auto_traits

The tracking issue for this feature is #13231

The `auto_traits` feature gate allows you to define auto traits.

Auto traits, like `Send` or `Sync` in the standard library, are marker traits that are automatically implemented for every type, unless the type, or a type it contains, has explicitly opted out via a negative impl. (Negative impls are separately controlled by the `negative_impls` feature.)

```
rust,ignore (partial-example) impl !Trait for Type {}
```

Example:

```
#![feature(negative_impls)]
#![feature(auto_traits)]

auto trait Valid {}

struct True;
struct False;

impl !Valid for False {}

struct MaybeValid<T>(T);

fn must_be_valid<T: Valid>(_t: T) { }

fn main() {
    // works
    must_be_valid( MaybeValid(True) );

    // compiler error - trait bound not satisfied
    // must_be_valid( MaybeValid(False) );
}
```

Automatic trait implementations

When a type is declared as an `auto trait`, we will automatically create impls for every struct/enum/union, unless an explicit impl is provided. These automatic impls contain a where clause for each field of the form `T: AutoTrait`, where `T` is the type of the field and `AutoTrait` is the auto trait in question. As an example, consider the struct `List` and the auto trait `Send`:

```
struct List<T> {
  data: T,
```

```
  next: Option<Box<List<T>>>,
}
```

Presuming that there is no explicit impl of `Send` for `List`, the compiler will supply an automatic impl of the form:

```
struct List<T> {
  data: T,
  next: Option<Box<List<T>>>,
}

unsafe impl<T> Send for List<T>
where
  T: Send, // from the field `data`
  Option<Box<List<T>>>: Send, // from the field `next`
{ }
```

Explicit impls may be either positive or negative. They take the form:

```
rust,ignore (partial-example) impl<...> AutoTrait for StructName<..>
{ } impl<...> !AutoTrait for StructName<..> { }
```

Coinduction: Auto traits permit cyclic matching

Unlike ordinary trait matching, auto traits are **coinductive**. This means, in short, that cycles which occur in trait matching are considered ok. As an example, consider the recursive struct `List` introduced in the previous section. In attempting to determine whether `List: Send`, we would wind up in a cycle: to apply the impl, we must show that `Option<Box<List>>: Send`, which will in turn require `Box<List>: Send` and then finally `List: Send` again. Under ordinary trait matching, this cycle would be an error, but for an auto trait it is considered a successful match.

Items

Auto traits cannot have any trait items, such as methods or associated types. This ensures that we can generate default implementations.

Supertraits

Auto traits cannot have supertraits. This is for soundness reasons, as the interaction of coinduction with implied bounds is difficult to reconcile.