generators

The tracking issue for this feature is: #43122

The `generators` feature gate in Rust allows you to define generator or coroutine literals. A generator is a "resumable function" that syntactically resembles a closure but compiles to much different semantics in the compiler itself. The primary feature of a generator is that it can be suspended during execution to be resumed at a later date. Generators use the `yield` keyword to "return", and then the caller can `resume` a generator to resume execution just after the `yield` keyword.

Generators are an extra-unstable feature in the compiler right now. Added in RFC 2033 they're mostly intended right now as a information/constraint gathering phase. The intent is that experimentation can happen on the nightly compiler before actual stabilization. A further RFC will be required to stabilize generators/coroutines and will likely contain at least a few small tweaks to the overall design.

A syntactical example of a generator is:

```
#![feature(generators, generator_trait)]

use std::ops::{Generator, GeneratorState};
use std::pin::Pin;

fn main() {
    let mut generator = || {
        yield 1;
        return "foo"
    };

    match Pin::new(&mut generator).resume(()) {
        GeneratorState::Yielded(1) => {}
        _ => panic!("unexpected value from resume"),
    }
    match Pin::new(&mut generator).resume(()) {
        GeneratorState::Complete("foo") => {}
        _ => panic!("unexpected value from resume"),
    }
}
```

Generators are closure-like literals which can contain a `yield` statement. The `yield` statement takes an optional expression of a value to yield out of the generator. All generator literals implement the `Generator` trait in the `std::ops` module. The `Generator` trait has one main method, `resume`, which resumes execution of the generator at the previous suspension point.

An example of the control flow of generators is that the following example prints all numbers in order:

```
#![feature(generators, generator_trait)]

use std::ops::Generator;
use std::pin::Pin;

fn main() {
    let mut generator = || {
        println!("2");
        yield;
        println!("4");
    };

    println!("1");
    Pin::new(&mut generator).resume(());
    println!("3");
    Pin::new(&mut generator).resume(());
    println!("5");
}
```

At this time the main intended use case of generators is an implementation primitive for async/await syntax, but generators will likely be extended to ergonomic implementations of iterators and other primitives in the future. Feedback on the design and usage is always appreciated!

The `Generator` trait

The `Generator` trait in `std::ops` currently looks like:

```
# #![feature(arbitrary_self_types, generator_trait)]
# use std::ops::GeneratorState;
# use std::pin::Pin;

pub trait Generator<R = ()> {
    type Yield;
    type Return;
    fn resume(self: Pin<&mut Self>, resume: R) -> GeneratorState<Self::Yield, Self::Return>;
}
```

The `Generator::Yield` type is the type of values that can be yielded with the `yield` statement. The `Generator::Return` type is the returned type of the generator. This is typically the last expression in a generator's definition or any value passed to `return` in a generator. The `resume` function is the entry point for executing the `Generator` itself.

The return value of `resume`, `GeneratorState`, looks like:

```
pub enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R),
}
```

The `Yielded` variant indicates that the generator can later be resumed. This corresponds to a `yield` point in a generator. The `Complete` variant indicates that the generator is complete and cannot be resumed again. Calling `resume` after a generator has returned `Complete` will likely result in a panic of the program.

Closure-like semantics

The closure-like syntax for generators alludes to the fact that they also have closure-like semantics. Namely:

- When created, a generator executes no code. A closure literal does not actually execute any of the closure's code on construction, and similarly a generator literal does not execute any code inside the generator when constructed.
- Generators can capture outer variables by reference or by move, and this can be tweaked with the `move` keyword at the beginning of the closure. Like closures all generators will have an implicit environment which is inferred by the compiler. Outer variables can be moved into a generator for use as the generator progresses.
- Generator literals produce a value with a unique type which implements the `std::ops::Generator` trait. This allows actual execution of the generator through the `Generator::resume` method as well as also naming it in return types and such.
- Traits like `Send` and `Sync` are automatically implemented for a `Generator` depending on the captured variables of the environment. Unlike closures, generators also depend on variables live across suspension points. This means that although the ambient environment may be `Send` or `Sync`, the generator itself may not be due to internal variables live across `yield` points being not-`Send` or not-`Sync`. Note that generators do not implement traits like `Copy` or `Clone` automatically.
- Whenever a generator is dropped it will drop all captured environment variables.

Generators as state machines

In the compiler, generators are currently compiled as state machines. Each `yield` expression will correspond to a different state that stores all live variables over that suspension point. Resumption of a generator will dispatch on the

current state and then execute internally until a `yield` is reached, at which point all state is saved off in the generator and a value is returned.

Let's take a look at an example to see what's going on here:

```
#![feature(generators, generator_trait)]

use std::ops::Generator;
use std::pin::Pin;

fn main() {
    let ret = "foo";
    let mut generator = move || {
        yield 1;
        return ret
    };

    Pin::new(&mut generator).resume(());
    Pin::new(&mut generator).resume(());
}
```

This generator literal will compile down to something similar to:

```
#![feature(arbitrary_self_types, generators, generator_trait)]

use std::ops::{Generator, GeneratorState};
use std::pin::Pin;

fn main() {
    let ret = "foo";
    let mut generator = {
        enum __Generator {
            Start(&'static str),
            Yield1(&'static str),
            Done,
        }

        impl Generator for __Generator {
            type Yield = i32;
            type Return = &'static str;

            fn resume(mut self: Pin<&mut Self>, resume: ()) -> GeneratorState<i32, &'static
                use std::mem;
                match mem::replace(&mut *self, __Generator::Done) {
                    __Generator::Start(s) => {
                        *self = __Generator::Yield1(s);
                        GeneratorState::Yielded(1)
                    }
```

```
                    __Generator::Yield1(s) => {
                        *self = __Generator::Done;
                        GeneratorState::Complete(s)
                    }

                    __Generator::Done => {
                        panic!("generator resumed after completion")
                    }
                }
            }
        }

        __Generator::Start(ret)
    };

    Pin::new(&mut generator).resume(());
    Pin::new(&mut generator).resume(());
}
```

Notably here we can see that the compiler is generating a fresh type, `__Generator` in this case. This type has a number of states (represented here as an `enum`) corresponding to each of the conceptual states of the generator. At the beginning we're closing over our outer variable `foo` and then that variable is also live over the `yield` point, so it's stored in both states.

When the generator starts it'll immediately yield 1, but it saves off its state just before it does so indicating that it has reached the yield point. Upon resuming again we'll execute the `return ret` which returns the `Complete` state.

Here we can also note that the `Done` state, if resumed, panics immediately as it's invalid to resume a completed generator. It's also worth noting that this is just a rough desugaring, not a normative specification for what the compiler does.