Sheet 5

1. Define a trait `Printable` that defines a method `print` that println`s` `self` to the console. Implement this trait for `i32`, `String` and `Vec<T>` where `T` implements `Printable`.
   Create a function `print` that takes a generic argument `T` that implements `Printable` and calls `print` on `T`.
   Decide whether to use monomorphization or dynamic dispatch for the `print` function.
2. Write a struct `Book` that contains two fields:

   - `title` of type `&str`
   - `cat` of type `Category` (`Category` is an enum with some variants)

   Write a struct `Library`, with a field `bookcases` of type `[Vec<Book>; 10]`. Every bookcase have 10 floors, and every floor can hold a number of books.
   Derive the `Debug` trait for `Library`, without manually implementing it for any of the types that you defined.
   Implement the trait `Default` for `Book` giving it a random `cat` and a random `title`.
   Implement `default_with_cat` for `Book`, that takes a `Category` and returns an instance with the specified category and use the `default` method for `title`. (hint: use the `..` operator)
   Derive the trait `Default` for `Category` and `Library`.
   Implement a trait `Populatable` that defines the `populate` function. Implement `Populatable` for `Library`, populating its bookcases with 3 books for each floor, using the default Category.
3. Define a function `restricted` that is generic over `T` and `U`.

   - `T` types should be comparable and debuggable.
   - `U` types should be displayable.

   The function should take two arguments `t1` and `t2` of type `T` and `u` of type `U` respectively.
   It should then print the smaller of `t1` and `t2` to the console together with `u` like this:

   ```
   minor: <t1>
   u:     <u>
   ```

   and return the smaller between `t1` and `t2`.
4. Define a struct `Tasks` that has the following fields: `tasks`: a `Vec<Task>` en define a struct `Task` that has the following fields: `name`: a `String``priority`: an `i32``done`: a `bool` implement useful methods for both structs (e.g. `new`). Implement the `Iterator` trait for `Tasks` such that it returns each task that has not been completed yet and removes the completed ones from the `Vec`.
5. Define the struct `Pair(i32, String)` and then implement the traits contained in `std::ops` and needed for adding the possibility to do the following operations, every operation must return another `Pair`:

- `Pair` + `i32`: add the `i32` to `Pair.0`
- `Pair` - `i32`: *same as above*
- `Pair` + `&str`: append the `&str` to `Pair.1`
- `Pair` - `&str`: search the `&str` in `Pair.1` and replace it with `""`
- `Pair` + `Pair`: like doing `Pair1 + Pair2.0 + Pair2.1`
- `Pair` - `Pair`: *same as above*
- `Pair` * `i32`: with `n` the `i32`, `Pair.0` to the `n`th power, `Pair.1` repeated `n` times.

6. Write a struct `Gate` generic over `S` that represents the state of the gate:

   - `Open`
   - `Closed`
   - `Stopped, with a field reason: String`

   The gate can be in one of these states at any given time. Each state has a different set of available methods:

   - `Open`: `close`
   - `Closed`: `open`
   - `Stopped`: `open`, `close`

   Each method takes ownership of `self`.
   The `close` method returns a `Result`:

   - `Ok<Gate<Closed>>` if the gate was successfully closed
   - `Err<Gate<Stopped>>` if the gate could not be closed

   The `open` method returns a `Result`:

   - `Ok<Gate<Open>>` if the gate was successfully opened
   - `Err<Gate<Stopped>>` if the gate could not be opened

   The `open` and `close` methods of the `Open` and `Closed` states have a random chance of failing (user defined).
   The `open` and `close` methods of the `Stopped` state always succeed.

   In order to define the state as a generic, the `Gate` struct should contain a "State Marker" field (i.e. a field of type `S`). This field is not used anywhere, but it is used to tell the compiler that the generic type `S` is used in the struct. If you're interested in a more detailed explanation, you can read about PhantomData: https://doc.rust-lang.org/std/marker/struct.PhantomData.html

7. Define two traits `Heatable` and `Friable` that have one method each named `cook`. Then define the trait `Heater` and `Frier` that have respectively two methods: `heat` and `fry`, where each method will take a reference to `self` and a trait object of `Heatable` and `Friable`.

   Then create two "cooker" structs:

- `Oven` that implements `Heater` that simply calls the method `cook` of the `Heatable` trait.
- `Pan` that implements `Heater` and `Frier`, the implementation is the same as above for each trait.

Then create two "food" structs:

- `Pie` that has one bool field `ready`
- `Carrot` that has one field of type `CarrotState`

`CarrotState` is an `enum` that has 4 variants: `Raw`, `Cooked`, `Fried`, `Burnt`.

Define the trait `Edible` that defines the method `eat`. Now implement the following traits:

- `Heatable` for `Pie`: if the pie is already ready it prints "you burned the pie!", otherwise it sets `ready` to true;
- `Heatable` for `Carrot`: if the carrot is not raw, then the carrot is burnt otherwise the carrot is cooked.
- `Friable` for `Carrot`: if the carrot is already fried, the carrot is burnt otherwise the carrot is fried.
- `Edible` for `Pie`: if the pie isn't ready it prints "you got stomach ache" otherwhise "yummy"
- `Edible` for `Carrot`: it prints,
  - "mmh, crunchy" if it's raw;
  - "mmh, yummy" if it's cooked;
  - "mmh, crispy" if it's fried;
  - "mmh, burnt" if it's burnt;