Virtual deducing `this`

Document #:	P3469R0
Date:	2024-10-16
Project:	Programming Language C++
Audience:	Evolution

Reply-to:	Mike Spertus <mike@enklu.com>>

Contents

1 Introduction

In C++23, member functions with explicit object parameters are not allowed to be `virtual`. This paper removes that restriction and shows how it can ease common use cases such as cloning and the visitor pattern.

C++23's deducing `this` (P0847) has of course been very successful at addressing its motivating use case of simplifying the creation of "two or four overloads of the same member function for different const- or ref-qualifiers." In the spirit embodied by the saying "great language features solve many problems," we believe that allowing member functions with explicit object parameters to be `virtual` would expand its applicability to valuable use cases frequently encountered by typical programmers, such as cloning and the visitor pattern.

In addition to enabling valuable use cases, we believe this would simplify the current mental model that while explicit object member functions are non-static member functions (dcl.fct/7), they are the only non-static member functions that cannot be virtual (class.mfct.non.static/4). Indeed, the author of this paper notes that he found this inconsistency so unexpected that he nearly gave his students the assignment of automating cloning with deducing `this` before discovering at the last minute that it does not work. As such, we regard this as increasing the functionality of the language by removing restrictions rather than adding new features, which is always desirable.

2 Use Cases

2.1 Cloning

Polymorphic cloning is the ability to make a copy of an object that preserves the run-time type. Programmers typically implement it by repeatedly writing a `clone()` method for all of the classes in their hierararchy. This is cumbersome and error-prone, providing a lot of friction for a common need. A `clone()` member function with

explicit object parameter would dispense with all of this repetitive boilerplate, resulting in what we believe to be simpler, more reliable code with clearer intent.

Before

```
struct Animal {
  virtual Animal *clone() const {
    return new Animal(*this);
  }
};

struct Cat : public Animal
{
  virtual Cat *clone() const {
    return new Cat(*this);
  }
};

struct Cow : public Animal
{
  virtual Cow *clone() const {
    return new Cow(*this);
  }
}

struct SiameseCat : public Cat
{
  // Forgot to override clone
};

struct Dog : public Animal
{
  virtual Animal *clone() const {
    return new Animal(*this);
  } // Cut-and-paste error
};
```

After

```
struct Animal {
  virtual Animal *clone() const;

  template<typename Self>
  virtual Self *clone(this Self const &self) {
    return new Self(self);
  }
};

struct Cat : public Animal
{
};

struct Cow : public Animal
{
}

struct SiameseCat : public Cat
{
};

struct Dog : public Animal
{
};
```

2.2 Virtual deducing `this` and concepts

Concepts and virtual deducing `this` work naturally together to solve problems such as this common variation of the use case above, where the `Animal` hierarchy contains abstract classes that cannot be cloned

Before

```
struct Animal {
  virtual Animal *clone() const = 0;
  virtual string eats() const = 0;
};

struct Mammal : public Animal {
}

struct Cat : public Mammal {
  virtual Cat *clone() const {
    return new Cat(*this);
  }
  string eats() override { return "mice"; }
};

struct Cow : public Mammal {
  virtual Cow *clone() const {
    return new Cow(*this);
  }
  string eats() override { return "grass"; }
}

struct Dog : public Mammal {
  virtual Dog *clone() const {
    return new Dog(*this);
  }
    string eats() override { return "dog food"; }
}
```

After

```
struct Animal {
  virtual Animal *clone() const = 0;
  virtual string eats() const = 0;

  template<typename Self>
  requires (!is_abstract<Self>)
  virtual Self *clone(this Self const &self) {
    return new Self(self);
  }
};

struct Mammal : public Animal {
}

struct Cat : public Mammal {
    string eats() override { return "mice"; }
};

struct Cow : public Mammal {
    string eats() override { return "grass"; }
}

struct Dog : public Mammal {
    string eats() override { return "dog food"; }
};
```

2.3 Cloning - Now with value categories

The common implementation is not fully correct due to ignoring of value categories. For example, we would like to clone an rvalue by moving it. We find that the following insightful statement from P0847 retains its full value here:

> There are many cases where we need two or four overloads of the same member function for different const- or ref-qualifiers. More than that, there are likely additional cases where a class should have four overloads of a particular member function but, due to developer laziness, doesn't. We think that there are enough such cases to merit a better solution than simply "write it, write it again, then write it two more times."

Before

```
struct Animal {
  virtual Animal *clone() const {
    return new Animal(*this);
  }

  virtual Animal *clone() && {
    return new Animal(move(*this));
  }
};

struct Cat : public Animal {
  virtual Cat *clone() const {
    return new Cat(*this);
  }

  virtual Cat *clone() && {
    return new Cat(move(*this));
  }
};

struct Cow : public Animal {
  virtual Cow *clone() const {
    return new Cow(*this);
  }

  virtual Cow *clone() && {
    return new Cow(move(*this));
  }
}

struct SiameseCat : public Cat {
  virtual SiameseCat *clone() const {
    return new SiameseCat(*this);
  }

  virtual SiameseCat *clone() && {
    return new SiameseCat(move(*this));
  }
};
```

After

```
struct Animal {
  virtual Animal *clone() const;
  virtual Animal *clone() &&;

  template<typename Self>
  virtual decay_t<Self> *
  clone(this Self &&self) {
    return new decay_t<Self>(forward<Self>(self));
  }
};

struct Cat : public Animal {
};

struct Cow : public Animal {
}

struct SiameseCat : public Cat {
};
```

2.4 Making it reusable

Since C++ has a consistent notion of copying and moving, we can package cloning into a `Clonable` base class so that even users that are not conversant with deducing `this` can add clonability to their hierarchies.:

```
// Provided once and for all by a library
struct Clonable {
  virtual Clonable *clone() const = 0;
  virtual Clonable *clone() && = 0;

  template<typename Self>
  requires (!is_abstract<Self>)
```

```
  decay_t<Self> *
  clone(this Self &&self) override {
    return new decay_t<Self>(forward<Self>(self));
  }
};
```

This further eases providing clonability.

Before

```
struct Animal {
  virtual Animal *clone() const {
    return new Animal(*this);
  }

  virtual Animal *clone() && {
    return new Animal(move(*this));
  }
};

struct Cat : public Animal {
  virtual Cat *clone() const {
    return new Cat(*this);
  }

  virtual Cat *clone() && {
    return new Cat(move(*this));
  }
};

struct Cow : public Animal {
  virtual Cow *clone() const {
    return new Cow(*this);
  }

  virtual Cow *clone() && {
    return new Cow(move(*this));
  }
}

struct SiameseCat : public Cat {
  virtual SiameseCat *clone() const {
    return new SiameseCat(*this);
  }

  virtual SiameseCat *clone() && {
    return new SiameseCat(move(*this));
  }
};
```

After

```
struct Animal : public Clonable {
};

struct Cat : public Animal {
};

struct Cow : public Animal {
}

struct SiameseCat : public Cat {
};
```

If virtual deducing `this` becomes part of the language, we will propose `Clonable` to LEWG.

2.5 The Visitor Pattern

The Visitor Pattern is a common way for client code to specify polymorphic behavior without modifying the classes it is using.

For example, suppose we wish to use the above animal hierarchy in a game that involves collecting animals. In this game, each animal type will have a different point value. For example, a `Cow` will be worth 5 points while a `SiameseCat` will be worth 15 points. While adding a virtual `collectibleValue` method is one approach, that would not work for externally provided libraries, and even if it were possible, cluttering up a library with virtual functions that only apply to a single application breaks encapsulation. Indeed, the additional behavior may contain application-specific code itself.

For such reasons, the visitor pattern is very useful, and I tell my students that whenever they are creating a class hierarchy, they should strongly consider enabling the visitor pattern to support client customization. As with the `clone()` example above, removing the `virtual` prohibition in deducing `this` is exactly what is needed:

```
// Once-only visitor pattern definitions unchanged
struct AnimalVisitor {
    virtual void visit(Animal &) = 0;
    virtual void visit(Cat &) = 0;
    virtual void visit(Cow &) = 0;
    virtual void visit(SiameseCat &) = 0;
}

// Desired behavior by runtime-type
struct CollectibleValueVisitor : public AnimalVisitor {
    CollectibleValueVisitor(int &value) : value(value) {}
    void visit(Animal &value) { value = 1; } // Ordinary animal
    virtual void visit(Cat &) { value = 5; }
    virtual void visit(Cow &) { value = 5; }
    virtual void visit(SiameseCat &) { value = 15; }
    int &value;
}
```

Before

```
// Extensive per-class boilerplate
struct Animal : {
  virtual Animal *clone() const {
    return new Animal (*this);
  }
  virtual Animal *clone() && {
    return new Animal(move(*this));
  }

  virtual void accept(AnimalVisitor& v) {
    v.accept(*this);
  }
};

struct Cat : public Animal {
  virtual Cat *clone() const {
    return new Cat(*this);
  }
  virtual Cat *clone() && {
    return new Cat(move(*this));
  }

  virtual void accept(AnimalVisitor& v) {
    v.accept(*this);
  }
};

struct Cow : public Animal {
  virtual Cow *clone() const {
    return new Cow(*this);
  }
  virtual Cow *clone() && {
    return new Cow(move(*this));
  }
  // Oops, forgot! Cow inadvertently gets
  // 1 point rather than the intended 5
}

struct SiameseCat : public Cat {
  virtual SiameseCat *clone() const {
    return new SiameseCat(*this);
  }

  virtual SiameseCat *clone() && {
    return new SiameseCat(move(*this));
  }

  virtual void accept(AnimalVisitor& v) {
    v.accept(*this);
  }
};
```

After

```
// No per-class boilerplate
struct Animal : public Clonable {
  virtual void accept(AnimalVisitor& v);

  template<typename Self>
  virtual void accept(this Self &self
                      AnimalVisitor &v) {
    v.accept(self);
  }
};

struct Cat : public Animal {
};

struct Cow : public Animal {
}

struct SiameseCat : public Cat {
};
```

3 Instantiation Triggers

The basic rule is that a virtual method template with explicit object parameter is triggered whenever it would define a virtual function that has been declared for that class (including virtual method declarations that can be overriden from `public` or `protected` base classes).

```
struct S {
  virtual void foo();
  template<typename Self>
  virtual void foo(this Self &self) { ... };
};
```

In the above, the second appearance of `virtual` is optional for consistency with existing overriding, but we would encourage our student to put in either `virtual` or `override` (Note that we are inclined to allow this use of `override` since it implements an existing declaration even though it is technically only an override in classes that derive from `S`).

If the declaration is not compatible with having a definition, then, as expected, none will be generated. In the following, `foo` is defined in `T` but remains pure virtual in `S`.

```
struct S {
  virtual void foo() = 0;

  template<typename Self>
  void foo(this Self &self) override { ... };
};

struct T : public S {
};
```

If a program defines a member function that would also be defined by an explicit object member function, we would like that to be regarded as ill-formed, no diagnostic required because the compiler has no way of telling whether the declaration will be defined in a different translation unit.

4 Comparison to P0847

The original deducing `this` proposal also discusses allowing `virtual`, describing it as a "maybe" but goes on to say that

> such a direction also doesn't provide the user with any ability that they didn't have before. It would purely be a style choice. As such, we don't consider the question of allowing virtual to be especially important at this time.

The use cases presented above are meant to make the case that this feature would provide the user with new abilities of practical value to justify revisiting the question. Our proposal is intended to conform to what that paper refers to as "same kind" overriding to mitigate technical issues. See the discussion there for further details.

P0847's discussion includes non-template virtual deducing `this`. While we did not discuss this above, our inclination would be to allow same-kind overriding in that case as well so that class.mfct.non.static could be modified to the consistent

> ~~An implicit object~~ member function may be declared virtual (11.7.3) or pure virtual (11.7.4).

eliminating the need for the user to memorize special-case exceptions.