Common tools for writing lints

You may need following tooltips to catch up with common operations.

Useful Rustc dev guide links:

- Stages of compilation
- Diagnostic items
- Type checking
- Ty module

Retrieving the type of an expression

Sometimes you may want to retrieve the type `Ty` of an expression `Expr`, for example to answer following questions:

- which type does this expression correspond to (using its `TyKind`)?
- is it a sized type?
- is it a primitive type?
- does it implement a trait?

This operation is performed using the `expr_ty()` method from the `TypeckResults` struct, that gives you access to the underlying structure `Ty`.

Example of use:

```
impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        // Get type of `expr`
        let ty = cx.typeck_results().expr_ty(expr);
        // Match its kind to enter its type
        match ty.kind {
            ty::Adt(adt_def, _) if adt_def.is_struct() => println!("Our `expr` is a
struct!"),
            _ => ()
        }
    }
}
```

Similarly in `TypeckResults` methods, you have the `pat_ty()` method to retrieve a type from a pattern.

Two noticeable items here:

- `cx` is the lint context `LateContext`. The two most useful data structures in this context are `tcx` and the `TypeckResults` returned by `LateContext::typeck_results`, allowing us to jump to type

definitions and other compilation stages such as HIR.

- `typeck_results`'s return value is `TypeckResults` and is created by type checking step, it includes useful information such as types of expressions, ways to resolve methods and so on.

Checking if an expr is calling a specific method

Starting with an `expr`, you can check whether it is calling a specific method `some_method`:

```
impl<'tcx> LateLintPass<'tcx> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx hir::Expr<'_>) {
        if_chain! {
            // Check our expr is calling a method
            if let hir::ExprKind::MethodCall(path, _, [self_arg, ..]) = &expr.kind;
            // Check the name of this method is `some_method`
            if path.ident.name == sym!(some_method);
            // Optionally, check the type of the self argument.
            // - See "Checking for a specific type"
            then {
                // ...
            }
        }
    }
}
```

Checking for a specific type

There are three ways to check if an expression type is a specific type we want to check for. All of these methods only check for the base type, generic arguments have to be checked separately.

```
use clippy_utils::ty::{is_type_diagnostic_item, is_type_lang_item};
use clippy_utils::{paths, match_def_path};
use rustc_span::symbol::sym;
use rustc_hir::LangItem;

impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        // Getting the expression type
        let ty = cx.typeck_results().expr_ty(expr);

        // 1. Using diagnostic items
        // The last argument is the diagnostic item to check for
        if is_type_diagnostic_item(cx, ty, sym::Option) {
            // The type is an `Option`
        }

        // 2. Using lang items
        if is_type_lang_item(cx, ty, LangItem::RangeFull) {
            // The type is a full range like `.drain(..)`
        }
```

```
        // 3. Using the type path
        // This method should be avoided if possible
        if match_def_path(cx, def_id, &paths::RESULT) {
            // The type is a `core::result::Result`
        }
    }
}
```

Prefer using diagnostic items and lang items where possible.

Checking if a type implements a specific trait

There are three ways to do this, depending on if the target trait has a diagnostic item, lang item or neither.

```
use clippy_utils::{implements_trait, is_trait_method, match_trait_method, paths};
use rustc_span::symbol::sym;

impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        // 1. Using diagnostic items with the expression
        // we use `is_trait_method` function from Clippy's utils
        if is_trait_method(cx, expr, sym::Iterator) {
            // method call in `expr` belongs to `Iterator` trait
        }

        // 2. Using lang items with the expression type
        let ty = cx.typeck_results().expr_ty(expr);
        if cx.tcx.lang_items()
            // we are looking for the `DefId` of `Drop` trait in lang items
            .drop_trait()
            // then we use it with our type `ty` by calling `implements_trait` from
Clippy's utils
            .map_or(false, |id| implements_trait(cx, ty, id, &[])) {
                // `expr` implements `Drop` trait
            }

        // 3. Using the type path with the expression
        // we use `match_trait_method` function from Clippy's utils
        // (This method should be avoided if possible)
        if match_trait_method(cx, expr, &paths::INTO) {
            // `expr` implements `Into` trait
        }
    }
}
```

> *Prefer using diagnostic and lang items, if the target trait has one.*

We access lang items through the type context `tcx`. `tcx` is of type `TyCtxt` and is defined in the `rustc_middle` crate. A list of defined paths for Clippy can be found in paths.rs

Checking if a type defines a specific method

To check if our type defines a method called `some_method`:

```
use clippy_utils::{is_type_diagnostic_item, return_ty};

impl<'tcx> LateLintPass<'tcx> for MyTypeImpl {
    fn check_impl_item(&mut self, cx: &LateContext<'tcx>, impl_item: &'tcx
ImplItem<'_>) {
        if_chain! {
            // Check if item is a method/function
            if let ImplItemKind::Fn(ref signature, _) = impl_item.kind;
            // Check the method is named `some_method`
            if impl_item.ident.name == sym!(some_method);
            // We can also check it has a parameter `self`
            if signature.decl.implicit_self.has_implicit_self();
            // We can go further and even check if its return type is `String`
            if is_type_diagnostic_item(cx, return_ty(cx, impl_item.hir_id), sym!
(string_type));
            then {
                // ...
            }
        }
    }
}
```

Dealing with macros and expansions

Keep in mind that macros are already expanded and desugaring is already applied to the code representation that you are working with in Clippy. This unfortunately causes a lot of false positives because macro expansions are "invisible" unless you actively check for them. Generally speaking, code with macro expansions should just be ignored by Clippy because that code can be dynamic in ways that are difficult or impossible to see. Use the following functions to deal with macros:

- `span.from_expansion()`: detects if a span is from macro expansion or desugaring. Checking this is a common first step in a lint.

  ```
  if expr.span.from_expansion() {
      // just forget it
      return;
  }
  ```

- `span.ctxt()`: the span's context represents whether it is from expansion, and if so, which macro call expanded it. It is sometimes useful to check if the context of two spans are equal.

  ```
  // expands to `1 + 0`, but don't lint
  1 + mac!()
  ```

```
if left.span.ctxt() != right.span.ctxt() {
    // the coder most likely cannot modify this expression
    return;
}
```

Note: Code that is not from expansion is in the "root" context. So any spans where `from_expansion` returns `true` can be assumed to have the same context. And so just using `span.from_expansion()` is often good enough.

- `in_external_macro(span)` : detect if the given span is from a macro defined in a foreign crate. If you want the lint to work with macro-generated code, this is the next line of defense to avoid macros not defined in the current crate. It doesn't make sense to lint code that the coder can't change.

  You may want to use it for example to not start linting in macros from other crates

  ```
  #[macro_use]
  extern crate a_crate_with_macros;

  // `foo` is defined in `a_crate_with_macros`
  foo!("bar");

  // if we lint the `match` of `foo` call and test its span
  assert_eq!(in_external_macro(cx.sess(), match_span), true);
  ```

- `span.ctxt()` : the span's context represents whether it is from expansion, and if so, what expanded it

One thing `SpanContext` is useful for is to check if two spans are in the same context. For example, in `a == b` , `a` and `b` have the same context. In a `macro_rules!` with `a == $b` , `$b` is expanded to some expression with a different context from `a` .

```
macro_rules! m {
    ($a:expr, $b:expr) => {
        if $a.is_some() {
            $b;
        }
    }
}

let x: Option<u32> = Some(42);
m!(x, x.unwrap());

// These spans are not from the same context
// x.is_some() is from inside the macro
// x.unwrap() is from outside the macro
assert_eq!(x_is_some_span.ctxt(), x_unwrap_span.ctxt());
```