C

Derivable Traits

In various places in the book, we’ve discussed the derive attribute, which you can apply to a struct or enum definition. The derive attribute generates code that will implement a trait with its own default implementation on the type you’ve annotated with the derive syntax.

In this appendix, we provide a reference of all the traits in the standard library that you can use with derive. Each section covers:

What operators and methods deriving this trait will enable

What the implementation of the trait provided by derive does

What implementing the trait signifies about the type

The conditions in which you’re allowed or not allowed to implement the trait

Examples of operations that require the trait

If you want different behavior from that provided by the derive attribute, consult the standard library documentation for each trait for details on how to manually implement them.

The traits listed here are the only ones defined by the standard library that can be implemented on your types using derive. Other traits defined in the standard library don’t have sensible default behavior, so it’s up to you to implement them in the way that makes sense for what you’re trying to accomplish.

An example of a trait that can’t be derived is Display, which handles formatting for end users. You should always consider the appropriate way to display a type to an end user. What parts of the type should an end user be allowed to see? What parts would they find relevant? What format of the data would be most relevant to them? The Rust compiler doesn’t have this insight, so it can’t provide appropriate default behavior for you.

The list of derivable traits provided in this appendix is not comprehensive: libraries can implement derive for their own traits, making the list of traits you can use derive with truly open ended. Implementing derive involves using a procedural macro, which is covered in “Macros” on page 449.

Debug for Programmer Output

The Debug trait enables debug formatting in format strings, which you indicate by adding :? within {} placeholders.

The Debug trait allows you to print instances of a type for debugging purposes, so you and other programmers using your type can inspect an instance at a particular point in a program’s execution.

The Debug trait is required, for example, in the use of the assert\_eq! macro. This macro prints the values of instances given as arguments if the equality assertion fails so programmers can see why the two instances weren’t equal.

PartialEq and Eq for Equality Comparisons

The PartialEq trait allows you to compare instances of a type to check for equality and enables use of the == and != operators.

Deriving PartialEq implements the eq method. When PartialEq is derived on structs, two instances are equal only if all fields are equal, and the instances are not equal if any fields are not equal. When derived on enums, each variant is equal to itself and not equal to the other variants.

The PartialEq trait is required, for example, with the use of the assert\_eq! macro, which needs to be able to compare two instances of a type for equality.

The Eq trait has no methods. Its purpose is to signal that for every value of the annotated type, the value is equal to itself. The Eq trait can only be applied to types that also implement PartialEq, although not all types that implement PartialEq can implement Eq. One example of this is floating-point number types: the implementation of floating-point numbers states that two instances of the not-a-number (NaN) value are not equal to each other.

An example of when Eq is required is for keys in a HashMap<K, V> so that the HashMap<K, V> can tell whether two keys are the same.

PartialOrd and Ord for Ordering Comparisons

The PartialOrd trait allows you to compare instances of a type for sorting purposes. A type that implements PartialOrd can be used with the <, >, <=, and >= operators. You can only apply the PartialOrd trait to types that also implement PartialEq.

Deriving PartialOrd implements the partial\_cmp method, which returns an   
Option<Ordering> that will be None when the values given don’t produce   
an ordering. An example of a value that doesn’t produce an ordering, even though most values of that type can be compared, is the NaN floating point value. Calling partial\_cmp with any floating-point number and the NaN floating-point value will return None.

When derived on structs, PartialOrd compares two instances by comparing the value in each field in the order in which the fields appear in the struct definition. When derived on enums, variants of the enum declared earlier in the enum definition are considered less than the variants listed later.

The PartialOrd trait is required, for example, for the gen\_range method from the rand crate that generates a random value in the range specified by a range expression.

The Ord trait allows you to know that for any two values of the annotated type, a valid ordering will exist. The Ord trait implements the cmp method, which returns an Ordering rather than an Option<Ordering> because a valid ordering will always be possible. You can only apply the Ord trait to types that also implement PartialOrd and Eq (and Eq requires PartialEq). When derived on structs and enums, cmp behaves the same way as the derived implementation for partial\_cmp does with PartialOrd.

An example of when Ord is required is when storing values in a BTreeSet<T>, a data structure that stores data based on the sort order of the values.

Clone and Copy for Duplicating Values

The Clone trait allows you to explicitly create a deep copy of a value, and the duplication process might involve running arbitrary code and copying heap data. See “Variables and Data Interacting with Clone” on page 67 for more information on Clone.

Deriving Clone implements the clone method, which when implemented for the whole type, calls clone on each of the parts of the type. This means all the fields or values in the type must also implement Clone to derive Clone.

An example of when Clone is required is when calling the to\_vec method on a slice. The slice doesn’t own the type instances it contains, but the vector returned from to\_vec will need to own its instances, so to\_vec calls clone on each item. Thus the type stored in the slice must implement Clone.

The Copy trait allows you to duplicate a value by only copying bits stored on the stack; no arbitrary code is necessary. See “Stack-Only Data: Copy” on page 68 for more information on Copy.

The Copy trait doesn’t define any methods to prevent programmers from overloading those methods and violating the assumption that no arbitrary code is being run. That way, all programmers can assume that copying a value will be very fast.

You can derive Copy on any type whose parts all implement Copy. A type that implements Copy must also implement Clone because a type that implements Copy has a trivial implementation of Clone that performs the same task as Copy.

The Copy trait is rarely required; types that implement Copy have optimizations available, meaning you don’t have to call clone, which makes the code more concise.

Everything possible with Copy you can also accomplish with Clone, but the code might be slower or have to use clone in places.

Hash for Mapping a Value to a Value of Fixed Size

The Hash trait allows you to take an instance of a type of arbitrary size and map that instance to a value of fixed size using a hash function. Deriving Hash implements the hash method. The derived implementation of the hash method combines the result of calling hash on each of the parts of the type, meaning all fields or values must also implement Hash to derive Hash.

An example of when Hash is required is in storing keys in a HashMap<K, V> to store data efficiently.

Default for Default Values

The Default trait allows you to create a default value for a type. Deriving Default implements the default function. The derived implementation of   
the default function calls the default function on each part of the type, meaning all fields or values in the type must also implement Default to derive Default.

The Default::default function is commonly used in combination with the struct update syntax discussed in “Creating Instances from Other Instances with Struct Update Syntax” on page 88. You can customize a few fields of a struct and then set and use a default value for the rest of the fields by using ..Default::default().

The Default trait is required when you use the method unwrap\_or\_default on Option<T> instances, for example. If the Option<T> is None, the method unwrap\_or\_default will return the result of Default::default for the type T stored in the Option<T>.