    Ok(())
}
```

Error code E0753

An inner doc comment was used in an invalid context.

Erroneous code example:

```
fn foo() {}
//! foo
// ^ error!
fn main() {}
```

Inner document can only be used before items. For example:

```
//! A working comment applied to the module!
fn foo() {
    //! Another working comment!
}
fn main() {}
```

In case you want to document the item following the doc comment, you might want to use outer doc comment:

```
/// I am an outer doc comment
#[doc = "I am also an outer doc comment!"]
fn foo() {
    // ...
}
```

Error code E0754

A non-ASCII identifier was used in an invalid context.

Erroneous code examples:

```
mod řųśť; // error!

#[no_mangle]
fn řųśť() {} // error!

fn main() {}
```

Non-ASCII can be used as module names if it is inlined or if a `#[path]` attribute is specified. For example:

```
mod řųśť { // ok!
    const IS_GREAT: bool = true;
}

fn main() {}
```

Error code E0755

The `ffi_pure` attribute was used on a non-foreign function.

Erroneous code example:

```
#![feature(ffi_pure)]

#[ffi_pure] // error!
pub fn foo() {}
```

The `ffi_pure` attribute can only be used on foreign functions which do not have side effects or infinite loops:

```
#![feature(ffi_pure)]

extern "C" {
    #[ffi_pure] // ok!
    pub fn strlen(s: *const i8) -> isize;
}
```

You can find more information about it in the unstable Rust Book.

Error code E0756

The `ffi_const` attribute was used on something other than a foreign function declaration.

Erroneous code example:

```
#![feature(ffi_const)]

#[ffi_const] // error!
pub fn foo() {}
```

The `ffi_const` attribute can only be used on foreign function declarations which have no side effects except for their return value:

```
#![feature(ffi_const)]

extern "C" {
    #[ffi_const] // ok!
    pub fn strlen(s: *const i8) -> i32;
}
```

You can get more information about it in the unstable Rust Book.

Error code E0757

A function was given both the `ffi_const` and `ffi_pure` attributes.

Erroneous code example:

```
#![feature(ffi_const, ffi_pure)]

extern "C" {
    #[ffi_const]
    #[ffi_pure] // error: `#[ffi_const]` function cannot be `#[ffi_pure]`
    pub fn square(num: i32) -> i32;
}
```

As `ffi_const` provides stronger guarantees than `ffi_pure`, remove the `ffi_pure` attribute:

```
#![feature(ffi_const)]

extern "C" {
    #[ffi_const]
    pub fn square(num: i32) -> i32;
}
```

You can get more information about `const` and `pure` in the GCC documentation on Common Function Attributes. The unstable Rust Book has more information about `ffi_const` and `ffi_pure`.

Error code E0758

A multi-line (doc-)comment is unterminated.

Erroneous code example:

```
/* I am not terminated!
```

The same goes for doc comments:

```
/*! I am not terminated!
```

You need to end your multi-line comment with `*/` in order to fix this error:

```
/* I am terminated! */
/*! I am also terminated! */
```

Error code E0759

Note: this error code is no longer emitted by the compiler.

Return type involving a trait did not require `'static` lifetime.

Erroneous code examples:

```
use std::fmt::Debug;

fn foo(x: &i32) -> impl Debug { // error!
    x
}

fn bar(x: &i32) -> Box<dyn Debug> { // error!
    Box::new(x)
}
```

Add `'static` requirement to fix them:

```
fn foo(x: &'static i32) -> impl Debug + 'static { // ok!
    x
}

fn bar(x: &'static i32) -> Box<dyn Debug + 'static> { // ok!
    Box::new(x)
}
```

Both `dyn Trait` and `impl Trait` in return types have an implicit `'static` requirement, meaning that the value implementing them that is being returned has to be either a `'static` borrow or an owned value.

In order to change the requirement from `'static` to be a lifetime derived from its arguments, you can add an explicit bound, either to an anonymous lifetime `'_` or some appropriate named lifetime.

```
fn foo(x: &i32) -> impl Debug + '_ {
    x
}
fn bar(x: &i32) -> Box<dyn Debug + '_> {
    Box::new(x)
}
```

These are equivalent to the following explicit lifetime annotations:

```
fn foo<'a>(x: &'a i32) -> impl Debug + 'a {
    x
}
fn bar<'a>(x: &'a i32) -> Box<dyn Debug + 'a> {
    Box::new(x)
}
```

Error code E0760

Note: this error code is no longer emitted by the compiler.

`async fn`/`impl trait` return type cannot contain a projection or `Self` that references lifetimes from a parent scope.

Erroneous code example:

```
struct S<'a>(&'a i32);

impl<'a> S<'a> {
    async fn new(i: &'a i32) -> Self {
        S(&22)
    }
}
```

To fix this error we need to spell out `Self` to `S<'a>`:

```
struct S<'a>(&'a i32);

impl<'a> S<'a> {
    async fn new(i: &'a i32) -> S<'a> {
        S(&22)
    }
}
```

This will be allowed at some point in the future, but the implementation is not yet complete. See the issue-61949 for this limitation.

Error code E0761

Multiple candidate files were found for an out-of-line module.

Erroneous code example:

```
// file: ambiguous_module/mod.rs

fn foo() {}

// file: ambiguous_module.rs

fn foo() {}

// file: lib.rs

mod ambiguous_module; // error: file for module `ambiguous_module`
                      // found at both ambiguous_module.rs and
                      // ambiguous_module/mod.rs
```

Please remove this ambiguity by deleting/renaming one of the candidate files.

Error code E0762

A character literal wasn't ended with a quote.

Erroneous code example:

```
static C: char = '●; // error!
```

To fix this error, add the missing quote:

```
static C: char = '●'; // ok!
```

Error code E0763

A byte constant wasn't correctly ended.

Erroneous code example:

```
let c = b'a; // error!
```

To fix this error, add the missing quote:

```
let c = b'a'; // ok!
```

Error code E0764

A mutable reference was used in a constant.

Erroneous code example:

```
#![feature(const_mut_refs)]

fn main() {
    const OH_NO: &'static mut usize = &mut 1; // error!
}
```

Mutable references (`&mut`) can only be used in constant functions, not statics or constants. This limitation exists to prevent the creation of constants that have a mutable reference in their final value. If you had a constant of `&mut i32` type, you could modify the value through that reference, making the constant essentially mutable.

While there could be a more fine-grained scheme in the future that allows mutable references if they are not "leaked" to the final value, a more conservative approach was chosen for now. `const fn` do not have this problem, as the borrow checker will prevent the `const fn` from returning new mutable references.

Remember: you cannot use a function call inside a constant or static. However, you can totally use it in constant functions:

```
#![feature(const_mut_refs)]

const fn foo(x: usize) -> usize {
    let mut y = 1;
    let z = &mut y;
    *z += x;
    y
}

fn main() {
    const FOO: usize = foo(10); // ok!
}
```

Error code E0765

A double quote string (`"`) was not terminated.

Erroneous code example:

```
let s = "; // error!
```

To fix this error, add the missing double quote at the end of the string:

```
let s = ""; // ok!
```

Error code E0766

A double quote byte string (`b"`) was not terminated.

Erroneous code example:

```
let s = b"; // error!
```

To fix this error, add the missing double quote at the end of the string:

```
let s = b""; // ok!
```

Error code E0767

An unreachable label was used.

Erroneous code example:

```
'a: loop {
    || {
        loop { break 'a } // error: use of unreachable label `'a`
    };
}
```

Ensure that the label is within scope. Labels are not reachable through functions, closures, async blocks or modules. Example:

```
'a: loop {
    break 'a; // ok!
}
```

Error code E0768

A number in a non-decimal base has no digits.

Erroneous code example:

```
let s: i32 = 0b; // error!
```

To fix this error, add the missing digits:

```
let s: i32 = 0b1; // ok!
```

Error code E0769

A tuple struct or tuple variant was used in a pattern as if it were a struct or struct variant.

Erroneous code example:

```
enum E {
    A(i32),
}

let e = E::A(42);

match e {
    E::A { number } => { // error!
        println!("{}", number);
    }
}
```

To fix this error, you can use the tuple pattern:

```
match e {
    E::A(number) => { // ok!
        println!("{}", number);
    }
}
```

Alternatively, you can also use the struct pattern by using the correct field names and binding them to new identifiers:

```
match e {
    E::A { 0: number } => { // ok!
        println!("{}", number);
    }
}
```

Error code E0770

The type of a const parameter references other generic parameters.

Erroneous code example:

```
fn foo<T, const N: T>() {} // error!
```

To fix this error, use a concrete type for the const parameter:

```
fn foo<T, const N: usize>() {}
```

Error code E0771

Note: this error code is no longer emitted by the compiler

A non- `'static` lifetime was used in a const generic. This is currently not allowed.

Erroneous code example:

```
#![feature(adt_const_params)]

fn function_with_str<'a, const STRING: &'a str>() {} // error!
```

To fix this issue, the lifetime in the const generic need to be changed to `'static`:

```
#![feature(adt_const_params)]

fn function_with_str<const STRING: &'static str>() {} // ok!
```

For more information, see GitHub issue #74052.

Error code E0772

Note: this error code is no longer emitted by the compiler.

A trait object has some specific lifetime `'1`, but it was used in a way that requires it to have a `'static` lifetime.

Example of erroneous code:

```
trait BooleanLike {}
trait Person {}

impl BooleanLike for bool {}

impl dyn Person {
    fn is_cool(&self) -> bool {
        // hey you, you're pretty cool
        true
    }
}

fn get_is_cool<'p>(person: &'p dyn Person) -> impl BooleanLike {
    // error: `person` has an anonymous lifetime `'p` but calling
    //        `print_cool_fn` introduces an implicit `'static` lifetime
    //        requirement
    person.is_cool()
}
```

The trait object `person` in the function `get_is_cool`, while already being behind a reference with lifetime `'p`, also has it's own implicit lifetime, `'2`.

Lifetime `'2` represents the data the trait object might hold inside, for example:

```
trait MyTrait {}

struct MyStruct<'a>(&'a i32);

impl<'a> MyTrait for MyStruct<'a> {}
```

With this scenario, if a trait object of `dyn MyTrait + '2` was made from `MyStruct<'a>`, `'a` must live as long, if not longer than `'2`. This allows the trait object's internal data to be accessed safely from any trait methods. This rule also goes for any lifetime any struct made into a trait object may have.

In the implementation for `dyn Person`, the `'2` lifetime representing the internal data was omitted, meaning that the compiler inferred the lifetime `'static`. As a result, the implementation's `is_cool` is inferred by the compiler to look like this:

```
fn is_cool<'a>(self: &'a (dyn Person + 'static)) -> bool {unimplemented!()}
```

While the `get_is_cool` function is inferred to look like this:

```
fn get_is_cool<'p, R: BooleanLike>(person: &'p (dyn Person + 'p)) -> R {
    unimplemented!()
}
```

Which brings us to the core of the problem; the assignment of type `&'_ (dyn Person + '_)` to type `&'_ (dyn Person + 'static)` is impossible.

Fixing it is as simple as being generic over lifetime `'2`, as to prevent the compiler from inferring it as `'static`:

```
impl<'d> dyn Person + 'd {/* ... */}

// This works too, and is more elegant:
//impl dyn Person + '_ {/* ... */}
```

See the [Rust Reference on Trait Object Lifetime Bounds][trait-objects] for more information on trait object lifetimes.

Error code E0773

A builtin-macro was defined more than once.

Erroneous code example:

```
#![feature(decl_macro)]
#![feature(rustc_attrs)]
#![allow(internal_features)]

#[rustc_builtin_macro]
pub macro test($item:item) {
    /* compiler built-in */
}

mod inner {
    #[rustc_builtin_macro]
    pub macro test($item:item) {
        /* compiler built-in */
    }
}
```

To fix the issue, remove the duplicate declaration:

```
#![feature(decl_macro)]
#![feature(rustc_attrs)]
#![allow(internal_features)]

#[rustc_builtin_macro]
pub macro test($item:item) {
    /* compiler built-in */
}
```

In very rare edge cases, this may happen when loading `core` or `std` twice, once with `check` metadata and once with `build` metadata. For more information, see #75176.

Error code E0774

`derive` was applied on something which is not a struct, a union or an enum.

Erroneous code example:

```
trait Foo {
    #[derive(Clone)] // error!
    type Bar;
}
```

As said above, the `derive` attribute is only allowed on structs, unions or enums:

```
#[derive(Clone)] // ok!
struct Bar {
    field: u32,
}
```

You can find more information about `derive` in the Rust Book.

Error code E0775

`#[cmse_nonsecure_entry]` is only valid for targets with the TrustZone-M extension.

Erroneous code example:

```
#![feature(cmse_nonsecure_entry)]

#[cmse_nonsecure_entry]
pub extern "C" fn entry_function() {}
```

To fix this error, compile your code for a Rust target that supports the TrustZone-M extension. The current possible targets are:

- `thumbv8m.main-none-eabi`
- `thumbv8m.main-none-eabihf`
- `thumbv8m.base-none-eabi`

Error code E0776

`#[cmse_nonsecure_entry]` functions require a C ABI

Erroneous code example:

```
#![feature(cmse_nonsecure_entry)]

#[no_mangle]
#[cmse_nonsecure_entry]
pub fn entry_function(input: Vec<u32>) {}
```

To fix this error, declare your entry function with a C ABI, using `extern "C"`.

Error code E0777

A literal value was used inside `#[derive]`.

Erroneous code example:

```
#[derive("Clone")] // error!
struct Foo;
```

Only paths to traits are allowed as argument inside `#[derive]`. You can find more information about the `#[derive]` attribute in the Rust Book.

```
#[derive(Clone)] // ok!
struct Foo;
```

Error code E0778

The `instruction_set` attribute was malformed.

Erroneous code example:

```
#![feature(isa_attribute)]

#[instruction_set()] // error: expected one argument
pub fn something() {}
fn main() {}
```

The parenthesized `instruction_set` attribute requires the parameter to be specified:

```
#![feature(isa_attribute)]

#[cfg_attr(target_arch="arm", instruction_set(arm::a32))]
fn something() {}
```

or:

```
#![feature(isa_attribute)]

#[cfg_attr(target_arch="arm", instruction_set(arm::t32))]
fn something() {}
```

For more information see the `instruction_set` attribute section of the Reference.

Error code E0779

An unknown argument was given to the `instruction_set` attribute.

Erroneous code example:

```
#![feature(isa_attribute)]

#[instruction_set(intel::x64)] // error: invalid argument
pub fn something() {}
fn main() {}
```

The `instruction_set` attribute only supports two arguments currently:

- arm::a32
- arm::t32

All other arguments given to the `instruction_set` attribute will return this error. Example:

```
#![feature(isa_attribute)]

#[cfg_attr(target_arch="arm", instruction_set(arm::a32))] // ok!
pub fn something() {}
fn main() {}
```

For more information see the `instruction_set` attribute section of the Reference.

Error code E0780

Cannot use `doc(inline)` with anonymous imports

Erroneous code example:

```
#[doc(inline)] // error: invalid doc argument
pub use foo::Foo as _;
```

Anonymous imports are always rendered with `#[doc(no_inline)]`. To fix this error, remove the `#[doc(inline)]` attribute.

Example:

```
pub use foo::Foo as _;
```

Error code E0781

The `C-cmse-nonsecure-call` ABI can only be used with function pointers.

Erroneous code example:

```
#![feature(abi_c_cmse_nonsecure_call)]

pub extern "C-cmse-nonsecure-call" fn test() {}
```

The `C-cmse-nonsecure-call` ABI should be used by casting function pointers to specific addresses.

Error code E0782

Trait objects must include the `dyn` keyword.

Erroneous code example:

```
trait Foo {}
fn test(arg: Box<Foo>) {} // error!
```

Trait objects are a way to call methods on types that are not known until runtime but conform to some trait.

Trait objects should be formed with `Box<dyn Foo>`, but in the code above `dyn` is left off.

This makes it harder to see that `arg` is a trait object and not a simply a heap allocated type called `Foo`.

To fix this issue, add `dyn` before the trait name.

```
trait Foo {}
fn test(arg: Box<dyn Foo>) {} // ok!
```

This used to be allowed before edition 2021, but is now an error.

Error code E0783

The range pattern `...` is no longer allowed.

Erroneous code example:

```
match 2u8 {
    0...9 => println!("Got a number less than 10"), // error!
    _ => println!("Got a number 10 or more"),
}
```

Older Rust code using previous editions allowed `...` to stand for exclusive ranges which are now signified using `..=`.

To make this code compile replace the `...` with `..=`.

```
match 2u8 {
    0..=9 => println!("Got a number less than 10"), // ok!
    _ => println!("Got a number 10 or more"),
}
```

Error code E0784

A union expression does not have exactly one field.

Erroneous code example:

```
union Bird {
    pigeon: u8,
    turtledove: u16,
}

let bird = Bird {}; // error
let bird = Bird { pigeon: 0, turtledove: 1 }; // error
```

The key property of unions is that all fields of a union share common storage. As a result, writes to one field of a union can overwrite its other fields, and size of a union is determined by the size of its largest field.

You can find more information about the union types in the Rust reference.

Working example:

```
union Bird {
    pigeon: u8,
    turtledove: u16,
}

let bird = Bird { pigeon: 0 }; // OK
```

Error code E0785

An inherent `impl` was written on a dyn auto trait.

Erroneous code example:

```
#![feature(auto_traits)]

auto trait AutoTrait {}

impl dyn AutoTrait {}
```

Dyn objects allow any number of auto traits, plus at most one non-auto trait. The non-auto trait becomes the "principal trait".

When checking if an impl on a dyn trait is coherent, the principal trait is normally the only one considered. Since the erroneous code has no principal trait, it cannot be implemented at all.

Working example:

```
#![feature(auto_traits)]

trait PrincipalTrait {}

auto trait AutoTrait {}

impl dyn PrincipalTrait + AutoTrait + Send {}
```

Error code E0786

A metadata file was invalid.

Erroneous code example:

```
use ::foo; // error: found invalid metadata files for crate `foo`
```

When loading crates, each crate must have a valid metadata file. Invalid files could be caused by filesystem corruption, an IO error while reading the file, or (rarely) a bug in the compiler itself.

Consider deleting the file and recreating it, or reporting a bug against the compiler.

Error code E0787

An unsupported naked function definition.

Erroneous code example:

```
#![feature(naked_functions)]

#[naked]
pub extern "C" fn f() -> u32 {
    42
}
```

The naked functions must be defined using a single inline assembly block.

The execution must never fall through past the end of the assembly code so the block must use `noreturn` option. The asm block can also use `att_syntax` and `raw` options, but others options are not allowed.

The asm block must not contain any operands other than `const` and `sym` .

Additional information

For more information, please see RFC 2972.

Error code E0788

A `#[coverage]` attribute was applied to something which does not show up in code coverage, or is too granular to be excluded from the coverage report.

For now, this attribute can only be applied to function, method, and closure definitions. In the future, it may be added to statements, blocks, and expressions, and for the time being, using this attribute in those places will just emit an `unused_attributes` lint instead of this error.

Example of erroneous code:

```
#[coverage(off)]
struct Foo;

#[coverage(on)]
const FOO: Foo = Foo;
```

`#[coverage(off)]` tells the compiler to not generate coverage instrumentation for a piece of code when the `-C instrument-coverage` flag is passed. Things like structs and consts are not coverable code, and thus cannot do anything with this attribute.

If you wish to apply this attribute to all methods in an impl or module, manually annotate each method; it is not possible to annotate the entire impl with a `#[coverage]` attribute.

Error code E0789

This error code is internal to the compiler and will not be emitted with normal Rust code.

The internal `rustc_allowed_through_unstable_modules` attribute must be used on an item with a `stable` attribute.

Erroneous code example:

```
// NOTE: both of these attributes are perma-unstable and should *never* be
//       used outside of the compiler and standard library.
#![feature(rustc_attrs)]
#![feature(staged_api)]
#![allow(internal_features)]

#![unstable(feature = "foo_module", reason = "...", issue = "123")]

#[rustc_allowed_through_unstable_modules]
// #[stable(feature = "foo", since = "1.0")]
struct Foo;
// ^^^ error: `rustc_allowed_through_unstable_modules` attribute must be
//            paired with a `stable` attribute
```

Typically when an item is marked with a `stable` attribute, the modules that enclose the item must also be marked with `stable` attributes, otherwise the item becomes *de facto* unstable. `#[rustc_allowed_through_unstable_modules]` is a workaround which allows an item to "escape" its unstable parent modules. This error occurs when an item is marked with `#[rustc_allowed_through_unstable_modules]` but no supplementary `stable` attribute exists. See #99288 for an example of `#[rustc_allowed_through_unstable_modules]` in use.

Error code E0790

You need to specify a specific implementation of the trait in order to call the method.

Erroneous code example:

```
trait Coroutine {
    fn create() -> u32;
}

struct Impl;

impl Coroutine for Impl {
    fn create() -> u32 { 1 }
}

struct AnotherImpl;

impl Coroutine for AnotherImpl {
    fn create() -> u32 { 2 }
}

let cont: u32 = Coroutine::create();
// error, impossible to choose one of Coroutine trait implementation
// Should it be Impl or AnotherImpl, maybe something else?
```

This error can be solved by adding type annotations that provide the missing information to the compiler. In this case, the solution is to use a concrete type:

```
trait Coroutine {
    fn create() -> u32;
}

struct AnotherImpl;

impl Coroutine for AnotherImpl {
    fn create() -> u32 { 2 }
}

let gen1 = AnotherImpl::create();

// if there are multiple methods with same name (different traits)
let gen2 = <AnotherImpl as Coroutine>::create();
```

Error code E0791

Static variables with the `#[linkage]` attribute within external blocks must have one of the following types, which are equivalent to a nullable pointer in C:

- `*mut T` or `*const T`, where `T` may be any type.
- An enumerator type with no `#[repr]` attribute and with two variants, where one of the variants has no fields, and the other has a single field of one of the following non-nullable types:
  - Reference type
  - Function pointer type

  The variants can appear in either order.

For example, the following declaration is invalid:

```
#![feature(linkage)]

extern "C" {
    #[linkage = "extern_weak"]
    static foo: i8;
}
```

The following declarations are valid:

```
#![feature(linkage)]

extern "C" {
    #[linkage = "extern_weak"]
    static foo: Option<unsafe extern "C" fn()>;

    #[linkage = "extern_weak"]
    static bar: Option<&'static i8>;

    #[linkage = "extern_weak"]
    static baz: *mut i8;
}
```

Error code E0792

A type alias impl trait can only have its hidden type assigned when used fully generically (and within their defining scope). This means

```
#![feature(type_alias_impl_trait)]

type Foo<T> = impl std::fmt::Debug;

fn foo() -> Foo<u32> {
    5u32
}
```

is not accepted. If it were accepted, one could create unsound situations like

```
#![feature(type_alias_impl_trait)]

type Foo<T> = impl Default;

fn foo() -> Foo<u32> {
    5u32
}

fn main() {
    let x = Foo::<&'static mut String>::default();
}
```

Instead you need to make the function generic:

```
#![feature(type_alias_impl_trait)]

type Foo<T> = impl std::fmt::Debug;

fn foo<U>() -> Foo<U> {
    5u32
}
```

This means that no matter the generic parameter to `foo`, the hidden type will always be `u32`. If you want to link the generic parameter to the hidden type, you can do that, too:

```
#![feature(type_alias_impl_trait)]

use std::fmt::Debug;

type Foo<T: Debug> = impl Debug;

fn foo<U: Debug>() -> Foo<U> {
    Vec::<U>::new()
}
```

Error code E0793

An unaligned references to a field of a packed struct got created.

Erroneous code example:

```
#[repr(packed)]
pub struct Foo {
    field1: u64,
    field2: u8,
}

unsafe {
    let foo = Foo { field1: 0, field2: 0 };
    // Accessing the field directly is fine.
    let val = foo.field1;
    // A reference to a packed field causes a error.
    let val = &foo.field1; // ERROR
    // An implicit `&` is added in format strings, causing the same error.
    println!("{}", foo.field1); // ERROR
}
```

Creating a reference to an insufficiently aligned packed field is undefined behavior and therefore disallowed. Using an `unsafe` block does not change anything about this. Instead, the code should do a copy of the data in the packed field or use raw pointers and unaligned accesses.

```
#[repr(packed)]
pub struct Foo {
    field1: u64,
    field2: u8,
}

unsafe {
    let foo = Foo { field1: 0, field2: 0 };

    // Instead of a reference, we can create a raw pointer...
    let ptr = std::ptr::addr_of!(foo.field1);
    // ... and then (crucially!) access it in an explicitly unaligned way.
    let val = unsafe { ptr.read_unaligned() };
    // This would *NOT* be correct:
    // let val = unsafe { *ptr }; // Undefined Behavior due to unaligned
load!

    // For formatting, we can create a copy to avoid the direct reference.
    let copy = foo.field1;
    println!("{}", copy);
    // Creating a copy can be written in a single line with curly braces.
    // (This is equivalent to the two lines above.)
    println!("{}", { foo.field1 });
}
```

Additional information

Note that this error is specifically about *references* to packed fields. Direct by-value access of those fields is fine, since then the compiler has enough information to generate the correct kind of access.

See issue #82523 for more information.

Error code E0794

A lifetime parameter of a function definition is called *late-bound* if it both:

1. appears in an argument type
2. does not appear in a generic type constraint

You cannot specify lifetime arguments for late-bound lifetime parameters.

Erroneous code example:

```
fn foo<'a>(x: &'a str) -> &'a str { x }
let _ = foo::<'static>;
```

The type of a concrete instance of a generic function is universally quantified over late-bound lifetime parameters. This is because we want the function to work for any lifetime substituted for the late-bound lifetime parameter, no matter where the function is called. Consequently, it doesn't make sense to specify arguments for late-bound lifetime parameters, since they are not resolved until the function's call site(s).

To fix the issue, remove the specified lifetime:

```
fn foo<'a>(x: &'a str) -> &'a str { x }
let _ = foo;
```

Additional information

Lifetime parameters that are not late-bound are called *early-bound*. Confusion may arise from the fact that late-bound and early-bound lifetime parameters are declared the same way in function definitions. When referring to a function pointer type, universal quantification over late-bound lifetime parameters can be made explicit:

```
trait BarTrait<'a> {}

struct Bar<'a> {
    s: &'a str
}

impl<'a> BarTrait<'a> for Bar<'a> {}

fn bar<'a, 'b, T>(x: &'a str, _t: T) -> &'a str
where T: BarTrait<'b>
{
    x
}

let bar_fn: for<'a> fn(&'a str, Bar<'static>) -> &'a str = bar; // OK
let bar_fn2 = bar::<'static, Bar>; // Not allowed
let bar_fn3 = bar::<Bar>; // OK
```

In the definition of `bar`, the lifetime parameter `'a` is late-bound, while `'b` is early-bound. This is reflected in the type annotation for `bar_fn`, where `'a` is universally quantified and `'b` is substituted by a specific lifetime. It is not allowed to explicitly specify early-bound lifetime arguments when late-bound lifetime parameters are present (as for `bar_fn2`, see issue #42868), although the types that are constrained by early-bound parameters can be specified (as for `bar_fn3`).

Error code E0795

Invalid argument for the `offset_of!` macro.

Erroneous code example:

```
#![feature(offset_of, offset_of_enum)]

let x = std::mem::offset_of!(Option<u8>, Some);
```

The `offset_of!` macro gives the offset of a field within a type. It can navigate through enum variants, but the final component of its second argument must be a field and not a variant.

The offset of the contained `u8` in the `Option<u8>` can be found by specifying the field name `0`:

```
#![feature(offset_of, offset_of_enum)]

let x: usize = std::mem::offset_of!(Option<u8>, Some.0);
```

The discriminant of an enumeration may be read with `core::mem::discriminant`, but this is not always a value physically present within the enum.

Further information about enum layout may be found at https://rust-lang.github.io/unsafe-code-guidelines/layout/enums.html.