Enhanced typename

Tian Liao

November 27, 2023

Document number:
Date: November 27, 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 extend the semantics of the *typename* keyword to allow it defines type aliases that represent type erasers who keep their underlying types associated and dispatch runtime invocations properly.
Here is a quick glance:

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

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

void foo() {
  {
    // use Material as a pointer.
    Color color;
    Material* material = color;
    material->apply();
  }
  {
    // use Material as a reference.
    Texture texture;
    Material& material = texture;
    material.apply();
  }
  {
    // host a Material in unique ptr.
    std::unique_ptr<Material> material{new Glass()};
    material->apply();
  }
}

// defining a type alias is not allowed.
void Material::apply() {} // compile error.

// instanciate a type alias is also not allowed.
Material some_material; // compile error.
```

A type alias can combine other type aliases to form a new type alias.

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

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

A type alias can declare fields.

```
typename Account {
  void RefreshData();
  std::string Name;
  std::string Email;
};

class WebAccount {
  void RefreshData() { /*...*/ }
  std::string Name;
  std::string Email;
};

void consume(Account& user) {
  user.RefreshData();
  UpdateUI(user.Name, user.Email);
}

void produce() {
  WebAccount user{ .Name = "Bob", .Email = "Bob@email.com" };
  consume(user);
}
```

A type alias can have specific constraints to control its construction, copiability, relocatability, etc.

```
typename NoCopyNoMove {
  NoCopyNoMove() = default;
  NoCopyNoMove(const NoCopyNoMove&) = delete;
  NoCopyNoMove(NoCopyNoMove&&) = delete;
};

void foo(NoCopyNoMove& a, NoCopyNoMove& b) {
  a = b; // compile error.
  a = std::move(b); // compile error.
}
```

A type alias can have function overloads.

```
typename Addition {
  void operator()() const;
  int operator()(int , int) const;
  float operator()(float, float) const;
};

void foo(const Addition& add) {
  add();
  add(1, 2);
  add(0.1f, 0.2f);
}
```

A type alias can be a template.

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

2.2 Library feature

```
namespace std {
template <class T, size_t MaxSize, size_t MaxAlign>
class poly_ptr;
} // namespace std

void foo() {
  {
    std::poly_ptr<Material> nouse;
    assert(!nouse.has_value()); // no value.
    nouse->apply(); // undefined behavior.
  }
  {
    Glass glass;
    {
      std::poly_ptr<Material> mat = &glass; // accepts a raw pointer.
      assert(dummy.has_value()); // contains value.
      mat->apply();
    }
    glass.apply(); // glass is still alive till here.
  }
  {
    auto color = std::make_unique<Color>(); // std::unique_ptr<Color>.
    std::poly_ptr<Material> mat = std::move(color); // accepts a smart ptr.
    assert(dummy.has_value()); // contains value.
    mat->apply();
```

```
  }
}
```

3 Motivation