Poly types

Tian Liao

December 7, 2023

Document number:
Date: December 7, 2023
Audience:
Authors: Tian Liao, Mingxin Wang
Reply-to: Tian Liao <tilia@microsoft.com>

1 Introduction

2 Proposal

2.1 Language feature

This proposal is going to add new keywords: *trait, declvptr.* Here is a quick glance:

```
// define some materials.
struct Color { void apply() {} };
struct Texture { void apply() {} };
struct Glass { void apply() {} };

// declare a type alias.
trait Material { void apply(); };

void foo() {
  {
    // refer to an instance in stack.
    Color color;
    declvptr(Material, Color)->apply(&color); // invokes Color::apply.
  }
  {
    // refer to an instance in heap.
    auto texture = new Texture();
    declvptr(Material, Texture)
      ->apply(texture); // invokes Texture::apply.
    delete texture;
  }
  {
    // refer to an instance hosted by fancy pointer.
    auto glass = std::make_unique<Glass>();
    declvptr(Material, Glass)
      ->apply(glass.get()); // invokes Glass::apply.
  }
  {
    // vptr dispatch accepts a void pointer.
    Color color;
    void* ptr = &color;
    declvptr(Material, Color)->apply(ptr); // invokes Color::apply.
  }
  {
    struct NonMaterial { void bar() {} };
    declvptr(Material, NonMaterial); // compile-time error.
  }
}
```

A trait can combine with other traits to form a trait.

```
trait Source{ void read(); };
trait Sink{ void write(); };

trait DuplexStream : Source, Sink {};
trait DuplexStreamEquvalent {
  void read();
  void write();
};
static_assert(std::is_same_v<DuplexStream, DuplexStreamEquvalent>);
```

A trait can declare fields.

```
trait Account {
  std::string Name;
  std::string Email;
};

struct WebAccount {
  std::string Name;
  std::string Email;
};

void foo() {
  WebAccount user{ .Name = "Bob", .Email = "Bob@email.com" };
  declvptr(Account, WebAccount)->Name(&user); // returns "Bob".
  declvptr(Account, WebAccount)
    ->Email(&user, "Bob2@email.com"); // user.Email changed.
}
```

A trait can have function overloads.

```
trait Overloads{
  void test();
  void test(int);
};

void consume(Overloads* vptr, void* obj) {
  vptr->test(obj); // invokes Overloads::test().
  vptr->test(obj, 0); // invokes Overloads::test(int);
}

void produce() {
  struct {
    void test() { print("void\n"); };
    void test(int) { print("int\n"); };
  } o;
  consumer(declvptr(Overloads, decltype(o)), &o);
}
```

A trait can be template.

```
template <typename T, std::size_t I>
trait GenericMaterial {
  using type = T;
  static constexpr std::size_t index = I;
  void apply(const T& target);
};
```

2.2 Library feature

```
namespace std {
template <trait Trait>
class poly_ptr {
 public:
  template <class U>
  poly_ptr(U* data)
    : vptr_(declvptr(Trait, U)), data_(data);
  auto operator->() {
    // TODO: return a pointer.
  };

  explicit operator bool() const {
    return data_ != nullptr &&
           vptr_ != nullptr;
  }

 private:
  Trait* vptr_ = nullptr;
  void* data_ = nullptr;
}
} // namespace std

void foo() {
  Glass glass;
  std::poly_ptr<Material> mat = &glass; // accepts a raw pointer.
  assert(static_cast<bool>(dummy)); // not nullptr.
  mat->apply();
}
```

3 Motivation