CMSC 330: Organization of Programming Languages

Box Smart Pointer, Trait Objects

Box<T> Smart Pointers

- `Box<T>` values point to heap-allocated data
  - The `Box<T>` value (the pointer) is on the stack, while its pointed-to `T` value is allocated on the heap
  - Has `Deref` trait – can be treated like a reference
    - More later
  - Has `Drop` trait – will drop its data when it dies

- Uses?
  - Reduce copying (via an ownership move)
  - Create dynamically sized objects
    - Particularly useful for recursive types

Example: Linked List

- Naïve attempt doesn't work
  - Compiler complains that it can't know the size of **List**
  - The **Cons** case is "inlined" into the **enum**
    - Since a **List** is recursive, it could be basically any size

```
enum List {
  Nil,
  Cons(i32,List)
}
```

- Use a **Box** to add an indirection
  - Now the size is fixed
    - **i32** + size of pointer
      - **Nil** tag smaller

```
enum List {
  Nil,
  Cons(i32,Box<List>)
}
```

Creating a LinkedList

```
enum List {
  Nil,
  Cons(i32,Box<List>)
}

use List::{Cons, Nil};

fn main() {
  let list = Cons(1,
    Box::new(Cons(2,
      Box::new(Nil))));
  … // data dropped at end of scope
}
```

Deref Trait

- If **x** is an **int** then **&x** is a **&{int}**
  - Can use ***** operator to dereference it, extracting the underlying value
    - ***(&x) == x**
- Can use ***** on **Box<T>** types
  - **Deref** trait requires **deref(&self) -> &T** method
  - So that ***x** translates to ***(x.deref())**

- **deref** returns type **&T** and not **T** so as not to relinquish ownership from inside the **Box** type

Deref Coercion

- The Rust compiler automatically inserts one or more calls to **`x.deref()`** to get the right type
  - When **`&T`** required but value **`x : U`** provided, where **`U`** implements **`Deref`** trait
  - In particular, at function and method calls
- Also a **`DerefMut`** trait, for when object is mutable
  - **`Deref`** coercion works with this too (see Rust book)

Example

```
fn hello(x:&str) {
    println!("hello {}",x);
}
fn main() {
    let m = Box::new(String::from("Rust"));
    hello(&m); //same as hello(&(*m)[..]);
}
```

- **`&m`** should have type **`&str`** to pass it to **`hello`**
- So, compiler calls **`m.deref()`** to get **`&String`**, and then **`deref()`** again to get **`&str`**

Drop Trait

- Provides the method **`fn drop(&mut self)`**
  - Called when the value implementing the trait goes out of scope
  - Should be used to free the underlying resources, e.g., heap memory

- May not call drop method manually
  - Would lead to a **double free** when Rust calls the method again at the end of a scope
  - Can call **`std::mem::drop`** function in some circumstances

Another Place Where Size Matters

- Recall **`Summarizable`**

```
pub trait Summarizable {
  fn summary(&self) -> String {
    String::from("none")
  }
}

impl Summarizable for i32 {…}
```

- Let's make a general summary-printing function
- First attempt: **`fn print_summary(s: Summarizable) {…}`**
  - This means the caller *moves* (or copies, if s is **`Copy`**) the argument to the function when calling it (**`s`** is not a reference)
  - This means the *data* in the argument needs to be moved/copied
  - How many bytes long is the data? Don't know; won't work

Still Not Right

- Recall **`Summarizable`**

```
pub trait Summarizable {
  fn summary(&self) -> String {
    String::from("none")
  }
}

impl Summarizable for i32 {…}
```

- Second attempt, also wrong:

```
fn print_summary(s: &Summarizable) {
  print!("{}", s.summary());
}
```

  - There are lots of implementations of **`summary`**
  - Which one should be invoked?

What's Missing: Receiver Type

- This code was OK; why?

```
let x:i32 = 42;
x.summarize();
```

- The compiler knows which **`summarize`** to call, since it knows **`x:i32`**

Dynamic Dispatch

```
fn print_summary(s: &Summarizable) {
  print!("{}", s.summary());
}
```

- Object oriented languages, like Java, accept code like the above because they have dynamic dispatch
  - The correct method is determined at run time

- To implement dispatch in Rust, we use trait objects
- A trait object pairs data with runtime type information
  - Think: **(42, "I am an i32!")**

Trait Objects

- Use type **`dyn Summarizable`**, wrapped in a **`Box`**

```
fn print_summarizable(s: Box<dyn Summarizable>) {
    println!("{}", s.summary());
}
```

- Callers simply use **`Box`** to put the data on the heap

```
pub fn main() {
    let b = Box::new(42);
    print_summarizable(b);
}
```

Why the Box?

- Could we do this instead?

```
fn print_s(s: dyn Summarizable) {
  println!("{}", s.summary());
}
```

- Error!

```
17 | fn print_s(s: dyn Summarizable) {
   |           ^ doesn't have a size known at compile-time
   |
   = help: the trait `Sized` is not implemented for `(dyn
Summarizable + 'static)`
help: function arguments must have a statically known size,
borrowed types always have a known size
```

Lesson: **`dyn Summarizable`** has different sizes; **`Box<T>`** has one

Box and Size

- **`Box<i32>`** is a pointer to a heap-allocated **`i32`**
- **`Box<dyn Summarizable>`** is a fat pointer to a heap-allocated **`Summarizable`**
  - That is: (type information, pointer to data on the heap)

Example

```
struct Enormous { // 512 bytes (4 * 128)
  a: [i32; 128],
}

impl Summarizable for Enormous {…}
```

```
println!("{}", std::mem::size_of::<Enormous>());
println!("{}", std::mem::size_of::<Box<Enormous>>());
println!("{}", std::mem::size_of::<Box<Summarizable>>());
println!("{}", std::mem::size_of::<Box<dyn Summarizable>>());
```

512
8
Error
16

Box: a Kind of Smart Pointer

- A smart pointer is a reference plus metadata, to provide additional capabilities
  - Originated in C++
  - Examples seen so far: `String`, `Vec<T>`, `Box<T>`
- Usually implemented as `struct`s
  - Which must implement the `Deref` and `Drop` traits
- New ones we will see: `Cell<T>`, `Rc<T>`, `Ref<T>`, …
- Check out *The Rustonomicon* for how to implement your own smart pointers!
  - https://doc.rust-lang.org/stable/nomicon/

Summary

- Use `Box<T>` to heap-allocate data, and reduce copying (via an ownership move)
  - Useful for non cyclic, immutable data structures
- Use trait objects, of type `Box<dyn Trait>`, to implement dynamic dispatch
  - For any trait type `Trait`
  - `Box` lets you use *fat pointers* for `dyn` trait objects, to provide runtime type information to enable dynamic dispatch
  - If you try to pass traits without `Box`, you may get errors about `Sized` because the compiler doesn't know how big things are