# Dynamic Polymorphism With Code Injection and Metaclasses

**Sy Brand** (they/them), @TartanLlama
C++ Developer Advocate at Microsoft

# Dynamic Polymorphism

# Polymorphism:

The provision of a single interface to entities of different types

## Dynamic Polymorphism

## Static Polymorphism

## Dynamic Polymorphism

- Run-time

## Static Polymorphism

- Compile-time

## Dynamic Polymorphism

- Run-time
- Different behaviour based on dynamic type

## Static Polymorphism

- Compile-time
- Different behaviour based on static type

## Dynamic Polymorphism

- Run-time
- Different behaviour based on dynamic type
- Typically implemented with inheritance

## Static Polymorphism

- Compile-time
- Different behaviour based on static type
- Typically implemented with overloading and templates

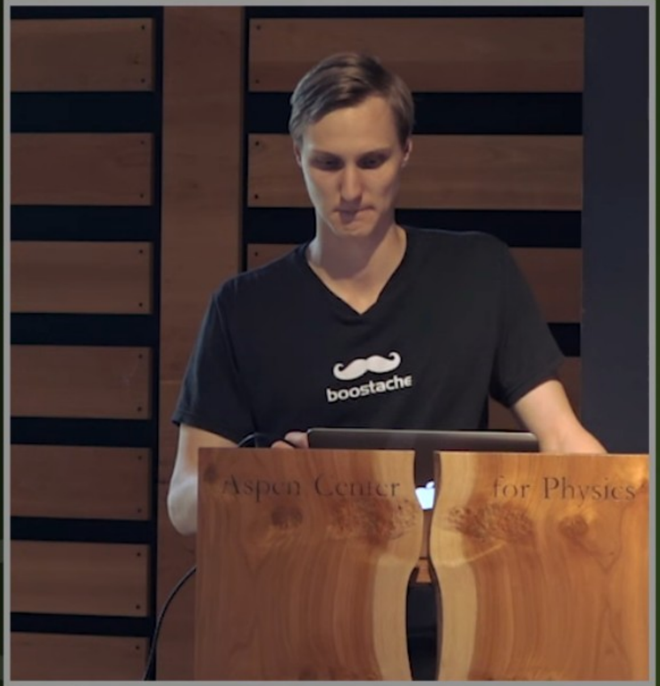
Louis Dionne

Runtime Polymorphism:
Back to Basics

# RUNTIME POLYMORPHISM: BACK TO THE BASICS

LOUIS DIONNE, C++NOW 2018

C++ now 2018
MAY 7-11
cppnow.org




**Louis Dionne**

Runtime Polymorphism:
Back to Basics

Video Sponsorship Provided By: JET BRAINS

# LISTEN TO SEAN PARENT, NOT ME

https://youtu.be/QGcVXgEVMJg

19

# Inheritance Is The Base Class of Evil

Sean Parent | Principal Scientist

# Problems with inheritance



- Often requires dynamic allocation

# Dynamic Allocation



```cpp
struct base{};
struct a : base{};
struct b : base{};

base make_base();

std::vector<base> v;
```

# Dynamic Allocation



```cpp
struct base{};
struct a : base{};
struct b : base{};

std::unique_ptr<base> make_base();

std::vector<std::unique_ptr<base>> v;
```

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes

# Intrusive Polymorphism



```
namespace mylib {
  struct base {
    virtual void do_thing();
  };
}

namespace otherlib {
  struct x {
    virtual void do_thing();
  };
}
```

# Intrusive Polymorphism



```
namespace mylib {
  struct base {
    virtual void do_thing();
  };
}

namespace otherlib {
  struct x {
    virtual void do_thing();
  };
}

mylib::base* b = new otherlib::x;
```

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics

# Problems with inheritance

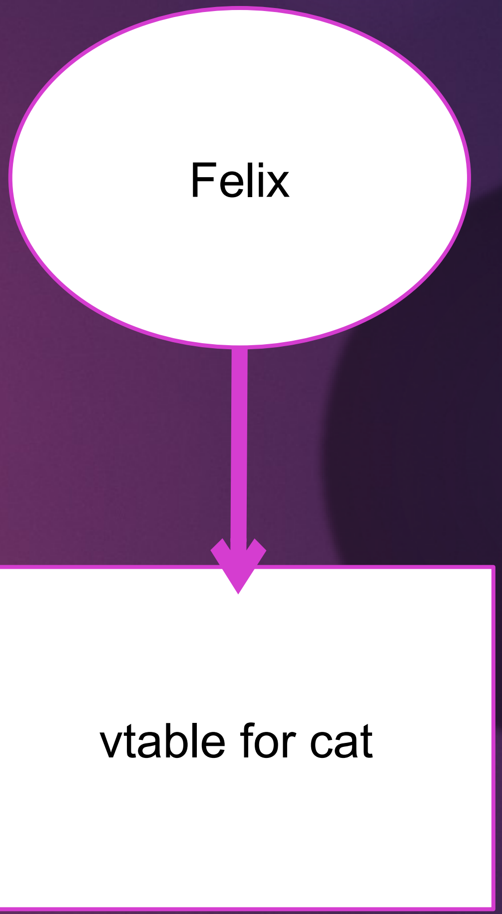


- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

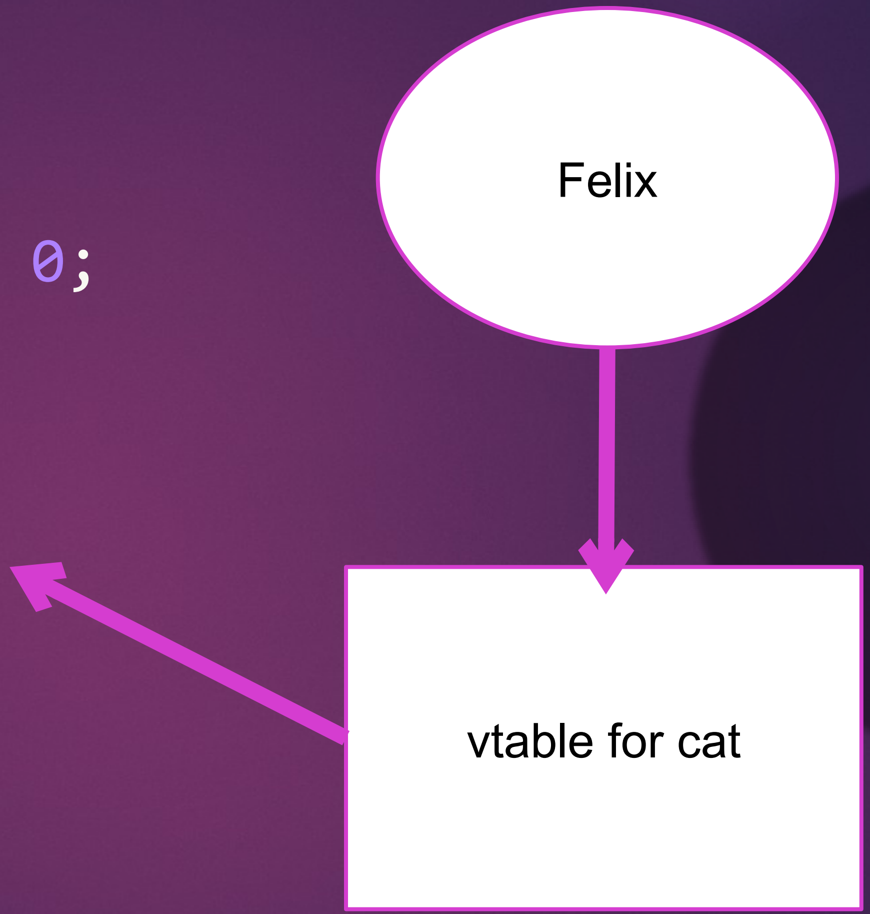
```cpp
struct animal {
  virtual ~animal();
  virtual void speak() = 0;
};

struct cat : animal {
  void speak() override;
};

struct dog : animal {
  void speak() override;
};
```

```cpp
struct animal {
  virtual ~animal();
  virtual void speak() = 0;
};

struct cat : animal {
  void speak() override;
};

struct dog : animal {
  void speak() override;
};
```

```
struct animal {
  virtual ~animal();
  virtual void speak() = 0;
};

struct cat : animal {
  void speak() override;
};

struct dog : animal {
  void speak() override;
};
```



```
struct animal {
  virtual ~animal();
  virtual void speak() = 0;
};

struct cat : animal {
  void speak() override;
};

struct dog : animal {
  void speak() override;
};
```



```
struct animal {
  // magic
};

struct cat {
  void speak();
};

struct dog {
  void speak();
};
```

```cpp
struct animal {
  // magic
};

struct cat {
  void speak();
};

struct dog {
  void speak();
};
```

```cpp
int main() {
  animal c = cat{};
  c.speak();

  animal d = dog{};
  d.speak();
}
```

# Hand-written virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```cpp
struct animal {
  virtual ~animal();
  virtual void speak() = 0;
};

struct cat : animal {
  void speak() override;
};

struct dog : animal {
  void speak() override;
};
```

# Hand-written virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
struct vtable {
   void (*speak)(void* ptr);
   void (*destroy_)(void* ptr);
};
```

# Hand-written virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
template<class Concrete>
constexpr vtable vtable_for {
 // function which calls speak
 // function which deletes object
};
```

```
template<class Concrete>
constexpr vtable vtable_for {
  [] (void* ptr) { static_cast<Concrete*>(ptr)->speak(); },
  [] (void* ptr) { delete static_cast<Concrete*>(ptr); }
};
```

# Hand-written virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
struct animal {
  void* concrete_;
  vtable const* vtable_;




};
```

```
struct animal {
  void* concrete_;
  vtable const* vtable_;

  template<class T>
  animal(T const& t) :
    concrete_(new T(t)),
    vtable_(&vtable_for<T>)
  {}
};
```

# Hand-written virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
struct animal {
  // ...

  void speak() { vtable_->speak(t_); }
  ~animal() { vtable_->destroy_(t_); }
};
```

```
struct cat {
  void speak();
};

struct dog {
  void speak();
};
```

```
struct cat {
  void speak();
};

struct dog {
  void speak();
};

int main() {
  animal c = cat{};
  c.speak();

  animal d = dog{};
  d.speak();
}
```

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

```
struct vtable {
  void (*speak)(void* ptr);
  void (*destroy_)(void* ptr);
  void* (*clone_)(void* ptr);
  void* (*move_clone_)(void* ptr);
};
```

```
struct vtable {
  void (*speak)(void* ptr);
  void (*destroy_)(void* ptr);
  void* (*clone_)(void* ptr);
  void* (*move_clone_)(void* ptr);
};

template<class T>
constexpr vtable vtable_for {
  [] (void* ptr) { static_cast<T*>(ptr)->speak(); },
  [] (void* ptr) { delete static_cast<T*>(ptr); },
  [] (void* ptr) -> void*
    { return new T(*static_cast<T*>(ptr)); },
  [] (void* ptr) -> void*
    { return new T(std::move(*static_cast<T*>(ptr))); }
};
```

```
struct animal {
  //...

  animal(animal const& rhs) :
    t_(rhs.vtable_->clone_(rhs.t_)),
    vtable_(rhs.vtable_)
  {}

  animal(animal&& rhs) :
    t_(rhs.vtable_->move_clone_(rhs.t_)),
    vtable_(rhs.vtable_)
  {}
};
```

```
int main() {
  animal a = cat{};
  a.speak();
  a = dog{};
  a.speak();

  animal b = a;
  b.speak();
}
```

```cpp
int main() {
  std::vector<animal> animals { cat{}, dog{} };

  for (auto&& a : animals) {
    a.speak();
  }
}
```

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

```
struct vtable {
  void (*speak)(void* ptr);
  void (*destroy_)(void* ptr);
  void* (*clone_)(void* ptr);
  void* (*move_clone_)(void* ptr);
};

template<class T>
constexpr vtable vtable_for {
  [] (void* ptr) { static_cast<T*>(ptr)->speak(); },
  [] (void* ptr) { delete static_cast<T*>(ptr); },
  [] (void* ptr) -> void*
    { return new T(*static_cast<T*>(ptr)); },
  [] (void* ptr) -> void*
    { return new T(std::move(*static_cast<T*>(ptr))); }
};

struct animal {
  void* t_;
  vtable const* vtable_;

  void speak() { vtable_->speak(t_); }
  ~animal() { vtable_->destroy_(t_); }

  template<class T>
  animal(T const& t) :
    t_(new T(t)),
    vtable_(&vtable_for<T>)
  {}

 animal(animal const& rhs) :
    t_(rhs.vtable_->clone_(rhs.t_)),
    vtable_(rhs.vtable_)
  {}

  animal(animal&& rhs) :
    t_(rhs.vtable_->move_clone_(rhs.t_)),
    vtable_(rhs.vtable_)
  {}

  animal& operator=(animal const& rhs) {
    t_ = rhs.vtable_->clone_(rhs.t_);
    vtable_ = rhs.vtable_; return *this;
  }

  animal& operator=(animal&& rhs) {
    t_ = rhs.vtable_->move_clone_(rhs.t_);
    vtable_ = rhs.vtable_; return *this;
  }
};
```

# Static Reflection

Reflection:

The ability of a program to introspect its own structure

# Static Reflection:

The ability of a program to introspect its own structure at compile-time `std::is_array<T>`

`std::is_same<T, U>`

Enum to string?
Iterate over members?

Document: P1240R1
Revises: P1240R0
Date: 10-03-2019
Audience: SG7

Authors: Wyatt Childers (wchilders@lock3software.com)
Andrew Sutton (asutton@uakron.edu)
Faisal Vali (faisalv@yahoo.com)
Daveed Vandevoorde (daveed@edg.com)

# Scalable Reflection in C++

```
template<Enum T>
constexpr std::string to_string(T value) {




}
```

```
template<Enum T>
constexpr std::string to_string(T value) {
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {



  }

}
```

```
template<Enum T>
constexpr std::string to_string(T value) {
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {



  }

}
```

```
template<Enum T>
constexpr std::string to_string(T value) {
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {



  }

}
```

```
template<Enum T>
constexpr std::string to_string(T value) {
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {
    if (exprid(e) == value) {

    }
  }

}
```

```
template<Enum T>
constexpr std::string to_string(T value) {
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {
    if (exprid(e) == value) {

    }
  }

}
```

```
template<Enum T>
constexpr std::string to_string(T value) {
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {
    if (exprid(e) == value) {
      return std::meta::name_of(e);
    }
  }

}
```

```
template<Enum T>
constexpr std::string to_string(T value) {
  for constexpr (auto e : std::meta::members_of(reflexpr(T)) {
    if (exprid(e) == value) {
      return std::meta::name_of(e);
    }
  }
  return "<unnamed>";
}
```

# Reifiers

# Reifiers



- typename(reflection): reflection -> type

# Reifiers



- typename(reflection): reflection -> type
- namespace(reflection): reflection -> namespace

# Reifiers



- typename(reflection): reflection -> type
- namespace(reflection): reflection -> namespace
- template(reflection): reflection -> template

# Reifiers



- typename(reflection): reflection -> type
- namespace(reflection): reflection -> namespace
- template(reflection): reflection -> template
- valueof(reflection): reflection -> value, so long as the reflection designates a constant expression

# Reifiers



- typename(reflection): reflection -> type
- namespace(reflection): reflection -> namespace
- template(reflection): reflection -> template
- valueof(reflection): reflection -> value, so long as the reflection designates a constant expression
- exprid(reflection): reflection -> id-expression representing the entity

# Reifiers



- typename(reflection): reflection -> type
- namespace(reflection): reflection -> namespace
- template(reflection): reflection -> template
- valueof(reflection): reflection -> value, so long as the reflection designates a constant expression
- exprid(reflection): reflection -> id-expression representing the entity
- [: reflection :] or unqualid(reflection): reflection -> identifier

# Reifiers



- typename(reflection): reflection -> type
- namespace(reflection): reflection -> namespace
- template(reflection): reflection -> template
- valueof(reflection): reflection -> value, so long as the reflection designates a constant expression
- exprid(reflection): reflection -> id-expression representing the entity
- [: reflection :] or unqualid(reflection): reflection -> identifier
- [< reflection >] or templarg(reflection): reflection -> template argument

# Code Injection

```
class point {
  int x;
  int y;
};
```

```
class point {
  int x;
  int y;

public:
  int get_x() const { return x; }
  int get_y() const { return y; }
};
```

```
class point {
  int x;
  int y;

  consteval {
    generate_getters(reflexpr(point));
  }
};
```

```
consteval void generate_getters(meta::info cls) {
  for (auto member : meta::members_of(cls)) {
    if (meta::is_nonstatic_data_member(member)) {
      generate_getter(member);
    }
  }
}
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {



  };
}
```

```
class point {
  int x;
  int y;

  consteval {
    generate_getters(reflexpr(point));
  }
};
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {




  };
}
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {
    int const&
    get_x() const {

    }
  };
}
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {
    int const&
    get_x() const {

    }
  };
}
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {
    typename(meta::type_of(%{member})) const&
    get_x() const {

    }
  };
}
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {
    typename(meta::type_of(%{member})) const&
    get_x() const {

    }
  };
}
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {
    typename(meta::type_of(%{member})) const&
    unqualid("get_", %{member})() const {

    }
  };
}
```

```
consteval void generate_getter(meta::info member) {
  -> fragment struct {
    typename(meta::type_of(%{member})) const&
    unqualid("get_", %{member})() const {
      return this->idexpr(%{member});
    }
  };
}
```

```
class point {
  int x;
  int y;

  consteval {
    generate_getters(reflexpr(point));
  }
};
```

```
class point {
  int x;
  int y;

public:
  int const& get_x() const { return x; }
  int const& get_y() const { return y; }
};
```

# Code Injection + Dynamic Polymorphism

```
struct animal {
  void* concrete_;
  vtable const* vtable_;

  // ctors, forwarding functions, etc.
};
```

```cpp
template <class Facade>
struct typeclass_for {
  void* concrete_;
  vtable<Facade> const* vtable_;

  // ctors, forwarding functions, etc.
};
```

```
template <class Facade>
struct typeclass_for {
  void* concrete_;
  vtable<Facade> const* vtable_;

  // ctors, forwarding functions, etc.
};

struct animal_facade {
  void speak();
};
```

```cpp
template <class Facade>
struct typeclass_for {
  void* concrete_;
  vtable<Facade> const* vtable_;

  // ctors, forwarding functions, etc.
};

struct animal_facade {
  void speak();
};

using animal = typeclass_for<animal_facade>;
```

# Code-injected virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

# Code-injected virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
struct vtable {
  void (*speak)(void* ptr);
  void (*destroy_)(void* ptr);
  void* (*clone_)(void* ptr);
  void* (*move_clone_)(void* ptr);
};
```

```
struct vtable {
  void (*speak)(void* ptr);
```

```
template <class Facade>
struct vtable {
  void (*speak)(void* ptr);
```

```
template <class Facade>
struct vtable {
  consteval {






  }
```

```
template <class Facade>
struct vtable {
  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) constexpr {



    });
  }
```

```
template <class Facade>
struct vtable {
  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) constexpr {
        -> fragment struct {
          void (*speak) (void* ptr);
        };
    });
  }
```

```
template <class Facade>
struct vtable {
  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) constexpr {
        -> fragment struct {
          void (*speak) (void* ptr);
        };
    });
  }
```

```
template <class Facade>
struct vtable {
  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) constexpr {
        -> fragment struct {
          typename(%{ret}) (*speak) (void* ptr);
        };
    });
  }
```

```
template <class Facade>
struct vtable {
  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) constexpr {
        -> fragment struct {
          typename(%{ret}) (*speak) (void* ptr);
        };
    });
  }
```

```
template <class Facade>
struct vtable {
  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) constexpr {
        -> fragment struct {
          typename(%{ret}) (*unqualid(%{func})) (void* ptr);
        };
    });
  }
```

```
template <class Facade>
struct vtable {
  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) constexpr {
        -> fragment struct {
          typename(%{ret}) (*unqualid(%{func})) (void* ptr,
                                      ->%{params});
        };
    });
  }
```

# Code-injected virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
vtable<Facade> table;

table.speak = [] (void* ptr) {
  static_cast<T*>(ptr)->speak();
};
```

```
table.speak = [] (void* ptr) {
  static_cast<T*>(ptr)->speak();
};
```

```
-> fragment {
  table.speak = [] (void* ptr) {
    static_cast<T*>(ptr)->speak();
  };
};
```

```
-> fragment {
  table.unqualid(%{func}) = [] (void* ptr) {
    static_cast<T*>(ptr)->unqualid(%{func})();
  };
};
```

```
-> fragment {
  table.unqualid(%{func}) = [] (void* ptr, ->%{params}) {
    static_cast<T*>(ptr)->unqualid(%{func})(unqualid(... %{params}));
  };
};
```

```
-> fragment {
  table.unqualid(%{func}) = [] (void* ptr, ->%{params}) {
    return static_cast<T*>(ptr)->unqualid(%{func})
                (unqualid(... %{params}));
  };
};
```

# Code-injected virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
template<class Facade>
struct typeclass_for {
  vtable<Facade> const* vtable_;
  void* concrete_;




};
```

```
template<class Facade>
struct typeclass_for {
  vtable<Facade> const* vtable_;
  void* concrete_;

  template<class T>
  typeclass_for(T const& t) :
    concrete_(new T(t)),
    vtable_(&vtable_for<Facade, T>)
  {}
};
```

# Code-injected virtual functions



- Declare vtable for the abstract interface
- Define vtable for a concrete type
- Capture the vtable pointers on construction
- Forward calls through the vtable

```
template<class Facade>
struct typeclass_for {

  consteval {
    for_each_declared_function(reflexpr(Facade),
      [](auto func, auto ret, auto params) consteval {
      -> fragment struct {
        typename(%{ret}) unqualid(%{func}) (->%{params}) {
          return this->vtable_->unqualid(%{func})
            (this->concrete_, unqualid(... %{params}));
        }
      };
    });
  }
};
```

```cpp
struct vtable {
  void (*speak)(void* ptr);
  void (*destroy_)(void* ptr);
  void* (*clone_)(void* ptr);
  void* (*move_clone_)(void* ptr);
};

template<class T>
constexpr vtable vtable_for {
  [] (void* ptr) { static_cast<T*>(ptr)->speak(); },
  [] (void* ptr) { delete static_cast<T*>(ptr); },
  [] (void* ptr) -> void*
    { return new T(*static_cast<T*>(ptr)); },
  [] (void* ptr) -> void*
    { return new T(std::move(*static_cast<T*>(ptr))); }
};

struct animal {
  void* t_;
  vtable const* vtable_;

  void speak() { vtable_->speak(t_); }
  ~animal() { vtable_->destroy_(t_); }

  template<class T>
  animal(T const& t) :
    t_(new T(t)),
    vtable_(&vtable_for<T>)
  {}

 animal(animal const& rhs) :
    t_(rhs.vtable_->clone_(rhs.t_)),
    vtable_(rhs.vtable_)
  {}

  animal(animal&& rhs) :
    t_(rhs.vtable_->move_clone_(rhs.t_)),
    vtable_(rhs.vtable_)
  {}

  animal& operator=(animal const& rhs) {
    t_ = rhs.vtable_->clone_(rhs.t_);
    vtable_ = rhs.vtable_; return *this;
  }

  animal& operator=(animal&& rhs) {
    t_ = rhs.vtable_->move_clone_(rhs.t_);
    vtable_ = rhs.vtable_; return *this;
  }
};
```

```
struct animal_facade {
  void speak();
};

using animal = typeclass_for<animal_facade>;
```

```
template <class Facade>
struct typeclass_for {
  void* concrete_;
  vtable<Facade> const* vtable_;

  // ctors, forwarding functions, etc.
};
```

```
template <class Facade, class StoragePolicy>
struct typeclass_for {
  StoragePolicy storage_;
  vtable<Facade> const* vtable_;

  // ctors, forwarding functions, etc.
};
```

```cpp
template <class Facade, class StoragePolicy, class VTablePolicy>
struct typeclass_for {
  StoragePolicy storage_;
  VTablePolicy::vtable_for<Facade> vtable;

  // ctors, forwarding functions, etc.
};
```

```
typeclass_for<animal_facade> an_animal;
```

```
typeclass_for<animal_facade> an_animal;

typeclass_for<animal_facade, remote_storage, remote_vtable> a1;
```

```
typeclass_for<animal_facade> an_animal;

typeclass_for<animal_facade, remote_storage, remote_vtable> a1;
typeclass_for<animal_facade,
        sbo_storage<32>, in_place_vtable> a2;
```

```
typeclass_for<animal_facade> an_animal;

typeclass_for<animal_facade, remote_storage, remote_vtable> a1;
typeclass_for<animal_facade,
              sbo_storage<32>, in_place_vtable> a2;

typeclass_for<animal_facade, in_place_storage,
        split_vtable<local <reflexpr(animal::speak)>,
                     remote<reflexpr(animal::walk)>> a3;
```

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

# Problems with inheritance



- Often requires dynamic allocation
- Ownership and nullability considerations
- Intrusive: requires modifying child classes
- No more value semantics
- Changes semantics for algorithms and containers

```
struct animal_facade {
  void speak();
};

using animal = typeclass_for<animal_facade>;
```

```
class(typeclass) animal {
  void speak();
};
```

# Metaclasses

Metaclass functions let programmers write a new kind of efficient abstraction: a user-defined named subset of classes that share common characteristics, typically (but not limited to):

- user-defined rules
- defaults, and
- generated functions

```
class point {
  int x;
  int y;
};
```

```
class(value) point {
  int x;
  int y;
};
```

```
consteval void value (meta::info source) {
  //injection statements
}
```

```
class(value) point {
  int x;
  int y;
};
```

```
class(value) point {
  int x;
  int y;
};
```

```
namespace __hidden {
  struct prototype {
    int x;
    int y;
  };
}
```

```
class(value) point {
  int x;
  int y;
};
```

```
namespace __hidden {
  struct prototype {
    int x;
    int y;
  };
}

struct point {
  using prototype = __hidden::prototype;
  consteval {
    value(reflexpr(prototype));
  }
};
```

```
class Shape {
public:
   virtual int area() const =0;
   virtual void scale_by(double factor) =0;
   // ... etc. virtual ~Shape() noexcept { };

   // be careful not to write nonpublic/nonvirtual function
}; // or copy/move function or data member; no enforcement
```

```
class(interface) Shape {
  int area() const;
  void scale_by(double factor);
};
```

```
consteval void interface(info source) {


};
```

```
consteval void interface(info source) {
  bool has_dtor = false;
  for (auto mem : range(source)) {








  }



};
```

```
consteval void interface(info source) {
  bool has_dtor = false;
  for (auto mem : range(source)) {
    compiler_require(!is_data_member(mem),
                     "interfaces may not contain data");
    compiler_require(!is_copy(mem) && !is_move(mem),
                     "interfaces may not copy or move; consider "
                     "a virtual clone() instead");







  }



};
```

```
consteval void interface(info source) {
  bool has_dtor = false;
  for (auto mem : range(source)) {
    compiler_require(!is_data_member(mem),
                     "interfaces may not contain data");
    compiler_require(!is_copy(mem) && !is_move(mem),
                     "interfaces may not copy or move; consider "
                     "a virtual clone() instead");
    if (has_default_access(mem)) make_public(mem);
    make_pure_virtual(mem);
    compiler_require(is_public(mem),
                     "interface functions must be public");
    has_dtor |= is_destructor(mem);

  }


};
```

```
consteval void interface(info source) {
  bool has_dtor = false;
  for (auto mem : range(source)) {
    compiler_require(!is_data_member(mem),
                     "interfaces may not contain data");
    compiler_require(!is_copy(mem) && !is_move(mem),
                     "interfaces may not copy or move; consider "
                     "a virtual clone() instead");
    if (has_default_access(mem)) make_public(mem);
    make_pure_virtual(mem);
    compiler_require(is_public(mem),
                     "interface functions must be public");
    has_dtor |= is_destructor(mem);
    -> mem;
  }


};
```

```
consteval void interface(info source) {
  bool has_dtor = false;
  for (auto mem : range(source)) {
    compiler_require(!is_data_member(mem),
                     "interfaces may not contain data");
    compiler_require(!is_copy(mem) && !is_move(mem),
                     "interfaces may not copy or move; consider "
                     "a virtual clone() instead");
    if (has_default_access(mem)) make_public(mem);
    make_pure_virtual(mem);
    compiler_require(is_public(mem),
                     "interface functions must be public");
    has_dtor |= is_destructor(mem);
    -> mem;
  }
  if (!has_dtor) -> __fragment struct Z {
    virtual ~X() noexcept {}
  };
};
```

```
consteval void typeclass (meta::info source) {
  //injection statements
}
```

```
class(typeclass) animal {
  void speak();
};

std::vector<animal> animals;
```

# Concerns



- Runtime performance
- Compile-time performance
- New and incompatible way of specifying concepts

# Future Work



**PFA: A Generic, Extendable and Efficient Solution for Polymorphic Programming**

| | |
|---|---|
| Document number: | P0957R4 |
| Date: | 2019-11-04 |
| Project: | Programming Language C++ |
| Audience: | LEWG, LWG, SG7 |
| Authors: | Mingxin Wang |
| Reply-to: | Mingxin Wang <mingxwa@microsoft.com> |

- https://wg21.link/p0957

# Future Wark

## We need a language mechanism for customization points

| | |
|---|---|
| Document #: | P2279R0 |
| Date: | 2021-01-15 |
| Project: | Programming Language C++ |
| Audience: | EWG |
| Reply-to: | Barry Revzin <barry.revzin@gmail.com> |

- https://wg21.link/p2279

# Links



- Code for this talk: https://godbolt.org/z/McT4Tb
- Prototype implementation: https://github.com/TartanLlama/typeclasses
- Experimental compiler: https://github.com/lock3/meta
- On Compiler Explorer: https://cppx.godbolt.org/
- Scalable Reflection in C++ Paper: https://wg21.link/P1240
- Metaclasses paper: https://wg21.link/P0707