Smart Pointers: Box

Learn to Code with Rust / Section Review

Pointers and References

- A **pointer** is a variable that contains an address in memory.
- A **reference** is a type/category of pointer that points to valid data.
- Rust's borrow checker guarantees that *references* point to valid/allocated data.

Raw Pointers

- A **raw pointer** is a variable that stores a memory address without any safety checks.
- A **raw pointer** *may* point to valid data but it can also point to deallocated memory.
- Create a raw pointer with **&raw const** before the value.
- Code that dereferences the raw pointer must be within an **unsafe** block. The developer acknowledges the risk of the code.

Smart Pointer

- A **smart pointer** is a type that *behaves like* a reference.
- A **smart pointer** can store additional information and perform more actions compared to a plain reference.
- Most **smart pointers** are built with structs.
- Smart pointers often own and manage their own data, typically on the heap.
- The advantage of smart pointers is that they are owned types that *behave like* pointers.

Built-In Smart Pointers

- The **String** and **Vec** types are smart pointers!
- They abstract away the storage and access of heap data.
- Code can interact with the String and Vec types as standard owned values.
- When the owner goes out of scope, it deallocates the stack entry for the String/Vec, which also cleans up the dynamically-sized heap memory.

The Box Smart Pointer

- The **Box** smart pointer stores a piece of data on the heap.
- The **Box** smart pointer hides away the complexity of dealing with the raw pointer to the heap data.
- The size of the heap data that the **Box**'s raw pointer points to can vary in size.
- The **Box**'s size on the stack stays consistent. The amount of memory it takes to store a raw pointer doesn't change depending on the size of the data that the raw pointer points to.

LinkedList and BinarySearchTree

- The **Box** is helpful when modelling recursive data structures, types that contain themselves.
- A **linked list** is a data structure that consists of a collection of connected nodes.
- A **node** represents an individual unit of data.
- A **binary search tree** is a tree data structure that speeds up search by halving the search area on every check.
- Nodes to the left of each node will have smaller values. Nodes to the right of each node will have larger values.

The **Deref** and **DerefMut** Traits

- Smart pointers will implement the **Deref** trait.
- The **Deref** trait defines what happens when code uses the dereference operator (*) on the smart pointer.
- The **Deref** trait requires a **deref** method that returns a reference to some data inside the type.
- The **Target** associated type sets the return value of the **deref** method, enabling a consistent (but dynamic) method signature.
- The complementary **DerefMut** trait allows overwriting content inside the smart pointer through a mutable reference.

Deref Coercions

- When given a reference to a type that implements the **Deref** trait, Rust will convert it into a reference of another type if necessary.
- If our code passes a **&String** argument to a function that has a **&str** parameter, deref coercion will convert the **&String** into a **&str**.
- Rust will continue performing deref coercion until it gets to the intended parameter type.
- The **Box** smart pointer implements the **Deref** trait to return a reference to the nested data.

Trait Objects I

- A **trait object** is an instance of some type that implements a specific trait.
- Trait objects enable code that does not have to couple itself to a single concrete type.
- Nest the trait object inside the **Box**. The **Box** occupies a fixed size on the stack; the heap data/trait object can be any size.
- At runtime, the program will determine the exact type and invoke the appropriate method on it. This feature is called **dynamic dispatch**.

Trait Objects II

- Add the **dyn** keyword before the shared trait that all types will implement.
- **Usecase**: Create a vector of dynamic types.
  - Use **Vec<Box<dyn MyTrait>>** to annotate its type.
- **Usecase:** Return a **Result** from a function where the **Err** variant can store multiple error types.
  - Use **Result<T, Box<dyn MyTrait>>** to annotate its type.

Custom Error Types

- Declare a struct (with optional data) and implement the **std::error::Error** trait.
- The **Error** trait is a subtrait of **Display** and **Debug**.
- The type must be representable as a string.
- All Rust error types implement the **Error** trait.