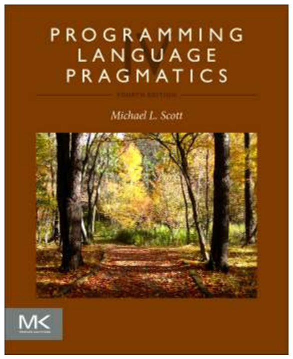
Objects & Traits in Rust

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 10

Prof. Jonathan Aldrich

Object-Oriented Programming (OOP)

Three key aspects:

- Encapsulation
  - An object is a grouping of state and behavior, and hides its implementation choices from the outside world
- Inheritance
  - Objects are related, and we can capture shared behavior in a way that multiple kinds of objects can use it without defining it themselves
- Dynamic dispatch
  - The same operation can be implemented in different ways; each object knows what implementation to use for each of its operations

This aspect is special:
- Unique to objects
- Present in all OO languages

Encapsulation

- An object is a grouping of state and behavior

```
let setImpl = {
      members : [1, 2, 3],
      isMember : function(x) {
            return this.members.includes(x);
      },
      add : function(x) {
            if (!isMember(x))
                  this.members.push(x);
      }
};
setImpl.add(4);      // uses the object
setImpl.isMember(4); // returns true
```

state

behavior

this refers to the current object instance

Encapsulation

- We can hide some of the object's state

```
interface IntSet {
      isMember : (x:number) => boolean
      add : (x:number) => void
}

let setImpl = { ... };
let set : IntSet = setImpl;

set.add(4);
set.isMember(4);
set.members
```

interface `IntSet` leaves out the `members` field. We can change that later without affecting clients.

Assigning to a variable of type `IntSet` hides everything that's not in the interface

It's a type error to access members that are not exposed in the interface

ELSEVIER

Classes

- A *class* is a template for objects. It defines structure & behavior used by all *instances* of the class

```
class IntSetClass {
      members : number[];
      constructor(m:number[]) {
            this.members = m;
      }
      isMember(x:number):boolean {
            return this.members.includes(x);
      }
      // add(x:number):void { ... }
}

let set2 : IntSetClass = new IntSetClass([1, 2]);
set2.add(5);
set2.isMember(5); // returns true
```

Dynamic Dispatch

- Every object knows its method implementations (whether defined in the object, or in that object’s class)
- When we invoke a method, the code for that object is run

```
class Dog {
    talk() { console.log("woof!"); }
}
class Cat {
    talk() { console.log("meow!"); }
}
let animals = [new Dog(), new Cat() ];
for (let a of animals)
    a.talk(); // prints woof! meow!
```

Inheritance

- Inheritance lets us reuse code from one class in another
    - Prototype: a variant where you reuse code from another object (see JavaScript)

```
class Collection {
    constructor(ms) { this.members = ms; }
    isMember(x) { return this.members.includes(x); }
    add(x) { this.members.push(x); }
    addAll(a) { for (let x of a) this.add(x); }
}
class Set extends Collection {
    constructor(ms) { super(ms); }
    add(x) { if (!this.isMember(x)) { super.add(x); } }
}
let set = new Set([]);
set.add(3);
set.addAll([3, 4]);
set.isMember(4);
```

Exercise

- Draw the frames on the runtime stack when 4 is added to the set in the call set.addAll([3, 4]). Show all methods that are in from main() through push()

```
class Collection {
    constructor(ms) { this.members = ms; }
    isMember(x) { return this.members.includes(x); }
    add(x) { this.members.push(x); }
    addAll(a) { for (let x of a) this.add(x); }
}
class Set extends Collection {
    constructor(ms) { super(ms); }
    add(x) { if (!this.isMember(x)) { super.add(x); } }
}
let set = new Set([]);
set.add(3);
set.addAll([3, 4]);
```

Why Objects Matter

- Encapsulation (not specific to objects)
  - Separate reasoning about a single module enhances correctness & finding bugs
  - Ability to change the internals of a module without affecting others enhances software evolution
- Inheritance
  - Some code patterns are difficult to reuse in any other way
    - Typically when you have a reusable part and a customizable part, and they both call each other
  - That said, many uses of inheritance can (and should) be replaced with composition
    - Common guideline: prefer composition to inheritance
- Dynamic dispatch
  - Architecturally important – support multiple independent & interoperating implementations of a common interface
  - Examples all over the place: mobile phone apps, Linux device drivers, graphical user interfaces, MapReduce, web frameworks

Traits in Rust

- Traits are Rust's equivalent of OO interfaces

```
struct Sprocket {
    name: String
}
struct Cog {
    id: u32
}
trait Sortable {
    fn get_sort_name(&self) -> String;
    fn less_than(&self, o: &dyn Sortable) -> bool {
        return self.get_sort_name() < o.get_sort_name()
    }
}
```

These types may be defined in a separate module, and may not be easy to chnage

A trait is an interface, possibly with some reusable method implementations

self is the name Rust uses for this

o is dyn, which allows dynamic dispatch

Traits in Rust

- Traits are Rust's equivalent of OO interfaces

```
struct Sprocket {
    name: String
}
struct Cog {
    id: u32
}
trait Sortable {
    fn get_sort_name(&self) -> String;
    fn less_than(&self, o: &dyn Sortable) -> bool {
        return self.get_sort_name() < o.get_sort_name()
    }
}
impl Sortable for Sprocket {
    fn get_sort_name(&self) -> String { return self.name.clone() }
}
impl Sortable for Cog {
    fn get_sort_name(&self) -> String {
        return "Cog".to_string() + &self.id.to_string()
    }
}
```

We implement the trait, providing `Sortable` functionality in a way appropriate for the specified type

Using Rust Traits

```
trait Sortable {
    fn get_sort_name(&self) -> String;
    fn less_than(&self, o: &dyn Sortable) -> bool {
        return self.get_sort_name() < o.get_sort_name()
    }
}
impl Sortable for Sprocket {
    fn get_sort_name(&self) -> String { return self.name.clone() }
}
impl Sortable for Cog {
    fn get_sort_name(&self) -> String {
        return "Cog".to_string() + &self.id.to_string()
    }
}
fn main() {
    let w1 = Cog { id: 3 };
    let s1 = Sprocket { name: "Spacely".to_string() };
    let w1_name = w1.get_sort_name();
    let s1_name = s1.get_sort_name();
    println!("{} < {}? {}", w1_name, s1_name, w1_name < s1_name);
         // in sorted order? yes; prints "true"
}
```

Two ways to dispatch

```
trait Sortable {
    fn get_sort_name(&self) -> String;
```

dyn enables dynamic dispatch
Implemented with a “fat pointer”: a pointer to the value and a pointer to the trait impl

```
    fn less_than(&self, o: &dyn Sortable) -> bool {
        return self.get_sort_name() < o.get_sort_name()
    }
```

impl means dispatch is static.
Must know at the call site which implementation is used

```
    fn less_than(&self, o: &impl Sortable) -> bool {
        return self.get_sort_name() < o.get_sort_name()
    }
```

impl is actually syntactic suger for a parameterized type.
(equivalent to above code)

```
    fn less_than<T: Sortable>(&self, o: &T) -> bool {
        return self.get_sort_name() < o.get_sort_name()
    }
}
```

Initialization and Finalization

- We defined the lifetime of an object to be the interval during which it occupies space and can hold data
  - Most object-oriented languages provide some sort of special mechanism to *initialize* an object automatically at the beginning of its lifetime
    - When written in the form of a subroutine, this mechanism is known as a *constructor*
    - A constructor does not allocate space
  - A few languages provide a similar *destructor* mechanism to *finalize* an object automatically at the end of its lifetime

Initialization and Finalization

Issues

- choosing a constructor
- references and values
  - If variables are references, then every object must be created explicitly - appropriate constructor is called
  - If variables are values, then object creation can happen implicitly as a result of elaboration
- execution order
  - When an object of a derived class is created in C++, the constructors for any base classes will be executed before the constructor for the derived class
- garbage collection

Dynamic Method Binding

- Data members of classes are implemented just like structures (records)
  - With (single) inheritance, derived classes have extra fields at the end
  - A pointer to the parent and a pointer to the child contain the same address - the child just knows that the struct goes farther than the parent does

Dynamic Method Binding

- Non-virtual functions require no space at run time; the compiler just calls the appropriate version, based on type of variable
  – Member functions are passed an extra, hidden, initial parameter: *this* (called *current* in Eiffel and *self* in Smalltalk)
- C++ philosophy is to avoid run-time overhead whenever possible(Sort of the legacy from C)
  – Languages like Smalltalk have (much) more run-time support

Dynamic Method Binding

- Virtual functions are the only thing that requires any trickiness (Figure 10.3)
  - They are implemented by creating a dispatch table (*vtable*) for the class and putting a pointer to that table in the data of the object
  - Objects of a derived class have a different dispatch table
    - In the dispatch table, functions defined in the parent come first, though some of the pointers point to overridden versions
    - You could put the whole dispatch table in the object itself, saving a little time, but potentially wasting a LOT of space

Dynamic Method Binding

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```

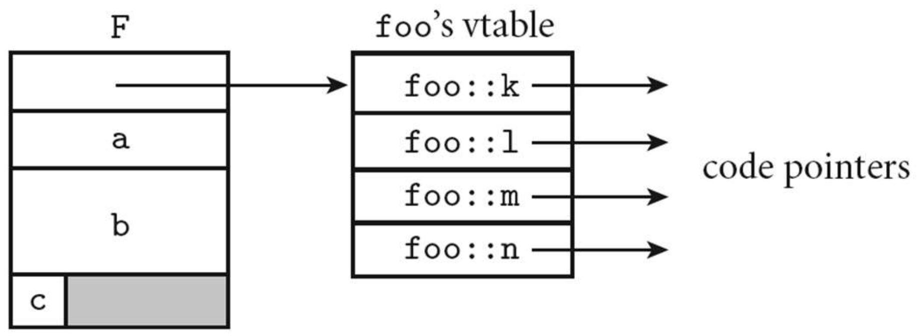


Figure 10.3 **Implementation of virtual methods.** The representation of object `F` begins with the address of the vtable for class `foo`. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of `F` consists of the representations of its fields.

Dynamic Method Binding

```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```

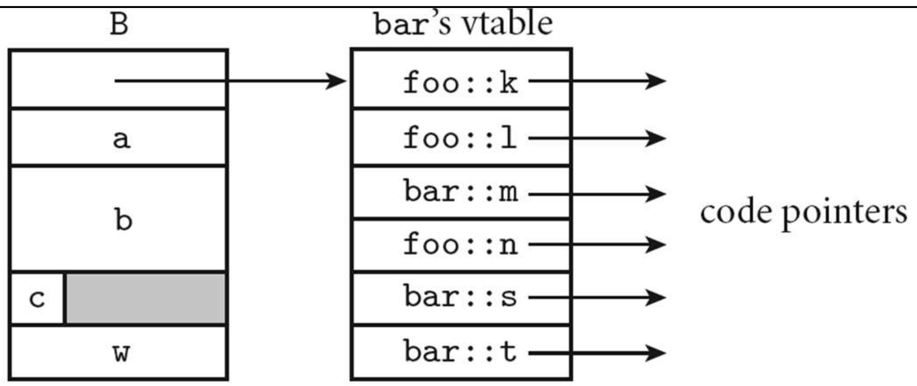


Figure 10.4 Implementation of single inheritance. As in Figure 10.3, the representation of object `B` begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for `foo`, except that one—`m`—has been overridden and now contains the address of the code for a different subroutine. Additional fields of `bar` follow the ones inherited from `foo` in the representation of `B`; additional virtual methods follow the ones inherited from `foo` in the vtable of class `bar`.

Dynamic Method Binding

- Note that if you can query the type of an object, then you need to be able to get from the object to run-time type info
  - The standard implementation technique is to put a pointer to the type info at the beginning of the vtable
  - Of course you only have a vtable in C++ if your class has virtual functions
    - That's why you can't do a dynamic_cast on a pointer whose static type doesn't have virtual functions

Mix-In Inheritance

- Classes can inherit from only one "real" parent
- Can "mix in" any number of interfaces, simulating multiple inheritance
- Interfaces appear in Java, C#, Go, Ruby, etc.
  - contain only abstract methods, no method bodies or fields
- Has become dominant approach, superseding true multiple inheritance

Mix-In Inheritance

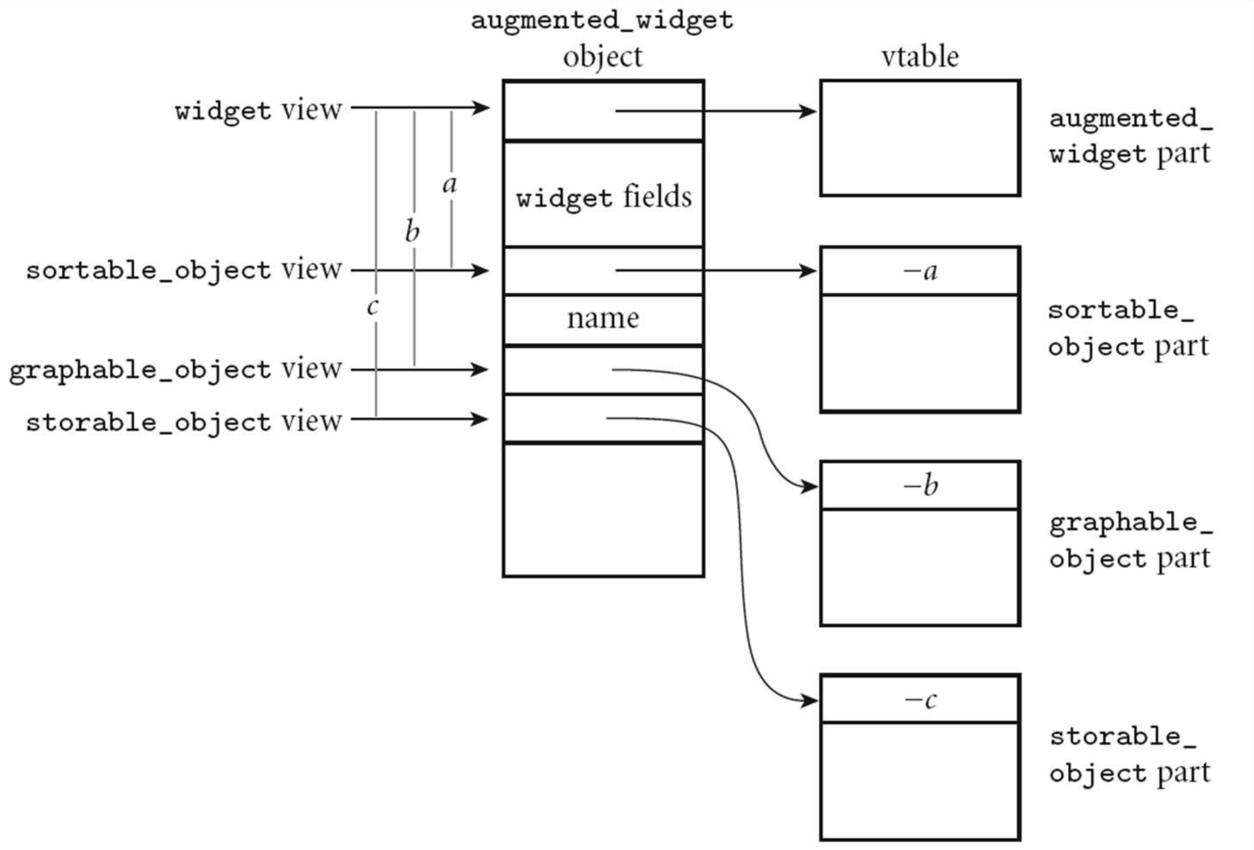


Figure 10.7 Implementation of mix-in inheritance. Objects of class `augmented_widget` contain four vtable addresses, one for the class itself (as in Figure 10.3), and three for the implemented interfaces. The view of the object that is passed to interface routines points directly at the relevant vtable pointer. The vtable then begins with a "`this` correction" offset, used to regenerate a pointer to the object itself.

Semantics

- We can use the static and dynamic semantics techniques we have learned to model objects

Source: Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. OOPSLA 1999.