RUST TRAITS IN C++

LOUIS DIONNE

https://a9.com/careers

CONSIDER A SIMPLE CLASS HIERARCHY

```
struct Shape { virtual int area() const = 0; };

struct Square : Shape { virtual int area() const override { ... } };
struct Circle : Shape { virtual int area() const override { ... } };
```

TRY TO ADD A NEW TYPE THAT YOU DON'T CONTROL

```
namespace lib {
  struct Hexagon { ... };
}

void draw(Shape*) { ... } // Can't accept an Hexagon!
```

HOW OFTEN HAVE YOU SEEN THIS?

```
struct Shape {
  virtual int area() const = 0;
  virtual int radius() const = 0;
};

struct Square : Shape {
  virtual int area() const { ... }
  virtual int radius() const {
    assert(false && "Square does not support the radius method!");
  }
};
```

BY THE WAY, ARE YOU HAPPY WITH THOSE ALLOCATIONS?

```
void draw(Shape*) { ... }

Shape* foo = new Square{...};
draw(foo);
```

ALSO, CAN YOU SPOT THE PROBLEM HERE?

```
struct Ellipsis : Shape {
  virtual int area() const override { ... }
  std::string name;
};

Shape* foo = new Ellipsis{"foo"};
draw(foo);
delete foo;
```

WHAT IF I NEED TO COPY THOSE THINGS?

```
std::vector<Shape*> shapes = ...;

std::vector<Shape*> new_shapes = shapes;
for (auto* shape : new_shapes) {
  shape->scale(2); // WRONG!!!
}
```

BOTTOM LINE: INHERITANCE IS A POOR MECHANISM

BUT WHY?

JUST LISTEN TO SEAN PARENT

REQUIREMENT OF A POLYMORPHIC TYPE COMES FROM ITS USE

A TYPE IS NOT POLYMORPHIC BY ITSELF, THE USAGE IS

INHERITANCE-BASED POLYMORPHISM BREAKS VALUE SEMANTICS

HIERARCHIES ARE NOT EXTENSIBLE AND INTRUSIVE

INHERITANCE COUPLES POLYMORPHISM WITH STORAGE

ENTER RUST TRAITS

```
struct Circle {
  x: f64,
  y: f64,
  radius: f64,
}

trait HasArea {
  fn area(&self) -> f64;
}

impl HasArea for Circle {
  fn area(&self) -> f64 {
    std::f64::consts::PI * (self.radius * self.radius)
  }
}
```

POWERFUL AND SIMPLE TO USE

```
fn print_area<T: HasArea>(shape: T) {
  println!("This shape has an area of {}", shape.area());
}

fn main() {
  let c = Circle {
    x: 0.0f64,
    y: 0.0f64,
    radius: 1.0f64,
  };

  print_area(c);
}
```

ONLY PROBLEM: IT'S NOT C++

DYNO GOT YOUR BACK

```
struct Circle {
  int x, y, radius;
};

struct HasArea : decltype(dyno::requires(
  "area"_s = dyno::function<int (dyno::T const&)>
)) { };

template <>
auto const dyno::concept_map<HasArea, Circle> = dyno::make_concept_map(
  "area"_s = [](Circle const& circle) {
    return 3.1415 * circle.radius * circle.radius;
  }
);
```

EASY TO USE

```
void print_area(has_area shape) {
  std::cout << "This shape has an area of " << shape.area();
}

int main() {
  Circle cirle{0, 0, 1};
  print_area(circle);
}
```

THE ONLY MISSING PART

```
struct has_area {
  template <typename Shape>
  has_area(Shape shape) : shape_{shape} { }
  int area() const { return shape_.virtual_("area"_s)(shape_); }
private:
  dyno::poly<HasArea> shape_;
};
```

CAN CUSTOMIZE STORAGE

```
struct has_area {
  ...
private:
  dyno::poly<HasArea>;                              // heap
  dyno::poly<HasArea, dyno::remote_storage>;    // heap
  dyno::poly<HasArea, dyno::local_storage<8>>;  // stack
  dyno::poly<HasArea, dyno::sbo_storage<8>>;    // stack, heap fallback
  dyno::poly<HasArea, dyno::non_owning_storage>;// just a ref
};
```

CAN CUSTOMIZE VTABLE

```
struct has_area {
  ...
private:
  using VTable = dyno::vtable<
    dyno::local<dyno::only<decltype("area"_s)>>,
    dyno::remote<dyno::everything_else>
  >;
  dyno::poly<HasArea, VTable> shape_;
};
```

BOTTOM LINE

- More flexible than inheritance
- Respects value semantics
- Full control over performance

TRY IT OUT!

http://github.com/ldionne/dyno

http://ldionne.com