Lints

In software, a "lint" is a tool used to help improve your source code. The Rust compiler contains a number of lints, and when it compiles your code, it will also run the lints. These lints may produce a warning, an error, or nothing at all, depending on how you've configured things.

Here's a small example:

```
$ cat main.rs
fn main() {
    let x = 5;
}
$ rustc main.rs
warning: unused variable: `x`
 --> main.rs:2:9
  |
2 |     let x = 5;
  |         ^
  |
  = note: `#[warn(unused_variables)]` on by default
  = note: to avoid this warning, consider using `_x` instead
```

This is the `unused_variables` lint, and it tells you that you've introduced a variable that you don't use in your code. That's not *wrong*, so it's not an error, but it might be a bug, so you get a warning.

Future-incompatible lints

Sometimes the compiler needs to be changed to fix an issue that can cause existing code to stop compiling. "Future-incompatible" lints are issued in these cases to give users of Rust a smooth transition to the new behavior. Initially, the compiler will continue to accept the problematic code and issue a warning. The warning has a description of the problem, a notice that this will become an error in the future, and a link to a tracking issue that provides detailed information and an opportunity for feedback. This gives users some time to fix the code to accommodate the change. After some time, the warning may become an error.

The following is an example of what a future-incompatible looks like:

```
warning: borrow of packed field is unsafe and requires unsafe function or block
(error E0133)
  --> lint_example.rs:11:13
   |
11 |     let y = &x.data.0;
   |             ^^^^^^^^^
   |
   = note: `#[warn(safe_packed_borrows)]` on by default
   = warning: this was previously accepted by the compiler but is being phased out;
it will become a hard error in a future release!
   = note: for more information, see issue #46043 <https://github.com/rust-
lang/rust/issues/46043>
   = note: fields of packed structs might be misaligned: dereferencing a misaligned
pointer or even just creating a misaligned reference is undefined behavior
```

For more information about the process and policy of future-incompatible changes, see RFC 1589.