

decltrait

Tian Liao

January 6, 2024

Document number:
Date: January 6, 2024
Audience:
Authors: Tian Liao, Mingxin Wang
Reply-to: Tian Liao <tilia@microsoft.com>

1 Introduction

2 Proposal

2.1 Language feature

decltrait-specifier:

```
decltrait ( class-name )  
decltrait ( class-name , expression )
```

where, *expression* must be such that it can be evaluated as a pointer whose type conforms to the public layout defined by the *class-name* type.

decltrait returns a value that represents a fancy pointer to a polymorphic type that matches the public layout of the *class-name* type.

decltrait with the same *class-names* specified deduces a same type generated by compiler, which is a fat pointer under the hood, and by dereference which, user can see all the public member functions and public non-static data members that the *class-name* type has.

Example:

```
struct Drawable { void print(); };  
struct Rectangle { void print() { std::println("Rectangle."); } };  
struct Circle { void print() { std::println("Circle."); } };  
  
void foo() {  
    auto poly_ptr = decltrait(Drawable);  
    assert(poly_ptr == nullptr); // empty because no target is assigned.  
  
    Rectangle rectangle;  
    poly_ptr = decltrait(Drawable, &rectangle);  
    poly_ptr->print(); // prints "Rectangle."  
  
    Circle circle;  
    poly_ptr = decltrait(Drawable, &circle);  
    poly_ptr->print(); // prints "Circle."  
  
    static_assert(std::is_same_v<  
        decltype(decltrait(Drawable, &rectangle)),  
        decltype(decltrait(Drawable, &circle))>); // true.  
}
```

For the example, the following implementation is simple and dirty hack that simulates a way to achieve *decltrait*:

```
#include <print>  
  
struct Drawable {  
    void print() = delete;  
    void resize(int) = delete;
```

```

};
struct Rectangle {
    void print() { std::println("Rectangle."); }
};
struct Circle {
    void print() { std::println("Circle."); }
};

namespace hack {

struct DrawableDynTrait {
    virtual void print() = 0;

protected:
    void* target;
};

struct Drawable_Rectangle : DrawableDynTrait {
    explicit Drawable_Rectangle(Rectangle* tgt) { target = tgt; }
    void print() override { static_cast<Rectangle*>(target)->print(); }
};

struct Drawable_Circle : DrawableDynTrait {
    explicit Drawable_Circle(Circle* tgt) { target = tgt; }
    void print() override { static_cast<Circle*>(target)->print(); }
};

template <class DynTrait>
class TraitPtr {
    friend struct HackFactory;

public:
    TraitPtr() : storage_{0} {}
    TraitPtr(const TraitPtr&) = default;
    TraitPtr(TraitPtr&&) = default;
    TraitPtr& operator=(const TraitPtr&) = default;
    TraitPtr& operator=(TraitPtr&&) = default;

    DynTrait* operator->()
    { return reinterpret_cast<DynTrait*>(storage_); }
    bool has_value() const {
        return reinterpret_cast<const void*>(storage_) != nullptr;
    }
    operator bool() const { return has_value(); }
    friend bool operator==(const TraitPtr& self, std::nullptr_t)
    { return !self; }

private:
    alignas(DynTrait) char storage_[sizeof(void*) * 2];
};

```

```

struct HackFactory {
    static auto DeclTrait_Drawable(Rectangle* target) {
        TraitPtr<DrawableDynTrait> ptr;
        new (ptr.storage_) Drawable_Rectangle{target}; // UB, but let's hack it
        return ptr;
    }

    static auto DeclTrait_Drawable(Circle* target) {
        TraitPtr<DrawableDynTrait> ptr;
        new (ptr.storage_) Drawable_Circle{target}; // UB, the same.
        return ptr;
    }
};

} // namespace hack

int main() {
    Rectangle rect;
    Circle circle;
    // auto trait = decltrait(Drawable, &rect);
    auto trait = hack::HackFactory::DeclTrait_Drawable(&rect);
    trait->print(); // prints Rectangle.

    // trait = decltrait(Drawable, &circle);
    trait = hack::HackFactory::DeclTrait_Drawable(&circle);
    trait->print(); // prints Circle.
}

```

3 Motivation