Polymorphism != Virtual

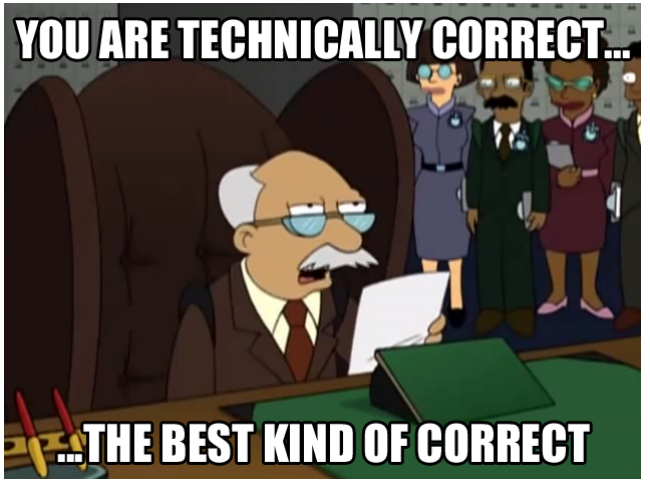
Flexible Runtime Polymorphism Without Inheritance

John R. Bandela, MD

What is polymorphism

What is polymorphism

- "Many Forms"


YOU ARE TECHNICALLY CORRECT...
...THE BEST KIND OF CORRECT

What is polymorphism?

What is polymorphism?

Providing a single interface to entities of different types

What is polymorphism?

Providing a single interface to entities of different types

- Bjarne Stroustrup

Polymorphism predated OOP

Polymorphism predated OOP

A + B

Polymorphism predated OOP

A + B

- Integers

Polymorphism predated OOP

A + B

- Integers
- Reals

Polymorphism predated OOP

A + B

- Integers
- Reals
- Complex numbers

Polymorphism predated OOP

A + B

- Integers
- Reals
- Complex numbers
- Vectors

Polymorphism predated OOP

A + B

- Integers
- Reals
- Complex numbers
- Vectors
- Matrices

Types of polymorphism

Types of polymorphism

- Static polymorphism

Types of polymorphism

- Static polymorphism
- Dynamic polymorphism

Static polymorphism

Static polymorphism

Overloading

Dynamic polymorphism

Dynamic polymorphism

Virtual Functions

What about variant?

What about variant?

Variant

- Types - Closed set
- Operations - Open set

What about variant?

Variant

- Types - Closed set
- Operations - Open set

Polymorphism in this talk

- Types - Open set
- Operations - Closed set

Motivating example

Motivating example

Shape

Motivating example

Shape

- Draw

Motivating example

Shape

- Draw
- Get Bounding Box

Motivating example

Shape

- Draw
- Get Bounding Box
- Translate

Motivating example

Shape

- Draw
- Get Bounding Box
- Translate
- Rotate

Overloading

Overloading

- Determines at compile time which function to call based on parameters

Overloading

- Determines at compile time which function to call based on parameters
- Many, many rules, but in general it just works

Overloading

- Determines at compile time which function to call based on parameters
- Many, many rules, but in general it just works
- No runtime overhead

Overloading example

```
class circle{
public:
  void draw() const;
  box get_bounding_box() const;
  void translate(double x, double y);
  void rotate(double degrees);

};
```

Overloading example

```
class box{
public:
  point upper_left()const;
  void set_upper_left(point);
  point lower_right()const;
  void set_lower_right(point);
  bool overlaps(const box&)const;

};
```

Overloading example

```
// Draw only if the shape will be visible.
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Overloading example

```
// Move and rotate
template<typename Shape>
void spin(const Shape& s){
  s.translate(4,2);
  s.rotate(45);
}
```

How do we use this

```
cirlce c{...};
spin(c);
smart_draw(c,viewport);
```

What about box

```
circle c;
smart_draw(c, viewport);
```

What about box

```
circle c;
smart_draw(c, viewport);
```

What about box

```
box b;
smart_draw(b,viewport);
```

What about box

```
box b;
smart_draw(b,viewport);
```

What to do

Write a facade?

What to do

Write a facade?

Use overloaded customization points

Use overloaded customization points

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Use overloaded customization points

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Use overloaded customization points

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(s);
}
```

Use overloaded customization points

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(s);
}
```

Provide default implementation

```
template<typename Shape>
void draw(const Shape& s){
 s.draw();
}
```

Use overloaded customization points

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(s);
}
```

Provide default implementation

```
template<typename Shape>
void draw(const Shape& s){
 s.draw();
}
```

Provide box implementation

```
void draw(const box& b){
// Draw box.
}
```

Smart Draw

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(x);
}
```

Spin

```
// Move and rotate
template<typename Shape>
void spin(const Shape& s){
  translate(s,4,2);
  rotate(s,45);
}
```

Use

```
cirlce c{...};
spin(c);
smart_draw(c,viewport);
```

Use

```
cirlce c{...};
spin(c);
smart_draw(c,viewport);
```

```
box b{...};
spin(b);
smart_draw(b,viewport);
```

Value semantics

```
circle c1;
```

Value semantics

```
circle c1;
```

```
circle c2 = c1;
```

Value semantics

```
circle c1;
```

```
circle c2 = c1;
```

```
spin(c1);
```

Value semantics

```
circle c1;
```

```
circle c2 = c1;
```

```
spin(c1);
```

```
smart_draw(c1);
smart_draw(c2);
```

Low Coupling

Low Coupling

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(x);
}
```

Low Coupling

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(x);
}
```

What operations does this depend on?

Low Coupling

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(x);
}
```

What operations does this depend on?

- Get Bounding Box
- Draw

PPP

PPP

Purchasing Power Parity?

PPP

Purchasing Power Parity?

Parent's Polymorphic Principles (PPP)

Parent's Polymorphic Principles (PPP)

- The requirement of a polymorphic type, by definition, comes from its use

Parent's Polymorphic Principles (PPP)

- The requirement of a polymorphic type, by definition, comes from its use
- There are no polymorphic types, only a polymorphic use of similar types

Parent's Polymorphic Principles (PPP)

- The requirement of a polymorphic type, by definition, comes from its use
- There are no polymorphic types, only a polymorphic use of similar types

From "Inheritance is the base class of evil" by Sean Parent

PPP

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(x);
}
```

PPP

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(x);
}
```

What has static polymorphism ever done for us?

What has static polymorphism ever done for us?

- ☑ Low boilerplate

What has static polymorphism ever done for us?

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class

What has static polymorphism ever done for us?

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics

What has static polymorphism ever done for us?

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling

What has static polymorphism ever done for us?

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP

What has static polymorphism ever done for us?

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ Performance

What has static polymorphism ever done for us?

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling
- ✅ PPP
- ✅ Performance
  - Obviously, performance. I mean, performance goes without saying.

Why don't we just stop now?

Why don't we just stop now?

- ✕ Requires everything to be a template and thus live in headers

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
```

Why don't we just stop now?

- ✕ Requires everything to be a template and thus live in headers

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
```

- ✕ Can't store in runtime containers

```
vector<???> shapes;
shapes.push_back(circle{});
shapes.push_back(box{});
```

Dynamic polymorphism - virtual

Dynamic polymorphism - virtual

Interface

Dynamic polymorphism - virtual

Interface

```
struct shape{
```

Dynamic polymorphism - virtual

Interface

```
struct shape{
```

```
  virtual void draw() const = 0;
```

Dynamic polymorphism - virtual

Interface

```
struct shape{

  virtual void draw() const = 0;

  virtual box get_bounding_box() const = 0;
```

Dynamic polymorphism - virtual

Interface

```
struct shape{

  virtual void draw() const = 0;

  virtual box get_bounding_box() const = 0;

  virtual void translate(double x, double y) = 0;
```

Dynamic polymorphism - virtual

Interface

```
struct shape{

  virtual void draw() const = 0;

  virtual box get_bounding_box() const = 0;

  virtual void translate(double x, double y) = 0;

  virtual void rotate(double degrees) = 0;
};
```

Don't forget the virtual destructor!

```
struct shape{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
  virtual ~shape(){}
};
```

Circle

```
class circle:public shape{
public:
  void draw() const override;
  box get_bounding_box() const override;
  void translate(double x, double y) override;
  void rotate(double degrees) override;
};
```

Smart Draw

```
void smart_draw(const shape& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Smart Draw

```
void smart_draw(const shape& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

☑ No longer a template

Smart Draw

```
void smart_draw(const shape& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Smart Draw

```
void smart_draw(const shape& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

❌ Increased coupling

Splitting Shape

```
struct shape{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
  virtual ~shape(){}
};
```

Splitting Shape

```
struct shape{
   virtual void draw() const = 0;
   virtual box get_bounding_box() const = 0;
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
  virtual ~shape(){}
};
```

Splitting Shape

```
struct drawing_interface{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
  virtual ~drawing_interface(){}
};
```

Splitting Shape

```
struct drawing_interface{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
  virtual ~drawing_interface(){}
};
```

```
struct transforming_interface{
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
  virtual ~transforming_interface(){}
};
```

Splitting Shape

```
struct drawing_interface{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
  virtual ~drawing_interface(){}
};
```

```
struct transforming_interface{
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
  virtual ~transforming_interface(){}
};
```

```
struct shape: drawing_interface, transforming_interface{};
```

Smart Draw

```
void smart_draw(const drawing_interface& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Spin

```
void spin(const transforming_interface& s){
  s.translate(4,2);
  s.rotate(45);
}
```

Use

```
std::vector<std::unique_ptr<shape>> shapes;

for(auto& s:shapes){
  spin(*s);
  smart_draw(*s,viewport);

}
```

Use

```
  std::vector<std::unique_ptr<shape>> shapes;

for(auto& s:shapes){
  spin(*s);
  smart_draw(*s,viewport);

}
```

Use

```
  std::vector<std::unique_ptr<shape>> shapes;

for(auto& s:shapes){
  spin(*s);
  smart_draw(*s,viewport);

}
```

- [x] Able to be stored in runtime containers

Did we do it?

Did we do it?

- [x] Able to be used in non-template functions.

Did we do it?

- [x] Able to be used in non-template functions.
- [x] Able to be stored in runtime containers


What did it cost?
Everything.

Costs

Costs

- ✕ Low boilerplate - Need inheritance hierarchy

Costs

- ✕ Low boilerplate - Need inheritance hierarchy
- ✕ Easy adaptation of existing class - Have to inherit from base

Costs

- ✕ Low boilerplate - Need inheritance hierarchy
- ✕ Easy adaptation of existing class - Have to inherit from base
- ✕ Value semantics - Pointer semantics

Costs

- ✕ Low boilerplate - Need inheritance hierarchy
- ✕ Easy adaptation of existing class - Have to inherit from base
- ✕ Value semantics - Pointer semantics
- ☑ ✕ Low coupling - Only if we are careful/lucky

Costs

- ✕ Low boilerplate - Need inheritance hierarchy
- ✕ Easy adaptation of existing class - Have to inherit from base
- ✕ Value semantics - Pointer semantics
- ☑ ✕ Low coupling - Only if we are careful/lucky
- ✕ PPP - Polymorphic types have to inherit from base

Costs

- ✕ Low boilerplate - Need inheritance hierarchy
- ✕ Easy adaptation of existing class - Have to inherit from base
- ✕ Value semantics - Pointer semantics
- ☑ ✕ Low coupling - Only if we are careful/lucky
- ✕ PPP - Polymorphic types have to inherit from base
- ☑ ✕ Performance - Inherent overhead in runtime dispatch

What would we like

What would we like

- ☑ Low boilerplate

What would we like

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class

What would we like

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics

What would we like

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling

What would we like

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP

What would we like

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch

What would we like

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch
- ☑ Able to be used in non-template functions.

What would we like

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch
- ☑ Able to be used in non-template functions.
- ☑ Able to be stored in runtime containers

Constraints

Constraints

- No MACROS

Constraints

- No MACROS
- Standard C++17

There is already a solution in the C++11 stdlib

History lesson

History lesson

1998

History lesson

1998

```
struct CommandInterface{virtual void execute() = 0;}
```

History lesson

1998

```
struct CommandInterface{virtual void execute() = 0;}
```

```
std::vector<CommandInterface*> commands;
```

History lesson

1998

```
struct CommandInterface{virtual void execute() = 0;}
```

```
std::vector<CommandInterface*> commands;
```

```
for(int i = 0; i < commands.size(); ++i)commands[i]->execute();
```

History lesson

1998

```
struct CommandInterface{virtual void execute() = 0;}
```

```
std::vector<CommandInterface*> commands;
```

```
for(int i = 0; i < commands.size(); ++i)commands[i]->execute();
```

2011

History lesson

1998

```
struct CommandInterface{virtual void execute() = 0;}
```

```
std::vector<CommandInterface*> commands;
```

```
for(int i = 0; i < commands.size(); ++i)commands[i]->execute();
```

2011

```
std::vector<std::function<void()>> commands;
```

History lesson

1998

```
struct CommandInterface{virtual void execute() = 0;}
```

```
std::vector<CommandInterface*> commands;
```

```
for(int i = 0; i < commands.size(); ++i)commands[i]->execute();
```

2011

```
std::vector<std::function<void()>> commands;
```

```
for(auto& command:commands)command();
```

std::function

std::function

- ☑ Low boilerplate

std::function

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class

std::function

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics

std::function

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling

std::function

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling
- ✅ PPP

std::function

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch

std::function

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch
- ☑ Able to be used in non-template functions.

std::function

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling
- ✅ PPP
- ✅ ❌ Performance - Inherent overhead in runtime dispatch
- ✅ Able to be used in non-template functions.
- ✅ Able to be stored in runtime containers

std::function

```
std::function<void()> f;
```

std::function

```
std::function<void()> f;
```

Function type

- Has the information on the parameters and return type of the function

std::function

```
std::function<void()> f;
```

Function type

- Has the information on the parameters and return type of the function
- We, can examine the parameters and return type, using partial template specialization.

function_ref

function_ref

- For passing to functions, sometimes you just need a reference type, instead of an owning type

function_ref

- For passing to functions, sometimes you just need a reference type, instead of an owning type
- Ongoing work on standardizing, but you can find open source implementations

Are we done?

Are we done?

- ✕ Only supports a single operation - function call operator

How to name operations

How to name operations

Function name

```
void rotate(double degrees);
```

How to name operations

Function name

```
void rotate(double degrees);
```

✕ C++17 cannot metaprogram names, except for macros

How to name operations

Function name

```
void rotate(double degrees);
```

✕ C++17 cannot metaprogram names, except for macros

Type

How to name operations

Function name

```
void rotate(double degrees);
```

✕ C++17 cannot metaprogram names, except for macros

Type

```
struct rotate{};
using rotate_signature = void(rotate,double);
```

How to name operations

Function name

```
void rotate(double degrees);
```

✕ C++17 cannot metaprogram names, except for macros

Type

```
struct rotate{};
using rotate_signature = void(rotate,double);
```

☑ C++17 metaprograms types like a boss

Polymorphic

Polymorphic

https://github.com/google/cpp-from-the-sky-down/blob/master/metaprogrammed_polymorphism/polymorphic.hpp

Three important things to say about polymorphic

Three important things to say about polymorphic

This is not an officially supported Google product.

Three important things to say about polymorphic

This is not an officially supported Google product.

This is not an officially supported Google product.

Three important things to say about polymorphic

This is not an officially supported Google product.

This is not an officially supported Google product.

This is not an officially supported Google product.

Polymorphic combines aspects of virtual and overloading

Polymorphic combines aspects of virtual and overloading

- Specify interface
  - Virtual

Polymorphic combines aspects of virtual and overloading

- Specify interface
  - Virtual
- Use overloading to connect a type to an operation
  - `poly_extend` is used as the overloaded extension point

Specify Interface

Specify Interface

Virtual

```
struct transforming_interface{
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
};

void spin(transforming_interface& s);
```

Specify Interface

Virtual

```
struct transforming_interface{
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
};

void spin(transforming_interface& s);
```

Polymorphic

Specify Interface

Virtual

```
struct transforming_interface{
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
};

void spin(transforming_interface& s);
```

Polymorphic

```
struct translate{};
struct rotate{};

void spin(polymorphic::ref<void(translate,double x,double y),
                           void(rotate,double degrees)> s);
```

Use overloading to connect an interface to an operation

Overloading

```
template<typename Shape>
void translate(Shape& s, double x, double y){
 s.translate(x,y);
}

void translate(box& b, double x, double y){
// Translate box.
}
```

Use overloading to connect an interface to an operation

Overloading

```
template<typename Shape>
void translate(Shape& s, double x, double y){
 s.translate(x,y);
}

void translate(box& b, double x, double y){
// Translate box.
}
```

Polymorphic

Use overloading to connect an interface to an operation

Overloading

```
template<typename Shape>
void translate(Shape& s, double x, double y){
 s.translate(x,y);
}

void translate(box& b, double x, double y){
// Translate box.
}
```

Polymorphic

```
template<typename Shape>
void poly_extend(translate,Shape& s, double x, double y){
 s.translate(x,y);
}

void poly_extend(translate,box& b, double x, double y){
// Translate box.
}
```

Spin implementation

Spin implementation

Virtual

Spin implementation

Virtual

```
void spin(transforming_interface& s){
  s.translate(4,2);
  s.rotate(45);
}
```

Spin implementation

Virtual

```
void spin(transforming_interface& s){
  s.translate(4,2);
  s.rotate(45);
}
```

Polymorphic

Spin implementation

Virtual

```
void spin(transforming_interface& s){
  s.translate(4,2);
  s.rotate(45);
}
```

Polymorphic

```
void spin(polymorphic::ref<void(translate,double x,double y),
                           void(rotate,double degrees)> s){
  s.call<translate>(4,2);
  s.call<rotate>(45);
}
```

Using spin

```
class box{
public:
  point upper_left()const;
  void set_upper_left(point);
  point lower_right()const;
  void set_lower_right(point);
  bool overlaps(const box&)const;
};

class circle{
public:
  void draw() const;
  box get_bounding_box() const;
  void translate(double x, double y);
  void rotate(double degrees);
};
```

Using spin

```
class box{
public:
  point upper_left()const;
  void set_upper_left(point);
  point lower_right()const;
  void set_lower_right(point);
  bool overlaps(const box&)const;
};

class circle{
public:
  void draw() const;
  box get_bounding_box() const;
  void translate(double x, double y);
  void rotate(double degrees);
};
```

Same definitions we used with overload example

Using spin

```
class box{
public:
  point upper_left()const;
  void set_upper_left(point);
  point lower_right()const;
  void set_lower_right(point);
  bool overlaps(const box&)const;
};

class circle{
public:
  void draw() const;
  box get_bounding_box() const;
  void translate(double x, double y);
  void rotate(double degrees);
};
```

Same definitions we used with overload example

```
circle c;
box b;
spin(c);
spin(b);
```

Using spin

```
class box{
public:
  point upper_left()const;
  void set_upper_left(point);
  point lower_right()const;
  void set_lower_right(point);
  bool overlaps(const box&)const;
};

class circle{
public:
  void draw() const;
  box get_bounding_box() const;
  void translate(double x, double y);
  void rotate(double degrees);
};
```

Same definitions we used with overload example

```
circle c;
box b;
spin(c);
spin(b);
```

✅ There are no polymorphic types, only a polymorphic use of similar types

What about const?

What about const?

```
template<typename Shape>
void smart_draw(const Shape& s, const box& viewport){
  auto bounding_box = get_bounding_box(s);
  if(viewport.overlaps(bounding_box)) draw(x);
}
```

Smart Draw

Smart Draw

Virtual

Smart Draw

Virtual

```
struct drawing_interface{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
};

void smart_draw(const drawing_interface& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Smart Draw

Virtual

```
struct drawing_interface{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
};

void smart_draw(const drawing_interface& s, const box& viewport){
  auto bounding_box = s.get_bounding_box();
  if(viewport.overlaps(bounding_box)) s.draw();
}
```

Polymorphic

```
struct draw{};
struct get_bounding_box{};

void smart_draw(polymorphic::ref<void(draw)const,
                                 box(get_bounding_box)const>, const box& viewport){
  auto bounding_box = s.call<get_bounding_box>();
  if(viewport.overlaps(bounding_box)) s.call<draw>();
}
```

Const polymorphic::ref

```
polymorphic::ref<void(draw)const,
                 box(get_bounding_box)const>
```

Const polymorphic::ref

```
polymorphic::ref<void(draw)const,
                 box(get_bounding_box)const>
```

As long as all the function types in a polymorphic::ref are `const` qualified, it will bind to a const object.

Using smart_draw

```
circle c;
smart_draw(std::as_const(c),viewport);
```

Polymorphic object

Polymorphic object

```
using shape = polymorphic::object<
                void(draw)const,
                box(get_bounding_box)const,
                void(translate,double x,double y),
                void(rotate,double degrees)>;
```

Value Semantics

Value Semantics

```
shape s1{circle{}};
shape s2 = s1;
spin(s1);
smart_draw(s1,viewport);
smart_draw(s2,viewport);
```

Store in collection

```
std::vector<shape> shapes;
shapes.push_back(circle{});
shapes.push_back(box{});

for(auto& s:shapes) smart_draw(s,viewport);
```

Low coupling

With virtual, we had to carefully group our member functions to avoid coupling

With virtual, we had to carefully group our member functions to avoid coupling

```
struct drawing_interface{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
  virtual ~drawing_interface(){}
};
struct transforming_interface{
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
  virtual ~transforming_interface(){}
};
struct shape: drawing_interface, transforming_interface{};
```

With virtual, we had to carefully group our member functions to avoid coupling

```
struct drawing_interface{
  virtual void draw() const = 0;
  virtual box get_bounding_box() const = 0;
  virtual ~drawing_interface(){}
};
struct transforming_interface{
  virtual void translate(double x, double y) = 0;
  virtual void rotate(double degrees) = 0;
  virtual ~transforming_interface(){}
};
struct shape: drawing_interface, transforming_interface{};
```

Polymorphic

- Does not matter how you group the function types.

Polymorphic conversions

Polymorphic conversions

```
void smart_draw(polymorphic::ref<void(draw)const,
                                 box(get_bounding_box)const> s, const box& viewport);
  auto bounding_box = s.call<get_bounding_box>();
  if(viewport.overlaps(bounding_box)) s.call<draw>();
}
```

Polymorphic conversions

```
void smart_draw(polymorphic::ref<void(draw)const,
                                 box(get_bounding_box)const> s, const box& viewport);
  auto bounding_box = s.call<get_bounding_box>();
  if(viewport.overlaps(bounding_box)) s.call<draw>();
}
```

```
using shape1 = polymorphic::object<
                 void(draw)const,
                 box(get_bounding_box)const,
                 void(translate,double x,double y),
                 void(rotate,double degrees)>;
shape1 s1{circle{}};
smart_draw(s1,viewport);
```

Polymorphic conversions

```
void smart_draw(polymorphic::ref<void(draw)const,
                                 box(get_bounding_box)const> s, const box& viewport);
  auto bounding_box = s.call<get_bounding_box>();
  if(viewport.overlaps(bounding_box)) s.call<draw>();
}
```

```
using shape1 = polymorphic::object<
                 void(draw)const,
                 box(get_bounding_box)const,
                 void(translate,double x,double y),
                 void(rotate,double degrees)>;
shape1 s1{circle{}};
smart_draw(s1,viewport);
```

```
using shape2 = polymorphic::object<
                 box(get_bounding_box)const,
                 void(translate,double x,double y),
                 void(draw)const,
                 void(rotate,double degrees)>;
shape2 s2{circle{}};
smart_draw(s2,viewport);
```

Benchmarks

- Windows Laptop - i7-8550U
- Fill a vector with different types of polymorphic objects, and then time how long it takes to iterate and call a method on each item.

Type	Median Time (ns)
Non-Virtual	121
Virtual	532
std::function	539
Polymorphic Ref	515
Polymorphic Object	483

Size

- `sizeof(ref)` typically `3*sizeof(void*)`
- `sizeof(object)` typically `4*sizeof(void*)`

Final Assessment

Final Assessment

- ☑ Low boilerplate

Final Assessment

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class

Final Assessment

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics

Final Assessment

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling

Final Assessment

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP

Final Assessment

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch

Final Assessment

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch
- ☑ Able to be used in non-template functions.

Final Assessment

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch
- ☑ Able to be used in non-template functions.
- ☑ Able to be stored in runtime containers

Show me the code?

Show me the code?

https://github.com/google/cpp-from-the-sky-down/blob/master/metaprogrammed_polymorphism/polymorphic.hpp

Show me the code?

https://github.com/google/cpp-from-the-sky-down/blob/master/metaprogrammed_polymorphism/polymorphic.hpp

247 lines, including comments and whitespace

VTable

- Use an array of function pointer
- Each position of the array, corresponds to a function type that is passed in
- There is a unique array for each type, and each sequence of polymorphic operations
- Each entry in the vtable, contains a trampoline function to jump to the appropriate overload

Trampoline Function

Trampoline Function

```
template <typename T, typename Signature> struct trampoline;
```

Trampoline Function

```
    template <typename T, typename Signature> struct trampoline;
```

```
template <typename T, typename Return, typename Method, typename... Parameters>
struct trampoline<T, Return(Method, Parameters...)> {
    static auto jump(void* t, Parameters... parameters) -> Return {
        return poly_extend(Method{}, *static_cast<T*>(t), parameters...);
    }
};
```

Trampoline Function

```
    template <typename T, typename Signature> struct trampoline;

template <typename T, typename Return, typename Method, typename... Parameters>
struct trampoline<T, Return(Method, Parameters...)> {
    static auto jump(void* t, Parameters... parameters) -> Return {
        return poly_extend(Method{}, *static_cast<T*>(t), parameters...);
    }
};
```

Trampoline Function

```
    template <typename T, typename Signature> struct trampoline;

template <typename T, typename Return, typename Method, typename... Parameters>
struct trampoline<T, Return(Method, Parameters...)> {
     static auto jump(void* t, Parameters... parameters) -> Return {
        return poly_extend(Method{}, *static_cast<T*>(t), parameters...);
    }
};
```

Trampoline Function

```
    template <typename T, typename Signature> struct trampoline;

template <typename T, typename Return, typename Method, typename... Parameters>
struct trampoline<T, Return(Method, Parameters...)> {
    static auto jump(void* t, Parameters... parameters) -> Return {
        return poly_extend(Method{}, *static_cast<T*>(t), parameters...);
    }
};
```

Trampoline Function

```
    template <typename T, typename Signature> struct trampoline;

template <typename T, typename Return, typename Method, typename... Parameters>
struct trampoline<T, Return(Method, Parameters...)> {
    static auto jump(void* t, Parameters... parameters) -> Return {
        return poly_extend(Method{}, *static_cast<T*>(t), parameters...);
    }
};
```

Build vtable from trampoline functions

Build vtable from trampoline functions

```
template <typename T, typename... Signatures>
inline const vtable_fun vtable[] = {    reinterpret_cast<vtable_fun>(trampoline<T, Signatures>::jump)... };
```

Build vtable from trampoline functions

```
template <typename T, typename... Signatures>
inline const vtable_fun vtable[] = {    reinterpret_cast<vtable_fun>(trampoline<T, Signatures>::jump)... };
```

Build vtable from trampoline functions

```
template <typename T, typename... Signatures>
inline const vtable_fun vtable[] = {    reinterpret_cast<vtable_fun>(trampoline<T, Signatures>::jump)... };
```

Calling the vtable

Calling the vtable

```
template <size_t I, typename Signature> struct vtable_caller;

template <size_t I, typename Method, typename Return, typename... Parameters>
struct vtable_caller<I, Return(Method, Parameters...)> {
    decltype(auto) operator()(const vtable_fun* vt, const std::uint8_t* permutation, Method, void* t,
        Parameters... parameters) const {
        return reinterpret_cast<ptr<Return(void*, Parameters...)>>(vt[permutation[I]])(
            t, parameters...);
    }
};
```

Calling the vtable

```
template <size_t I, typename Signature> struct vtable_caller;

template <size_t I, typename Method, typename Return, typename... Parameters>
struct vtable_caller<I, Return(Method, Parameters...)> {
    decltype(auto) operator()(const vtable_fun* vt, const std::uint8_t* permutation, Method, void* t,
        Parameters... parameters) const {
        return reinterpret_cast<ptr<Return(void*, Parameters...)>>(vt[permutation[I]])(
            t, parameters...);
    }
};
```

Calling the vtable

```
template <size_t I, typename Signature> struct vtable_caller;

template <size_t I, typename Method, typename Return, typename... Parameters>
struct vtable_caller<I, Return(Method, Parameters...)> {
    decltype(auto) operator()(const vtable_fun* vt, const std::uint8_t* permutation, Method, void* t,
        Parameters... parameters) const {
        return reinterpret_cast<ptr<Return(void*, Parameters...)>>(vt[permutation[I]])(
            t, parameters...);
    }
};
```

Calling the vtable

```
template <size_t I, typename Signature> struct vtable_caller;

template <size_t I, typename Method, typename Return, typename... Parameters>
struct vtable_caller<I, Return(Method, Parameters...)> {
    decltype(auto) operator()(const vtable_fun* vt, const std::uint8_t* permutation, Method, void* t,
        Parameters... parameters) const {
        return reinterpret_cast<ptr<Return(void*, Parameters...)>>(vt[permutation[I]])(
            t, parameters...);
    }
};
```

Calling the vtable

```
template <size_t I, typename Signature> struct vtable_caller;

template <size_t I, typename Method, typename Return, typename... Parameters>
struct vtable_caller<I, Return(Method, Parameters...)> {
    decltype(auto) operator()(const vtable_fun* vt, const std::uint8_t* permutation, Method, void* t,
        Parameters... parameters) const {
        return reinterpret_cast<ptr<Return(void*, Parameters...)>>(vt[permutation[I]])(
            t, parameters...);
    }
};
```

Permutations

```
void smart_draw(polymorphic::ref<void(draw)const,
                                 box(get_bounding_box)const> s, const box& viewport);
  auto bounding_box = s.call<get_bounding_box>();
  if(viewport.overlaps(bounding_box)) s.call<draw>();
}
```

Permutations

```
void smart_draw(polymorphic::ref<void(draw)const,
                                 box(get_bounding_box)const> s, const box& viewport);
  auto bounding_box = s.call<get_bounding_box>();
  if(viewport.overlaps(bounding_box)) s.call<draw>();
}
```

```
using shape2 = polymorphic::object<
                 box(get_bounding_box)const,
                 void(translate,double x,double y),
                 void(draw)const,
                 void(rotate,double degrees)>;
```

Permutations

```
void smart_draw(polymorphic::ref<void(draw)const,
                                 box(get_bounding_box)const> s, const box& viewport);
  auto bounding_box = s.call<get_bounding_box>();
  if(viewport.overlaps(bounding_box)) s.call<draw>();
}
```

```
using shape2 = polymorphic::object<
                 box(get_bounding_box)const,
                 void(translate,double x,double y),
                 void(draw)const,
                 void(rotate,double degrees)>;
```

```
s.permutation_ = {2,0}
```

Ref Impl

```
template <typename Holder, typename Sequence, typename... Signatures>
    class ref_impl;

template <typename Holder, size_t... I, typename... Signatures>
    class ref_impl<Holder, std::index_sequence<I...>, Signatures...> {
```

Ref Impl

```
template <typename Holder, typename Sequence, typename... Signatures>
    class ref_impl;

template <typename Holder, size_t... I, typename... Signatures>
    class ref_impl<Holder, std::index_sequence<I...>, Signatures...> {
            const detail::vtable_fun* vptr_;
```

Ref Impl

```
template <typename Holder, typename Sequence, typename... Signatures>
    class ref_impl;

template <typename Holder, size_t... I, typename... Signatures>
    class ref_impl<Holder, std::index_sequence<I...>, Signatures...> {
            const detail::vtable_fun* vptr_;
             std::array<std::uint8_t, sizeof...(Signatures)> permutation_;
            Holder t_;
```

Ref Impl

```
template <typename Holder, typename Sequence, typename... Signatures>
    class ref_impl;

template <typename Holder, size_t... I, typename... Signatures>
    class ref_impl<Holder, std::index_sequence<I...>, Signatures...> {
            const detail::vtable_fun* vptr_;
            std::array<std::uint8_t, sizeof...(Signatures)> permutation_;
             Holder t_;
```

Ref Impl

```
template <typename Holder, typename Sequence, typename... Signatures>
    class ref_impl;

template <typename Holder, size_t... I, typename... Signatures>
    class ref_impl<Holder, std::index_sequence<I...>, Signatures...> {
            const detail::vtable_fun* vptr_;
            std::array<std::uint8_t, sizeof...(Signatures)> permutation_;
            Holder t_;
             static constexpr overload<vtable_caller<I, Signatures>...> call_vtable{};
```

Ref Impl

```
template <typename Holder, typename Sequence, typename... Signatures>
    class ref_impl;

template <typename Holder, size_t... I, typename... Signatures>
    class ref_impl<Holder, std::index_sequence<I...>, Signatures...> {
            const detail::vtable_fun* vptr_;
            std::array<std::uint8_t, sizeof...(Signatures)> permutation_;
            Holder t_;
            static constexpr overload<vtable_caller<I, Signatures>...> call_vtable{};

            template <typename Method, typename... Parameters>
             decltype(auto) call(Parameters&&... parameters) const {
                return call_vtable(vptr_, permutation_.data(), Method{}, t_.get_ptr(),
                    std::forward<Parameters>(parameters)...);
            }
```

Ref Impl

```
template <typename Holder, typename Sequence, typename... Signatures>
    class ref_impl;

template <typename Holder, size_t... I, typename... Signatures>
    class ref_impl<Holder, std::index_sequence<I...>, Signatures...> {
            const detail::vtable_fun* vptr_;
            std::array<std::uint8_t, sizeof...(Signatures)> permutation_;
            Holder t_;
            static constexpr overload<vtable_caller<I, Signatures>...> call_vtable{};

            template <typename Method, typename... Parameters>
            decltype(auto) call(Parameters&&... parameters) const {
                return call_vtable(vptr_, permutation_.data(), Method{}, t_.get_ptr(),
                    std::forward<Parameters>(parameters)...);
            }
```

Ref alias

```
template <typename... Signatures>
using ref = detail::ref_impl<
    detail::ptr_holder<std::conditional_t<
    std::conjunction_v<detail::is_const_signature<Signatures>...>,
    const void, void>>,
    std::make_index_sequence<sizeof...(Signatures)>, Signatures...>;
```

Ref alias

```
template <typename... Signatures>
using ref = detail::ref_impl<
    detail::ptr_holder<std::conditional_t<
    std::conjunction_v<detail::is_const_signature<Signatures>...>,
    const void, void>>,
    std::make_index_sequence<sizeof...(Signatures)>, Signatures...>;
```

Object alias

```
template <typename... Signatures>
using object = detail::ref_impl<
    std::conditional_t<
    std::conjunction_v<detail::is_const_signature<Signatures>...>,
    detail::shared_ptr_holder, detail::value_holder >,
    std::make_index_sequence<sizeof...(Signatures)>, Signatures...>;
```

Object alias

```
template <typename... Signatures>
using object = detail::ref_impl<
    std::conditional_t<
    std::conjunction_v<detail::is_const_signature<Signatures>...>,
    detail::shared_ptr_holder, detail::value_holder >,
    std::make_index_sequence<sizeof...(Signatures)>, Signatures...>;
```

Polymorphic

Polymorphic

- ☑ Low boilerplate

Polymorphic

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class

Polymorphic

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics

Polymorphic

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling

Polymorphic

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling
- ✅ PPP

Polymorphic

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch

Polymorphic

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch
- ☑ Able to be used in non-template functions.

Polymorphic

- ☑ Low boilerplate
- ☑ Easy adaptation of existing class
- ☑ Value semantics
- ☑ Low coupling
- ☑ PPP
- ☑ ✕ Performance - Inherent overhead in runtime dispatch
- ☑ Able to be used in non-template functions.
- ☑ Able to be stored in runtime containers

Polymorphic

- ✅ Low boilerplate
- ✅ Easy adaptation of existing class
- ✅ Value semantics
- ✅ Low coupling
- ✅ PPP
- ✅ ✕ Performance - Inherent overhead in runtime dispatch
- ✅ Able to be used in non-template functions.
- ✅ Able to be stored in runtime containers

References

- Sean Parent - Inheritance is the base class of evil - https://www.youtube.com/watch?v=bIhUE5uUFOA
- Louis Dionne - CppCon 2017 - Runtime polymorphism: Back to basics - https://www.youtube.com/watch?v=gVGtNFg4ay0

https://github.com/google/cpp-from-the-sky-down/blob/master/metaprogrammed_polymorphism/polymorphic.hpp

P1521

Types as function names

- A proposal I am working on to allow types as function names

P1521

Types as function names

- A proposal I am working on to allow types as function names
- Universal function call syntax
- Extension methods
- Extension points
- Overload sets
- Forwarding proxies proxies
- Adding monadic bind to pre-existing monad types
- Making a monadic type automatically map/bind

P1521

```
template<typename Shape>
void poly_extend(translate,Shape& s, double x, double y){
 s.translate(x,y);
}
```

P1521

```
template<typename Shape>
void poly_extend(translate,Shape& s, double x, double y){
 s.translate(x,y);
}
```

```
template<typename Shape>
void translate.(Shape& s, double x, double y){
 s.translate(x,y);
}
```

P1521

```
void spin(polymorphic::ref<void(translate,double x,double y),
                           void(rotate,double degrees)> s){
  s.call<translate>(4,2);
  s.call<rotate>(45);
}
```

P1521

```
void spin(polymorphic::ref<void(translate,double x,double y),
                           void(rotate,double degrees)> s){
  s.call<translate>(4,2);
  s.call<rotate>(45);
}
```

```
void spin(polymorphic::ref<void(translate,double x,double y),
                           void(rotate,double degrees)> s){
  s.translate.(4,2);
  s.rotate.(45);
}
```