transparent_unions

The tracking issue for this feature is #60405

The `transparent_unions` feature allows you mark `unions` as `#[repr(transparent)]`. A `union` may be `#[repr(transparent)]` in exactly the same conditions in which a `struct` may be `#[repr(transparent)]` (generally, this means the `union` must have exactly one non-zero-sized field). Some concrete illustrations follow.

```
#![feature(transparent_unions)]

// This union has the same representation as `f32`.
#[repr(transparent)]
union SingleFieldUnion {
    field: f32,
}

// This union has the same representation as `usize`.
#[repr(transparent)]
union MultiFieldUnion {
    field: usize,
    nothing: (),
}
```

For consistency with transparent `structs`, `unions` must have exactly one non-zero-sized field. If all fields are zero-sized, the `union` must not be `#[repr(transparent)]`:

```
#![feature(transparent_unions)]

// This (non-transparent) union is already valid in stable Rust:
pub union GoodUnion {
    pub nothing: (),
}

// Error: transparent union needs exactly one non-zero-sized field, but has 0
// #[repr(transparent)]
// pub union BadUnion {
//     pub nothing: (),
// }
```

The one exception is if the `union` is generic over `T` and has a field of type `T`, it may be `#[repr(transparent)]` even if `T` is a zero-sized type:

```
#![feature(transparent_unions)]
```

```rust
// This union has the same representation as `T`.
#[repr(transparent)]
pub union GenericUnion<T: Copy> { // Unions with non-`Copy` fields are unstable.
    pub field: T,
    pub nothing: (),
}

// This is okay even though `()` is a zero-sized type.
pub const THIS_IS_OKAY: GenericUnion<()> = GenericUnion { field: () };
```

Like transarent `structs`, a transparent `union` of type `U` has the same layout, size, and ABI as its single non-ZST field. If it is generic over a type `T`, and all its fields are ZSTs except for exactly one field of type `T`, then it has the same layout and ABI as `T` (even if `T` is a ZST when monomorphized).

Like transparent `structs`, transparent `unions` are FFI-safe if and only if their underlying representation type is also FFI-safe.

A `union` may not be eligible for the same nonnull-style optimizations that a `struct` or `enum` (with the same fields) are eligible for. Adding `#[repr(transparent)]` to `union` does not change this. To give a more concrete example, it is unspecified whether `size_of::<T>()` is equal to `size_of::<Option<T>>()`, where `T` is a `union` (regardless of whether or not it is transparent). The Rust compiler is free to perform this optimization if possible, but is not required to, and different compiler versions may differ in their application of these optimizations.