META POLYMORPHISM

Jonathan Boccara

@JoBoccara

Fluent {C++}

MUREX™ FINANCIAL SOFTWARE

META

POLYMORPHISM

?

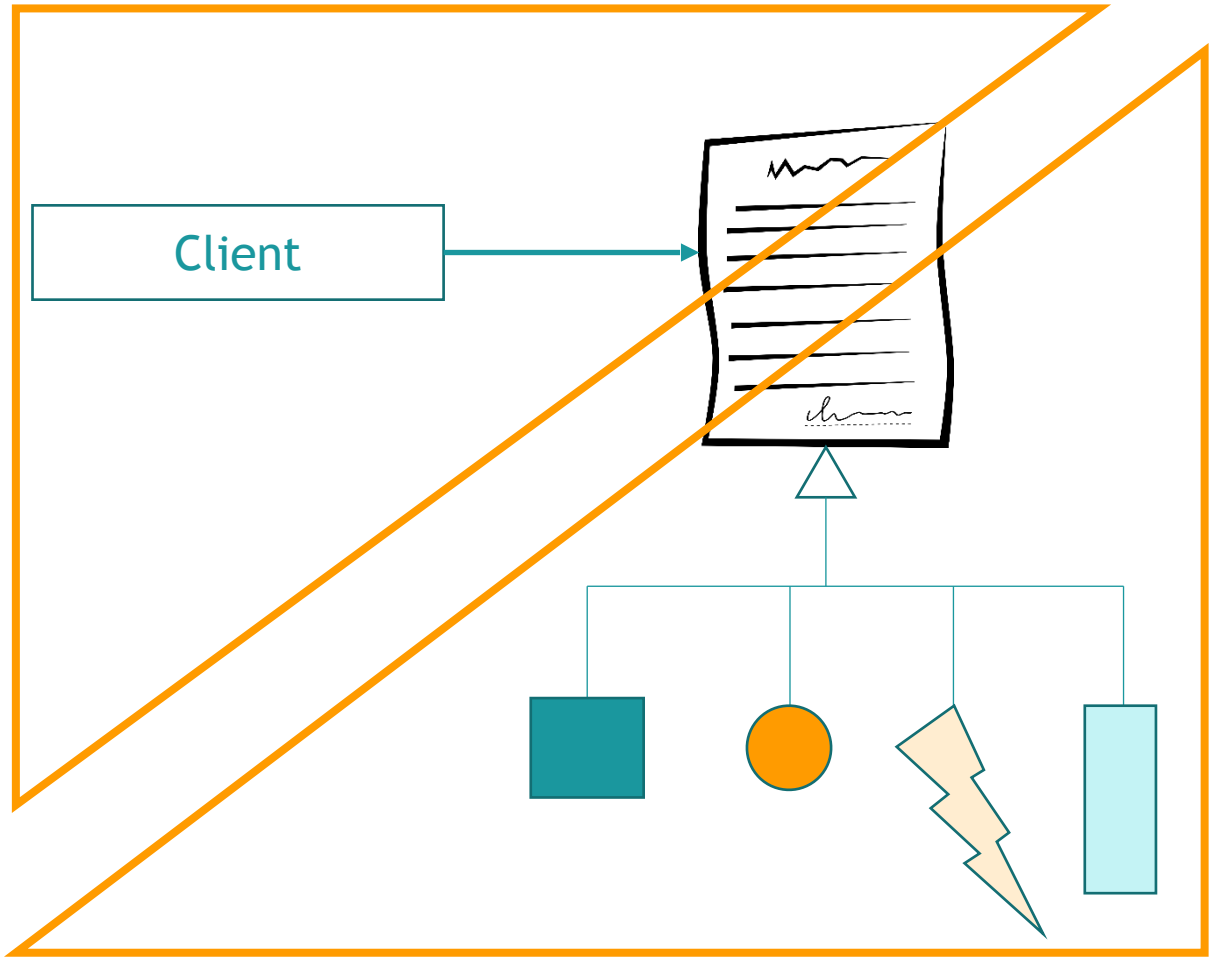
Polymorphism

An interface implementable by a

variety of components

poly morphism

Polymorphism is not going away

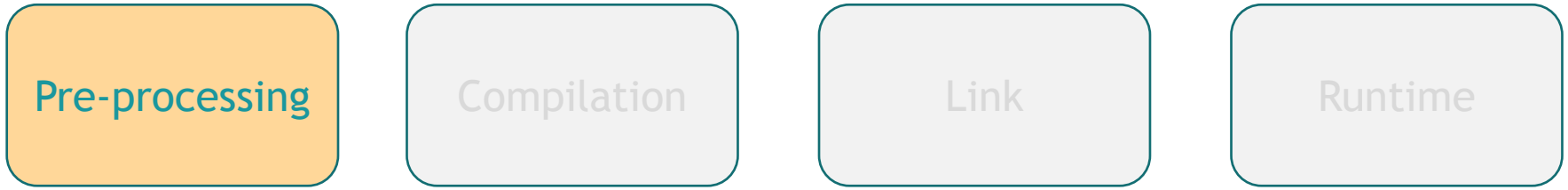


Modularity

Moments of resolution in C++

Pre-processing

Moments of resolution in C++



Text operations on the source code

```
#define A B
```

DISCLAIMER

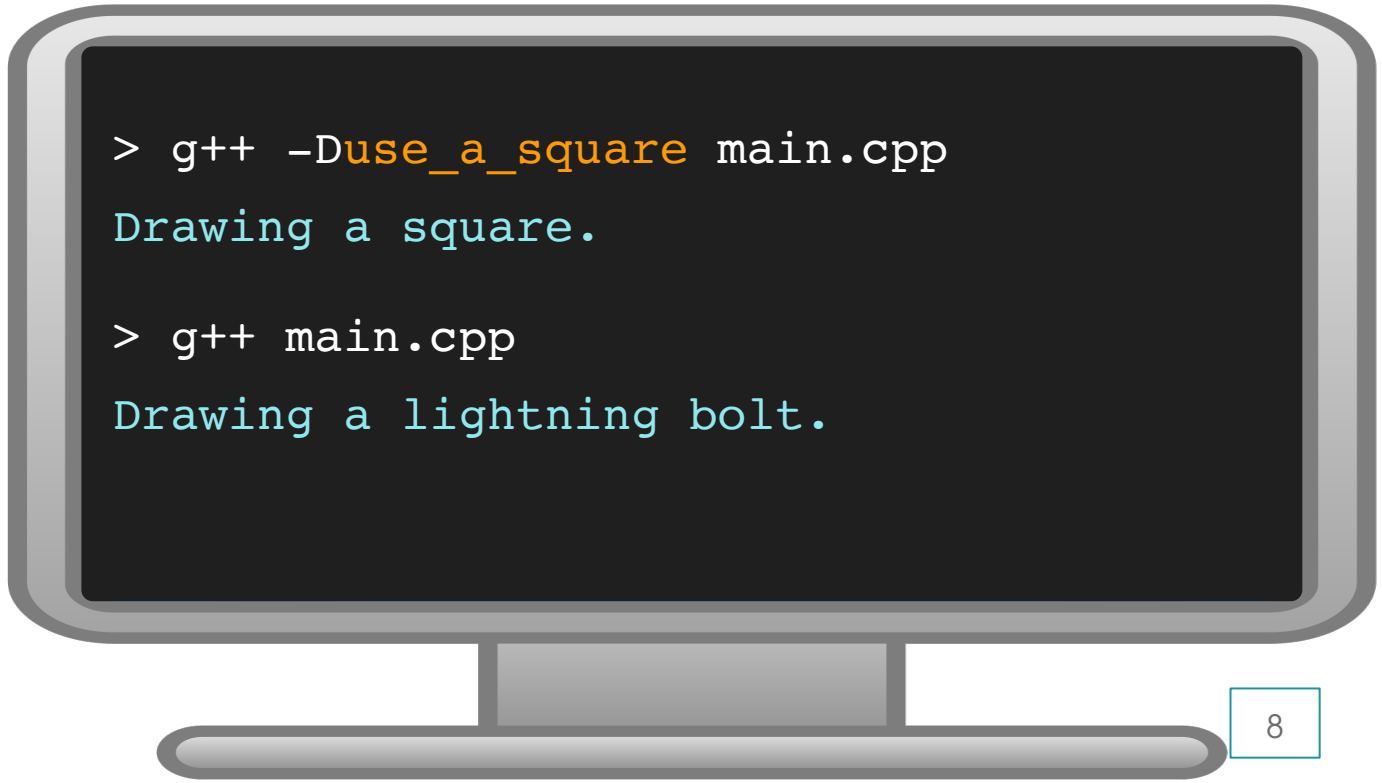
Macros are Bad. Except when they aren't.

The views and opinions exposed in this talk do not encourage you to use macros or polymorphism with macros, unless you are willing to try at your own risks. This talk presents polymorphism in the broader sense and includes macros for the sole purpose of illustrating polymorphism in a broader context, and asserts in no way that macros are always good, always bad, or whether it always depends. In fact we are willing to assert that macros are not always good, but still, that's not the point here.

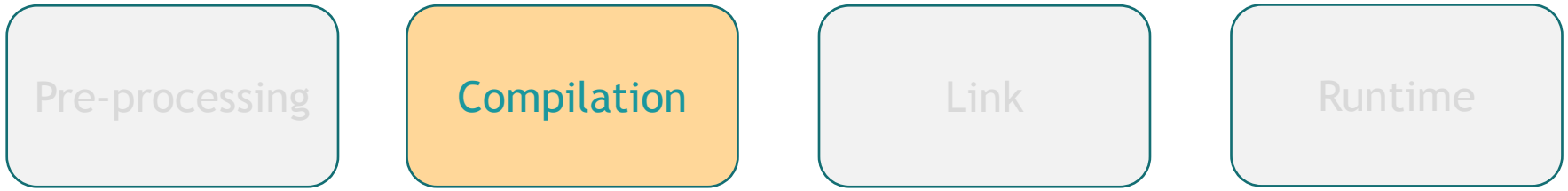
```
struct Square
{
    void draw() { std::cout << "Drawing a square."; }
};
struct LightningBolt
{
    void draw() { std::cout << "Drawing a lightning bolt."; }
};
```

```
#ifdef use_a_square
    #define Shape Square
#else
    #define Shape LightningBolt
#endif
```

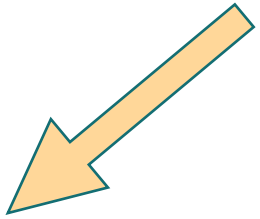
```
int main()
{
    Shape shape;
    shape.draw();
}
```

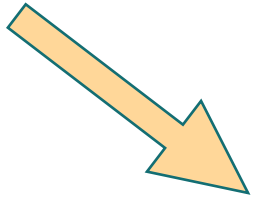


Moments of resolution in C++



```
template <typename Shape>
void f(Shape shape)
{
    shape.draw();
}
```





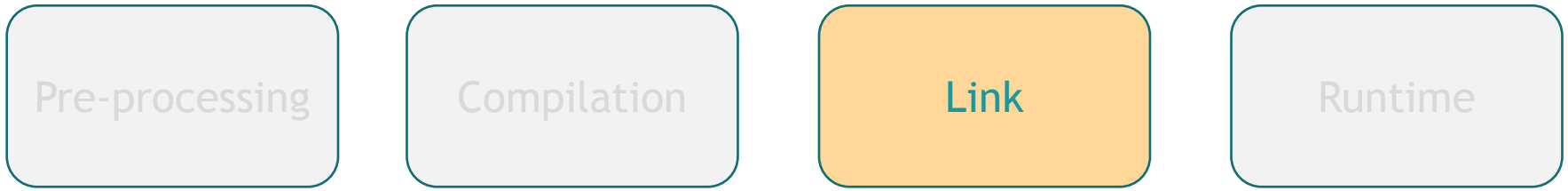
```
void f(Square shape)
{
    shape.draw();
}
```

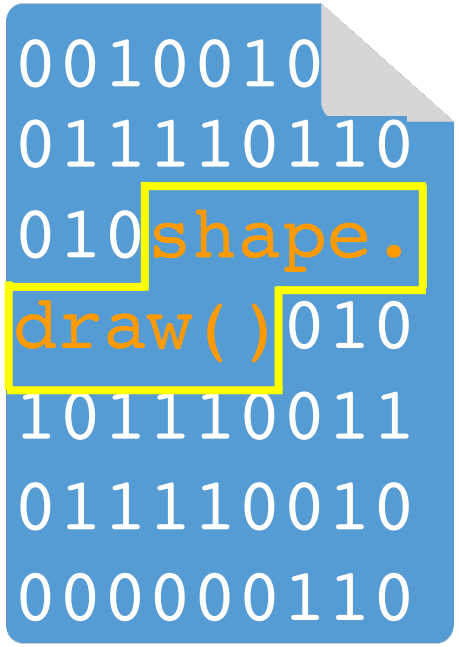
```
Square square;

f(square);
```

```
void f(LightningBolt shape)
{
    shape.draw();
}
```

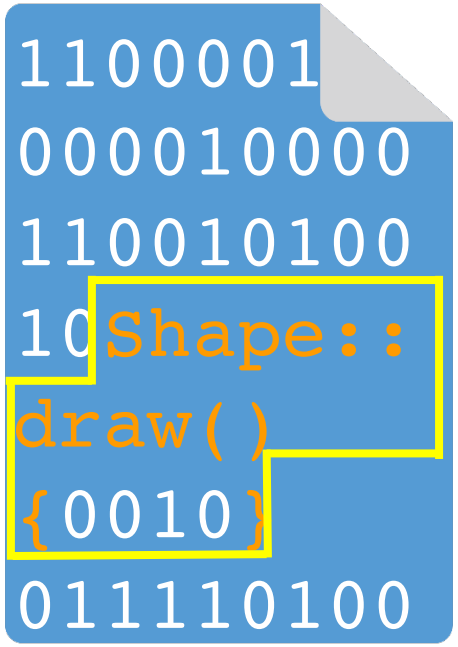
```
LightningBolt lightningBolt;

f(lightningBolt);
```

Moments of resolution in C++

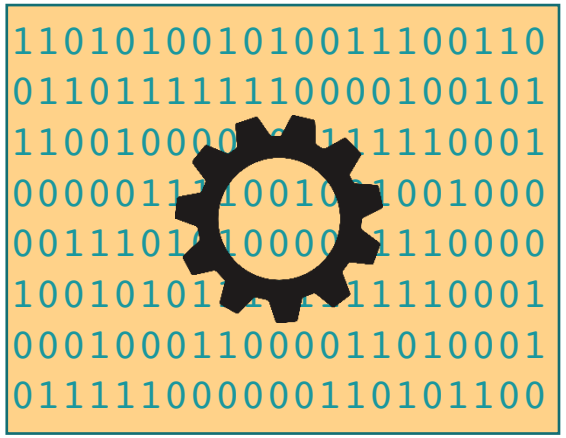




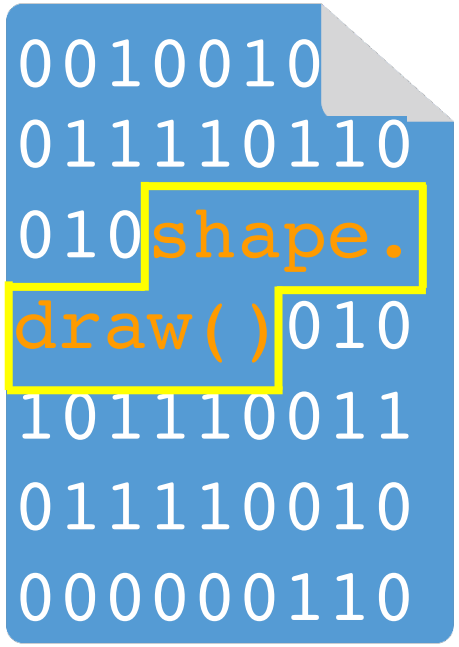
caller.o



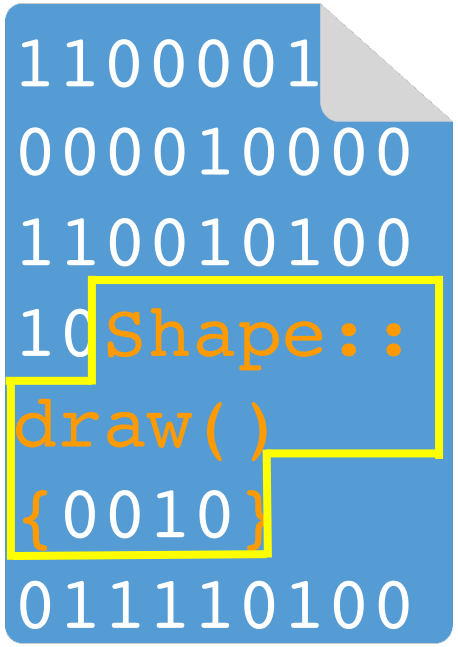
square.o



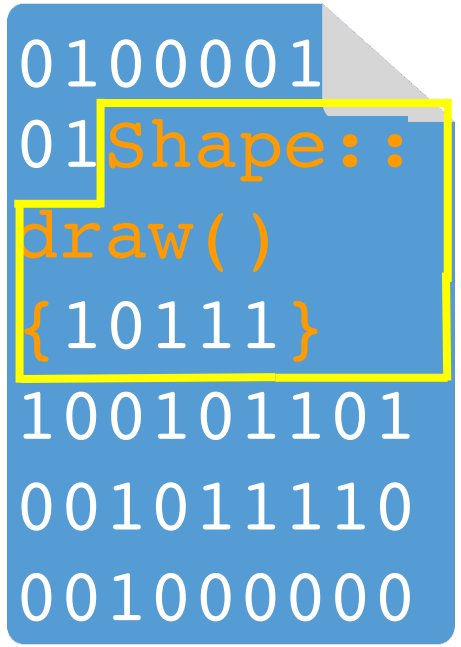
executable1



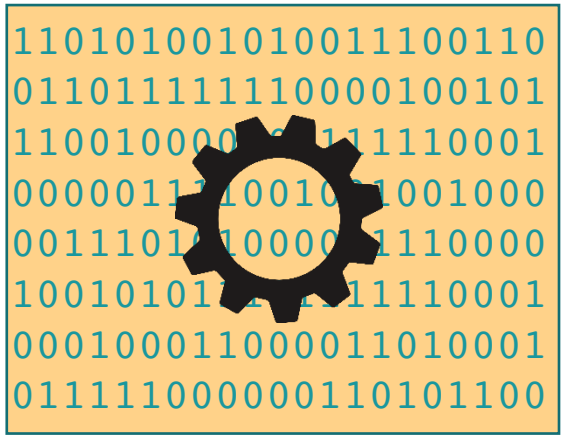
caller.o



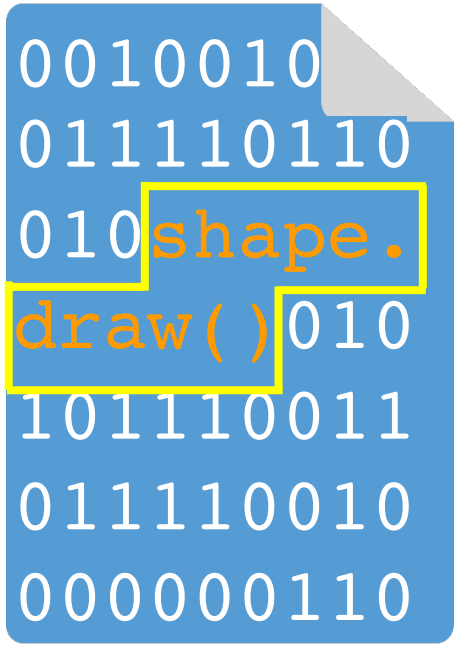
square.o



lightningbolt.o

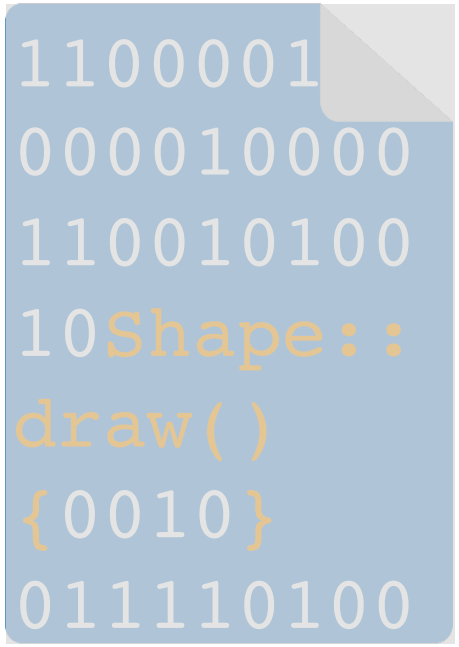


executable1



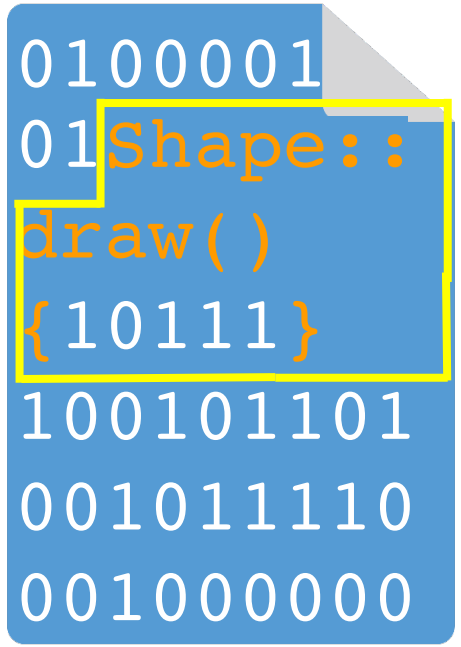
caller.o

+

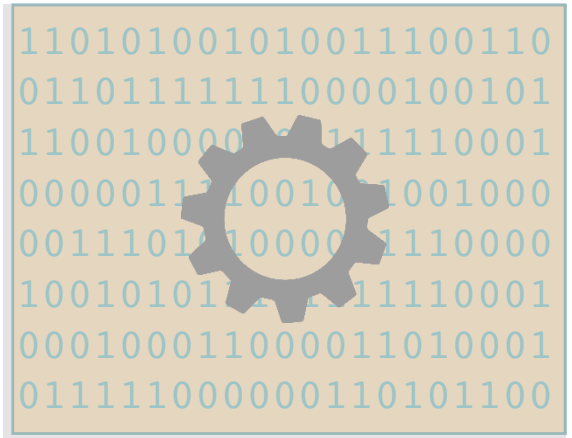


square.o

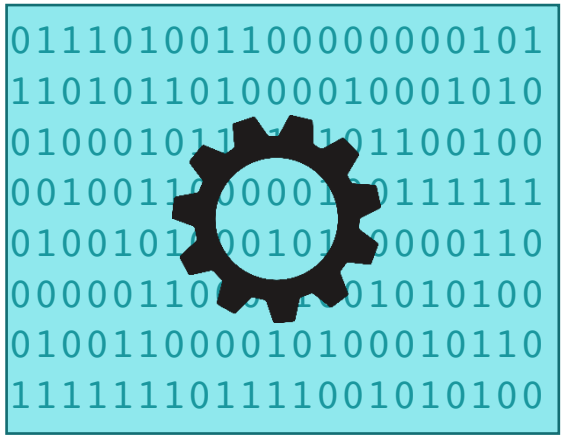
+



lightningbolt.o



executable1

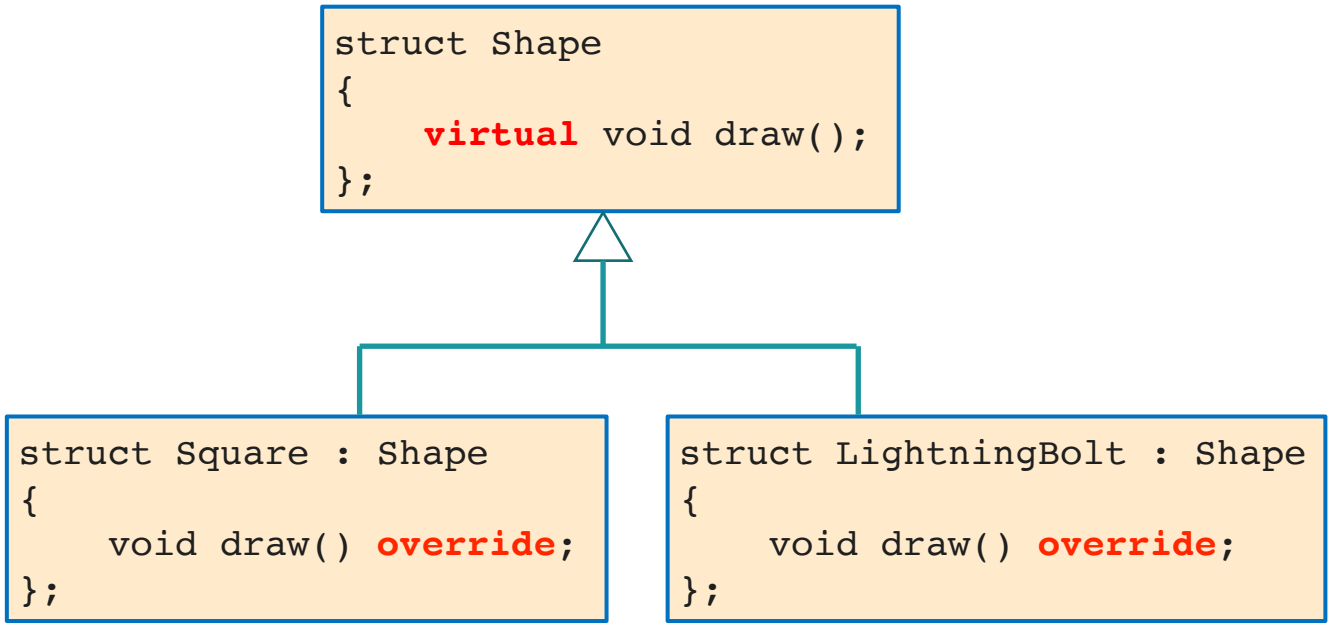


executable2

Moments of resolution in C++

Pre-processing | Compilation | Link | Runtime

Inheritance
Resolved at every call



Application: mocking

```
struct TestedClass
{
    void f(ProductionArgument const& arg);
};
```

```
struct ProductionArgument
{
    void doSomething();
};
```

Extract Interface

```
struct TestedClass
{
    void f(IArgument const& arg);
};
```

```
struct IArgument
{
    virtual void doSomething();
};
```

```
struct ProductionArgument : IArgument
{
    void doSomething() override;
};
```

```
struct MockArgument : IArgument
{
    void doSomething() override;
};
```

Runtime polymorphism

Application: mocking

```
struct TestedClass
{
    void f(ProductionArgument const& arg);
};
```

```
struct ProductionArgument
{
    void doSomething();
};
```

Extract Interface

at compile-time

```
struct TestedClass
{
    template<typename TArgument>
    void f(TArgument const& arg);
};
```

```
(IMPLICIT INTERFACE)

    doSomething()
```

Compile-time polymorphism

```
class ProductionArgument
{
    void doSomething();
};
```

```
class MockArgument
{
    void doSomething();
};
```

Application: mocking

```
struct TestedClass
{
    void f(ProductionArgument const& arg);
};
```

```
struct ProductionArgument
{
    void doSomething();
};
```

Extract Interface at compile-time

```
struct TestedClass
{
    template<typename TArgument>
    void f(TArgument const& arg)
    {
        arg.doSomething()
    }
};
```

```
(IMPLICIT CONTRACT)

    doSomething()
```

Compile-time polymorphism

```
class ProductionArgument
{
    void doSomething();
};
```

```
class MockArgument
{
    void doSomething();
};
```

Explicit instantiation

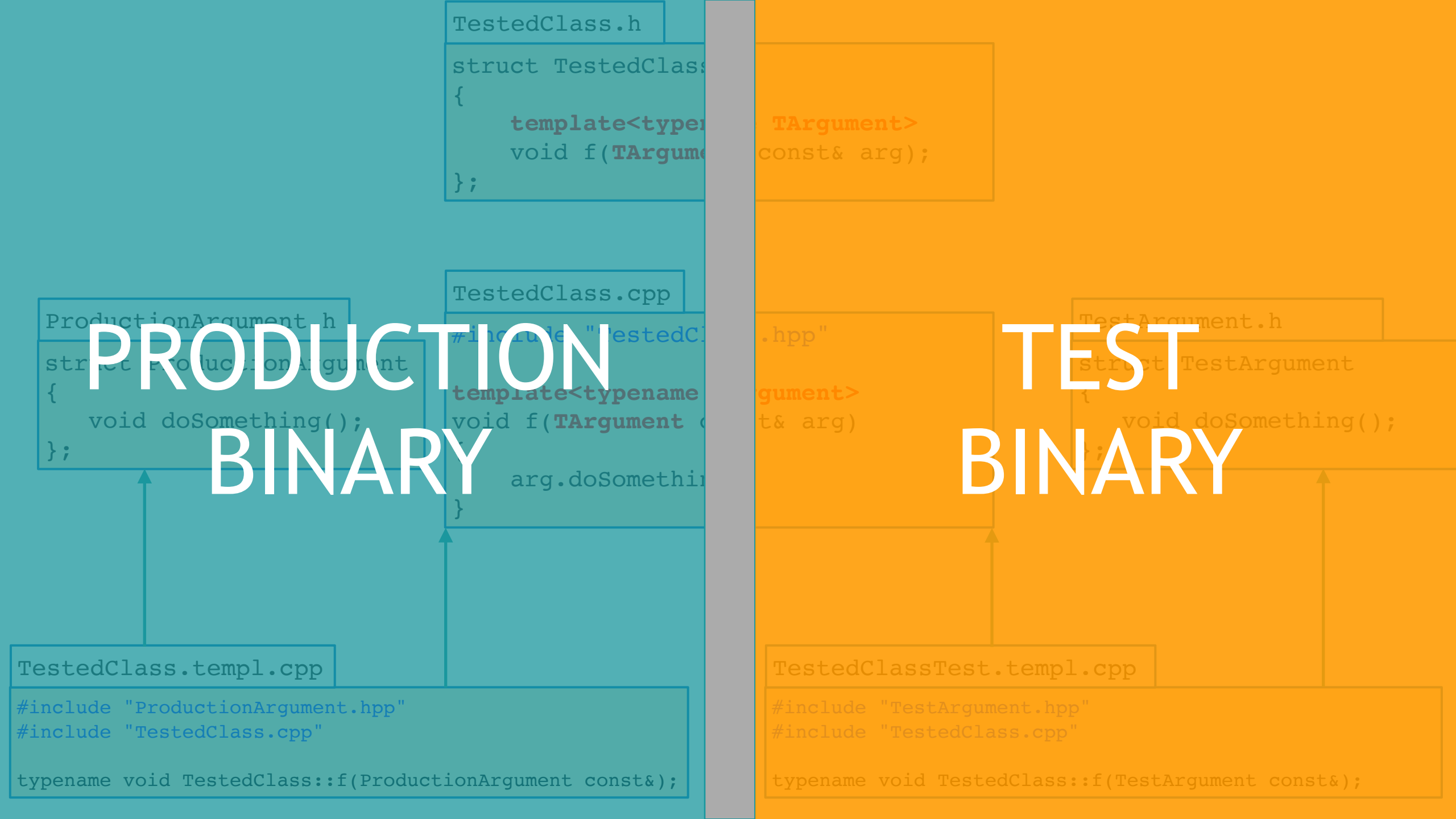
```
struct TestedClass
{
public:
    template<typename TArgument>
    void f(TArgument const& arg);
};
```

TestedClass.h


```
#include "TestedClass.h"
#include "Argument.h"
#include "TestArgument.h"

template<typename TArgument>
void TestedClass::f(TArgument const& arg)
{
    arg.doSomething();
}

template void TestedClass::f(ProductionArgument);
template void TestedClass::f(TestArgument);
```

TestedClass.cpp


THE CHINESE
WALL


TestedClass.h
struct TestedClass
{
template<typen
void f(TArgume
};
TArgument>
const& arg);
TestedClass.cpp
#include "TestedCl
.hpp"
template<typename
gument>
void f(TArgument
t& arg)
arg.doSomethi
}
ProductionArgument.h
struct ProductionArgument
{
void doSomething();
};
PRODUCTION BINARY
TestArgument.h
struct TestArgument
{
void doSomething();
};
TEST BINARY
TestedClass.templ.cpp
#include "ProductionArgument.hpp"
#include "TestedClass.cpp"
typename void TestedClass::f(ProductionArgument const&);
TestedClassTest.templ.cpp
#include "TestArgument.hpp"
#include "TestedClass.cpp"
typename void TestedClass::f(TestArgument const&);

TestedClass.h

```
struct TestedClass
{
    template<typename TArgument>
    void f(TArgument const& arg);
};
```

bit.do/chinesewall

TestedClass.cpp

```
#include "TestedClass.hpp"

template<typename TArgument>
void f(TArgument const& arg)
{
    arg.doSomething();
}
```

ProductionArgument.h

```
struct ProductionArgument
{
   void doSomething();
};
```

TestArgument.h

```
struct TestArgument
{
   void doSomething();
};
```

TestedClass.templ.cpp

```
#include "ProductionArgument.hpp"
#include “TestedClass.cpp"

typename void TestedClass::f(ProductionArgument const&);
```

TestedClassTest.templ.cpp

```
#include "TestArgument.hpp"
#include “TestedClass.cpp"

typename void TestedClass::f(ProductionArgument const&);
```

Moments of resolution in C++ FOR MOCKING

Pre-processing

Mockaron

In the code

THE CHINESE WALL

In the code

NOT in the code

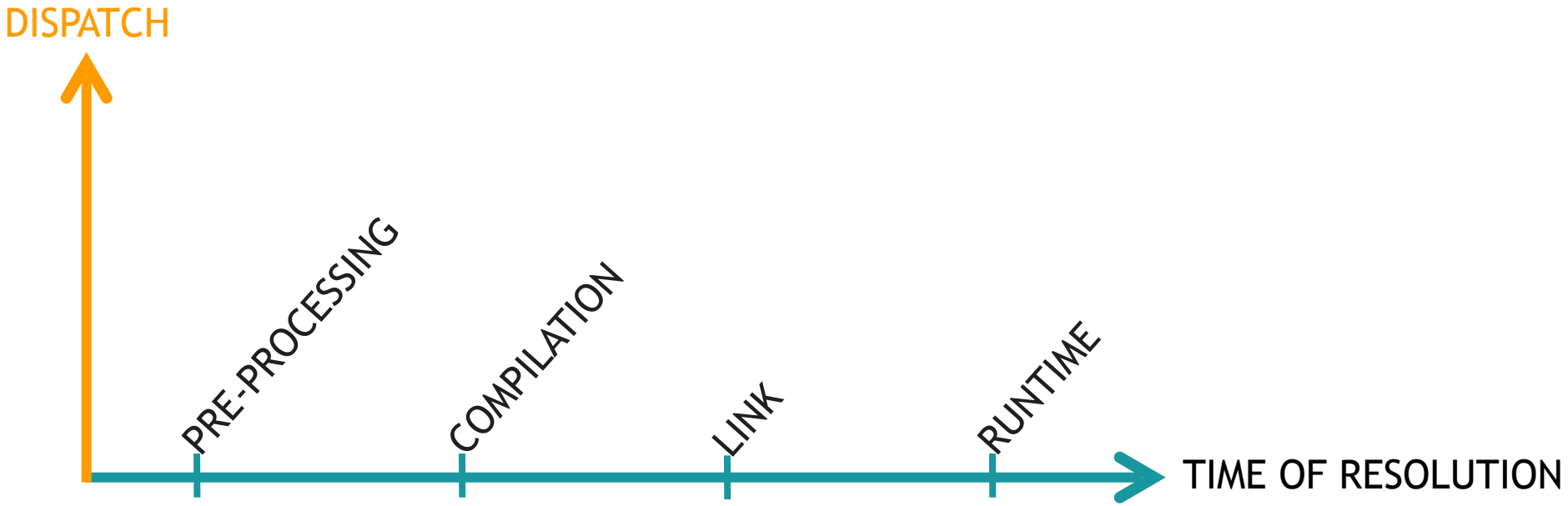
Virtual functions

In the code

Moments of resolution in C++

Pre-processing	Compilation	Link	Runtime
In the code	In the code	NOT in the code	In the code

The bigger picture of polymorphism



Dispatching

Dispatching
= Resolution of polymorphism
= Selection of one implementation

```
Shape const& shape = …

shape.draw();
```

dynamic type?
- Square
- LightningBolt

```
struct Shape
{
    virtual void draw() const = 0;
    virtual ~Shape();
};

struct Square : Shape
{
    void draw() const override;
};

struct LightningBolt : Shape
{
    void draw() const override;
};
```

Single dispatch = dispatch based on one criterion
Double dispatch = dispatch based on two criteria
and more generally, Multiple dispatch = dispatch based on several criteria

“C++ doesn’t have multiple dispatch”

Really?

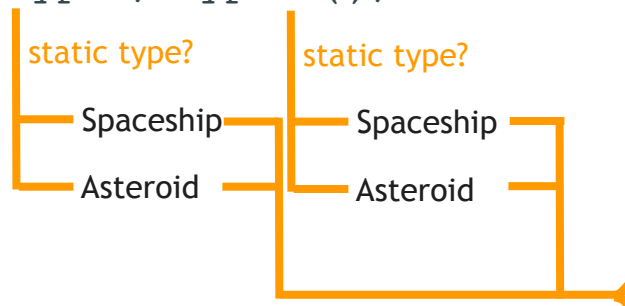
Moments of resolution in C++

Pre-processing

"C++ doesn't have multiple dispatch"

```
struct Asteroid{};
struct Spaceship{};
```

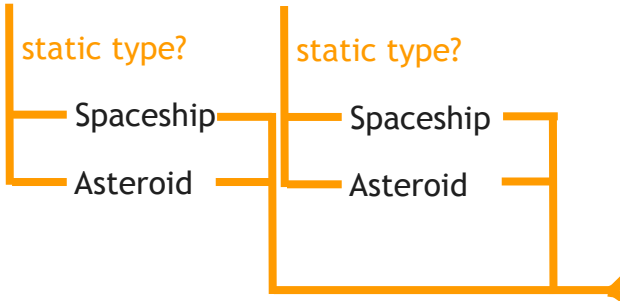
```
collide<Type1, Type2>();
```




```
template<typename Type1, typename Type2>
void collide();
{
    return collide<Type2, Type1>();
}

template<>
void collide<Asteroid, Asteroid>()
{
    std::cout << "The asteroids rebounded.\n";
}

template<>
void collide<Spaceship, Spaceship>()
{
    std::cout << "The two spaceships blew up.\n";
}

template<>
void collide<Spaceship, Asteroid>()
{
    std::cout << "The spaceship was crushed.\n";
}
```

```
struct Asteroid{};
struct Spaceship{};
```

```
collide(spaceObject1, spaceObject2);
```




```
template<typename Type1, typename Type2>
void collide(Type1 const& object1, Type2 const& object2)
{
    return collide(object2, object1);
}

void collide(Spaceship const& , Asteroid const& )
{
    std::cout << "The spaceship was crushed.\n";
}

void collide(Asteroid const& , Asteroid const& )
{
    std::cout << "The asteroids rebounded.\n";
}

void collide(Spaceship const& , Spaceship const& )
{
    std::cout << "The two spaceships blew up.\n";
}
```

Moments of resolution in C++

Pre-processing	Compilation	Link	Runtime
C++ doesn't have multiple dispatch	Multiple dispatch on templates Multiple dispatch on overloads	?	"C++ doesn't have multiple dispatch" Really?

Double dispatch Visitor-style

```
struct Operation
{
    virtual ~Operation() = 0;
};

struct X : Operation
{



};

struct Y : Operation
{



};

struct Z : Operation
{



};
```

```
struct Object
{

    virtual ~Object() = 0;
};

struct A : Object
{



};

struct B : Object
{



};

struct C : Object
{



};
```

```
Operation const& operation = …
Object const& object = …

perform(operation, object);
```

```
struct Operation
{
    virtual void operateOnA(A const&) const = 0;
    virtual void operateOnB(B const&) const = 0;
    virtual void operateOnC(C const&) const = 0;
    virtual ~Operation() = 0;
};

struct X : Operation
{
    void operateOnA(A const&) const override { std::cout << "X on A\n"; }
    void operateOnB(B const&) const override { std::cout << "X on B\n"; }
    void operateOnC(C const&) const override { std::cout << "X on C\n"; }
};

struct Y : Operation
{
    void operateOnA(A const&) const override { std::cout << "Y on A\n"; }
    void operateOnB(B const&) const override { std::cout << "Y on B\n"; }
    void operateOnC(C const&) const override { std::cout << "Y on C\n"; }
};

struct Z : Operation
{
    void operateOnA(A const&) const override { std::cout << "Z on A\n"; }
    void operateOnB(B const&) const override { std::cout << "Z on B\n"; }
    void operateOnC(C const&) const override { std::cout << "Z on C\n"; }
};
```

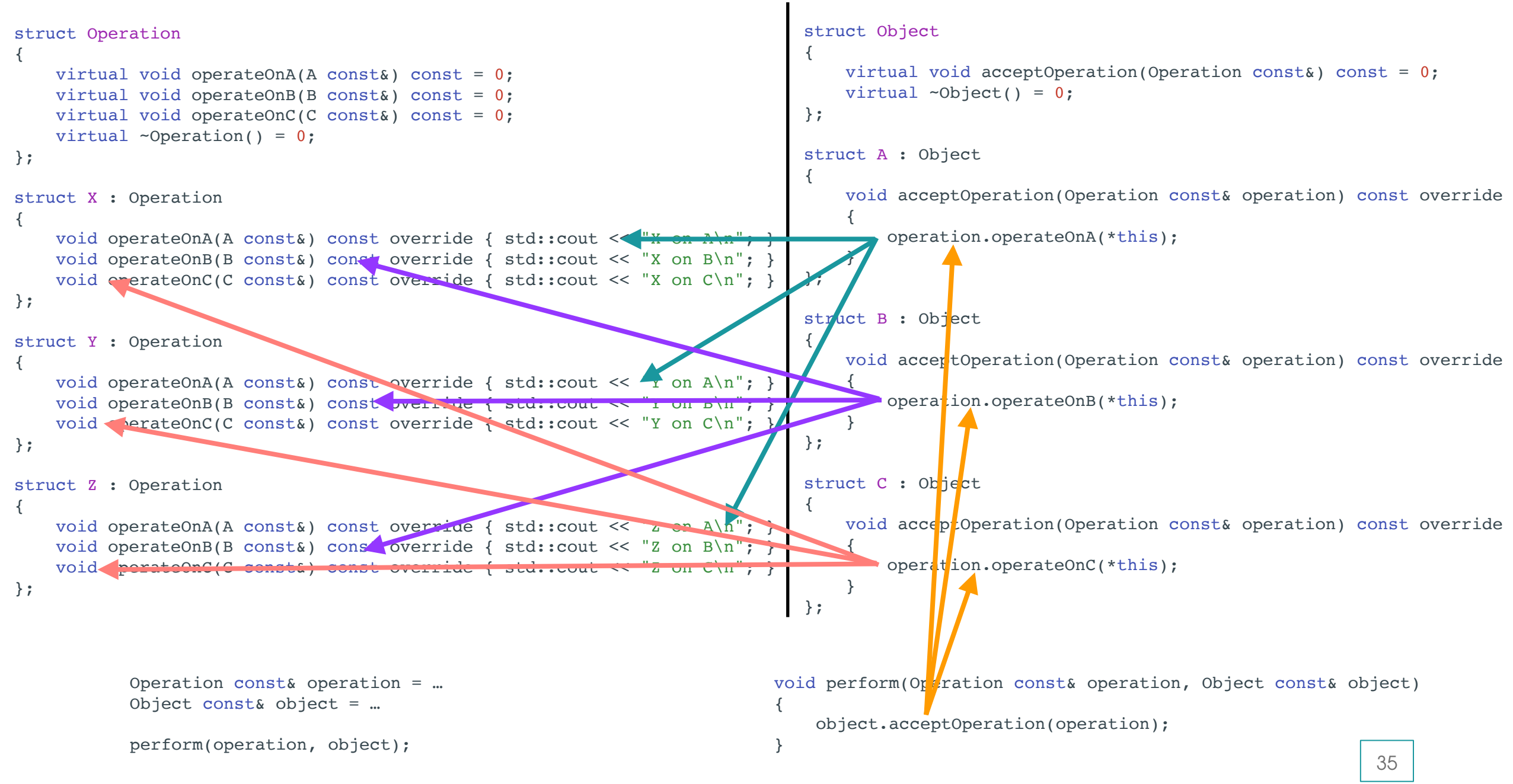
```
struct Object
{

    virtual ~Object() = 0;
};

struct A : Object
{



};

struct B : Object
{



};

struct C : Object
{



};
```

```
Operation const& operation = …
Object const& object = …

perform(operation, object);
```

```
struct Operation
{
    virtual void operateOnA(A const&) const = 0;
    virtual void operateOnB(B const&) const = 0;
    virtual void operateOnC(C const&) const = 0;
    virtual ~Operation() = 0;
};

struct X : Operation
{
    void operateOnA(A const&) const override { std::cout << "X on A\n"; }
    void operateOnB(B const&) const override { std::cout << "X on B\n"; }
    void operateOnC(C const&) const override { std::cout << "X on C\n"; }
};

struct Y : Operation
{
    void operateOnA(A const&) const override { std::cout << "Y on A\n"; }
    void operateOnB(B const&) const override { std::cout << "Y on B\n"; }
    void operateOnC(C const&) const override { std::cout << "Y on C\n"; }
};

struct Z : Operation
{
    void operateOnA(A const&) const override { std::cout << "Z on A\n"; }
    void operateOnB(B const&) const override { std::cout << "Z on B\n"; }
    void operateOnC(C const&) const override { std::cout << "Z on C\n"; }
};
```

```
struct Object
{
    virtual void acceptOperation(Operation const&) const = 0;
    virtual ~Object() = 0;
};

struct A : Object
{
    void acceptOperation(Operation const& operation) const override
    {
        operation.operateOnA(*this);
    }
};

struct B : Object
{
    void acceptOperation(Operation const& operation) const override
    {
        operation.operateOnB(*this);
    }
};

struct C : Object
{
    void acceptOperation(Operation const& operation) const override
    {
        operation.operateOnC(*this);
    }
};
```

```
Operation const& operation = …
Object const& object = …

perform(operation, object);
```



```
struct Operation
{
    virtual void operateOnA(A const&) const = 0;
    virtual void operateOnB(B const&) const = 0;
    virtual void operateOnC(C const&) const = 0;
    virtual ~Operation() = 0;
};

struct X : Operation
{
    void operateOnA(A const&) const override { std::cout << "X on A\n"; }
    void operateOnB(B const&) const override { std::cout << "X on B\n"; }
    void operateOnC(C const&) const override { std::cout << "X on C\n"; }
};

struct Y : Operation
{
    void operateOnA(A const&) const override { std::cout << "Y on A\n"; }
    void operateOnB(B const&) const override { std::cout << "Y on B\n"; }
    void operateOnC(C const&) const override { std::cout << "Y on C\n"; }
};

struct Z : Operation
{
    void operateOnA(A const&) const override { std::cout << "Z on A\n"; }
    void operateOnB(B const&) const override { std::cout << "Z on B\n"; }
    void operateOnC(C const&) const override { std::cout << "Z on C\n"; }
};
```

```
struct Object
{
    virtual void acceptOperation(Operation const&) const = 0;
    virtual ~Object() = 0;
};

struct A : Object
{
    void acceptOperation(Operation const& operation) const override
    {
        operation.operateOnA(*this);
    }
};

struct B : Object
{
    void acceptOperation(Operation const& operation) const override
    {
        operation.operateOnB(*this);
    }
};

struct C : Object
{
    void acceptOperation(Operation const& operation) const override
    {
        operation.operateOnC(*this);
    }
};
```

```
Operation const& operation = …
Object const& object = …

perform(operation, object);
```

```
void perform(Operation const& operation, Object const& object)
{
    object.acceptOperation(operation);
}
```

```
class X
{
public:

    virtual ~X() = 0;
};


class A : public X
{

};

class B : public X
{

};
```

```
A a1;
A a2;

X& xa1 = a1;
X& xa2 = a2;

xa2 = xa1;
```

Does it compile?

What does it do?

What *should* it do?

```
class X
{
public:

    virtual ~X() = 0;
};


class A : public X
{

};

class B : public X
{

};
```

```
A a;
B b;

X& xa = a;
X& xb = b;

xb = xa;
```

Does it compile?

What does it do?

What *should* it do?

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
class A : public X
{
public:
    A& operator=(X const& other) override
    {
        if (auto* aOther = dynamic_cast<A const*>(&other))
        {
            *this = *aOther;
        }
        return *this;
    }
};
```

```
class B : public X
{
public:
    B& operator=(X const& other) override
    {
        if (auto* bOther = dynamic_cast<B const*>(&other))
        {
            *this = *bOther;
        }
        return *this;
    }
};
```

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
class A : public X
{
public:
    A& operator=(X const& other) override
    {
        if (auto* aOther = dynamic_cast<A const*>(&other))
        {
            *this = *aOther;
        }
        else
        {
            // error handling code
        }
        return *this;
    }
};
```

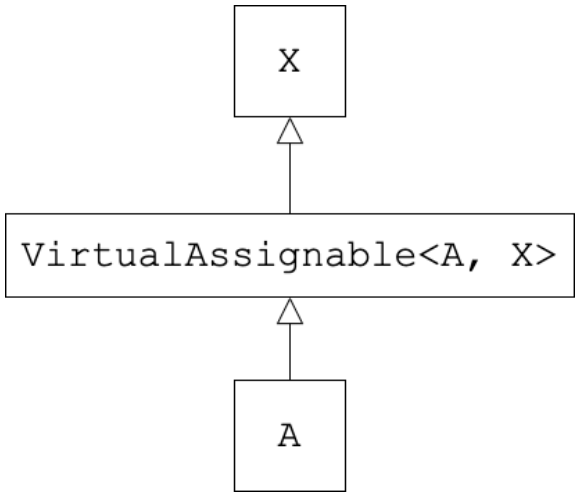
```
class B : public X
{
public:
    B& operator=(X const& other) override
    {
        if (auto* bOther = dynamic_cast<B const*>(&other))
        {
            *this = *bOther;
        }
        else
        {
            // error handling code
        }
        return *this;
    }
};
```

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
class A : public X
{

};
```

```
class B : public X
{

};
```

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual ~X() = 0;
};
```




```
template<typename Derived, typename Base>
struct VirtualAssignable : Base
{
    VirtualAssignable& operator=(Base const& other) override
    {
        auto& thisDerived = static_cast<Derived&>(*this);
        if (auto* otherDerived = dynamic_cast<Derived const*>(&other))
        {
            thisDerived = *otherDerived;
        }
        else
        {
            // error handling code
        }
        return thisDerived;
    }
};
```

```
class A : public XirtualAssignable<A, X>
{

};
```

```
class B : public XirtualAssignable<B, X>
{

};
```

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
class A : public VirtualAssignable<A, X>
{

};
```

```
class B : public VirtualAssignable<B, X>
{

};
```

```
template<typename Derived, typename Base,
template<typename Derived, typename Base>rtCompatibleTypeFailed>
struct VirtualAssignable : Base
{
    VirtualAssignable& operator=(Base const& other) override
    {
        auto& thisDerived = static_cast<Derived&>(*this);
        if (auto* otherDerived = dynamic_cast<Derived const*>(&other))
        {
            thisDerived = *otherDerived;
        }
        else
        {
            // error handling code}();
        }
        return thisDerived;
    }
};
```

```
struct AssertCompatibleTypeFailed
{
    void operator()()
    {
        assert(("Incompatible types for assignment", false));
    }
};
```

```
class X
{
public:
    virtual bool operator==(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
class A : public X
{
    bool operator==(A const& other);
};
```

```
class B : public X
{
    bool operator==(B const& other);
};
```

```
class X
{
public:
    virtual bool operator==(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
template<typename Derived, typename Base>
struct VirtualComparable : Base
{
    bool operator==(real_base const& other) override
    {
        auto& thisDerived = static_cast<Derived&>(*this);
        if (auto* otherDerived = dynamic_cast<Derived const*>(&other))
        {
            return thisDerived == *otherDerived;
        }
        else
        {
            return false;
        }
    }
};
```

```
class A : public VirtualComparable<A, X>
{
    bool operator==(A const& other);
};
```

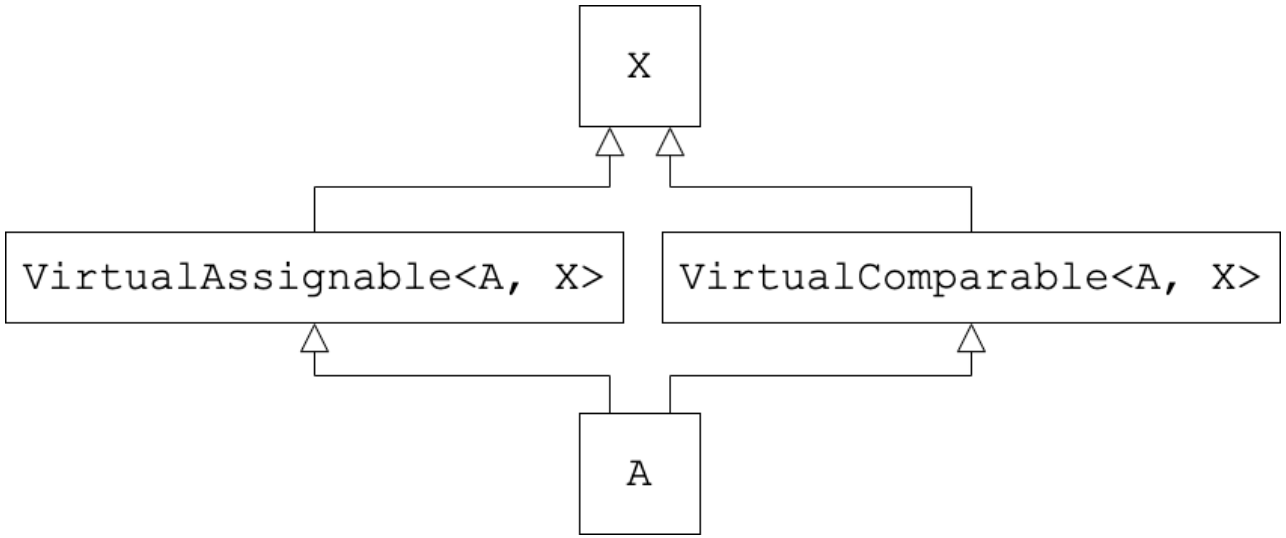
```
class B : public VirtualComparable<B, X>
{
    bool operator==(B const& other);
};
```

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual bool operator==(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
class A : public X
{
    bool operator==(A const& other);
};
```

```
class B : public X
{
    bool operator==(B const& other);
};
```

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual bool operator==(X const& other) = 0;
    virtual ~X() = 0;
};
```

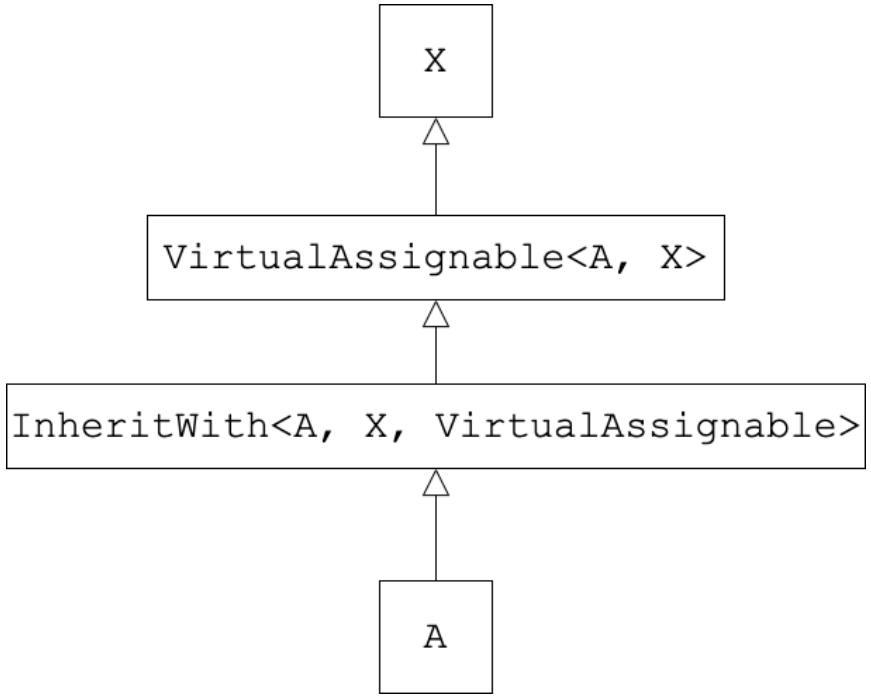



```
class A : public VirtualAssignable<A, X>, public VirtualComparable<A, X>
{               {
                    bool operator==(A const& other);
};              };
```

```
class X
{
public:
    virtual X& operator=(X const& other) = 0;
    virtual bool operator==(X const& other) = 0;
    virtual ~X() = 0;
};
```

```
class A : InheritWith<A, X, VirtualAssignable, VirtualComparable>
{

};
```

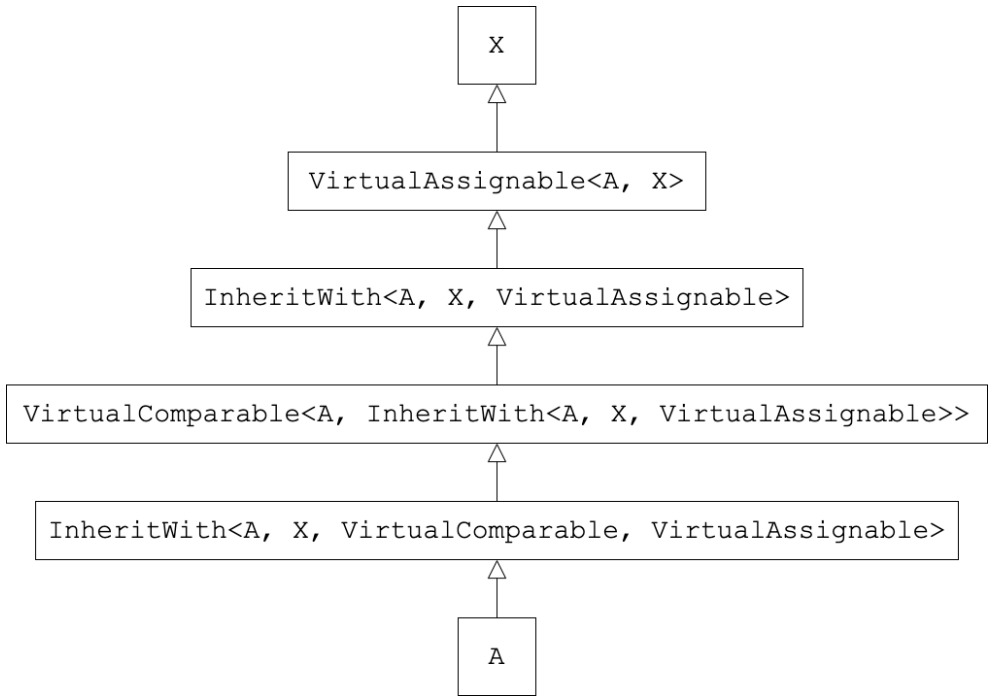
```
template<typename Derived, typename Base,
  template<typename, typename> class VirtualSkill>
struct InheritWith<Derived, Base, VirtualSkill> : VirtualSkill<Derived, Base> {};
```




```
class A : InheritWith<A, X, VirtualAssignable>
{

};
```

```
template<typename Derived, typename Base,
  template<typename, typename> class VirtualSkill>
struct InheritWith<Derived, Base, VirtualSkill> : VirtualSkill<Derived, Base> {};
```

```
template<typename Derived, typename Base,
  template<typename, typename> class VirtualSkill, template<typename, typename> class... VirtualSkills>
struct InheritWith : VirtualSkill<Derived, InheritWith<Derived, Base, VirtualSkills...>> {};
```



bit.do/inheritwith

```
class A : InheritWith<A, X, VirtualAssignable, VirtualComparable>
{

};
```

Scott Meyer's double dispatch

More Effective C++ — Item 31 (in essence):

```
class GameObject
{
    virtual ~GameObject(){}
};

class Asteroid : public GameObject
{

};

class Spaceship : public GameObject
{

};
```

```
CollisionFunctions collisionFunctions;

collisionFunctions.registerFunction<Asteroid, Asteroid>(
 [](GameObject&, GameObject&){ std::cout << "The two asteroids rebounded.\n"; });

collisionFunctions.registerFunction<Asteroid, Spaceship>(
 [](GameObject&, GameObject&){ std::cout << "The spaceship was crushed.\n"; });

collisionFunctions.registerFunction<Spaceship, Spaceship>(
 [](GameObject&, GameObject&){ std::cout << "The two spaceships blew up.\n"; });


Asteroid asteroid;
Spaceship spaceship;

collisionFunctions.call(asteroid, asteroid);
collisionFunctions.call(asteroid, spaceship);
collisionFunctions.call(spaceship, asteroid);
collisionFunctions.call(spaceship, spaceship);
```

```
class CollisionFunctions
{
public:
    template<typename Type1, typename Type2, typename CollisionFunction>
    void registerFunction(CollisionFunction collisionFunction)
    {
        collisionFunctions_[std::pair(&typeid(Type1), &typeid(Type2))] = collisionFunction;
    }

    void call(GameObject& GameObject1, GameObject& GameObject2)
    {
        auto collisionFunction = collisionFunctions_[std::pair(&typeid(GameObject1), &typeid(GameObject2))];
        if (!collisionFunction)
        {
            collisionFunction = collisionFunctions_[std::pair(&typeid(GameObject2), &typeid(GameObject1))];
        }
        collisionFunction(GameObject1, GameObject2);
    }

private:
    std::map<std::pair<const std::type_info*, const std::type_info*>, void(*)(GameObject&, GameObject&)>
collisionFunctions_;
};
```

Scott Meyer's double dispatch

More Effective C++ — Item 31 (in essence):

```
class GameObject
{
    virtual ~GameObject(){}
};

class Asteroid : public GameObject
{

};

class Spaceship : public GameObject
{

};
```

```
CollisionFunctions collisionFunctions;

collisionFunctions.registerFunction<Asteroid, Asteroid>(
 [](GameObject&, GameObject&){ std::cout << "The two asteroids rebounded.\n"; });
collisionFunctions.registerFunction<Asteroid, Spaceship>(
 [](GameObject&, GameObject&){ std::cout << "The spaceship was crushed.\n"; });
collisionFunctions.registerFunction<Spaceship, Spaceship>(
 [](GameObject&, GameObject&){ std::cout << "The two spaceships blew up.\n"; });

Asteroid asteroid;
Spaceship spaceship;

collisionFunctions.call(asteroid, asteroid);
collisionFunctions.call(asteroid, spaceship);
collisionFunctions.call(spaceship, asteroid);
collisionFunctions.call(spaceship, spaceship);
```

```
The two asteroids rebounded.
The spaceship was crushed.
The spaceship was crushed.
The two spaceships blew up.
```

Multiple dispatch in C++

Pre-processing	Compilation	Link	Runtime
C++ doesn't have multiple dispatch	Multiple dispatch on templates Multiple dispatch on overloads	?	"C++ doesn't have multiple dispatch" Really? Visitor style `VirtualAssignable` type_info -> function

The bigger picture of polymorphism

Nature of polymorphism

“f has a polymorphic parameter”

```
f(x);
```

x can be…

- … a value of a fixed type
- … a value of various types
- … a function
- … a group of functions

Nature of polymorphism

"f has a polymorphic parameter"

```
f(x);
```

x can be…

- … a value of a fixed type
- … a value of various types
- … a function
- … a group of functions

A group of functions

Color change:

```
RGB computeColor(RGB inputColor);

void log(RGB inputColor, RGB outputColor);
```

```
struct RGB
{
    int r;
    int g;
    int b;
};
```

A group of functions

Brighten:

```
RGB computeColor(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = std::min(255., inputColor.r * 1.1);
    outputColor.g = std::min(255., inputColor.g * 1.1);
    outputColor.b = std::min(255., inputColor.b * 1.1);

    return outputColor;
}

void log(RGB inputColor, RGB outputColor)
{
    std::cout << "The color was brightened from "
              << to_string(inputColor)
              << " to "
              << to_string(outputColor);
}
```

Darken:

```
RGB computeColor(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = inputColor.r / 1.1;
    outputColor.g = inputColor.g / 1.1;
    outputColor.b = inputColor.b / 1.1;

    return outputColor;
}

void log(RGB inputColor, RGB outputColor)
{
    std::cout << "The color was darkened from "
              << to_string(inputColor)
              << " to "
              << to_string(outputColor);
}
```

Nature of polymorphism

“f has a polymorphic parameter”

```
f(x);
```

x can be…

- … a value of a fixed type
- … a value of various types
- … a function
- … a group of functions

Nature of polymorphism

Pre-processing

- Value of various types

```
#define type X
```

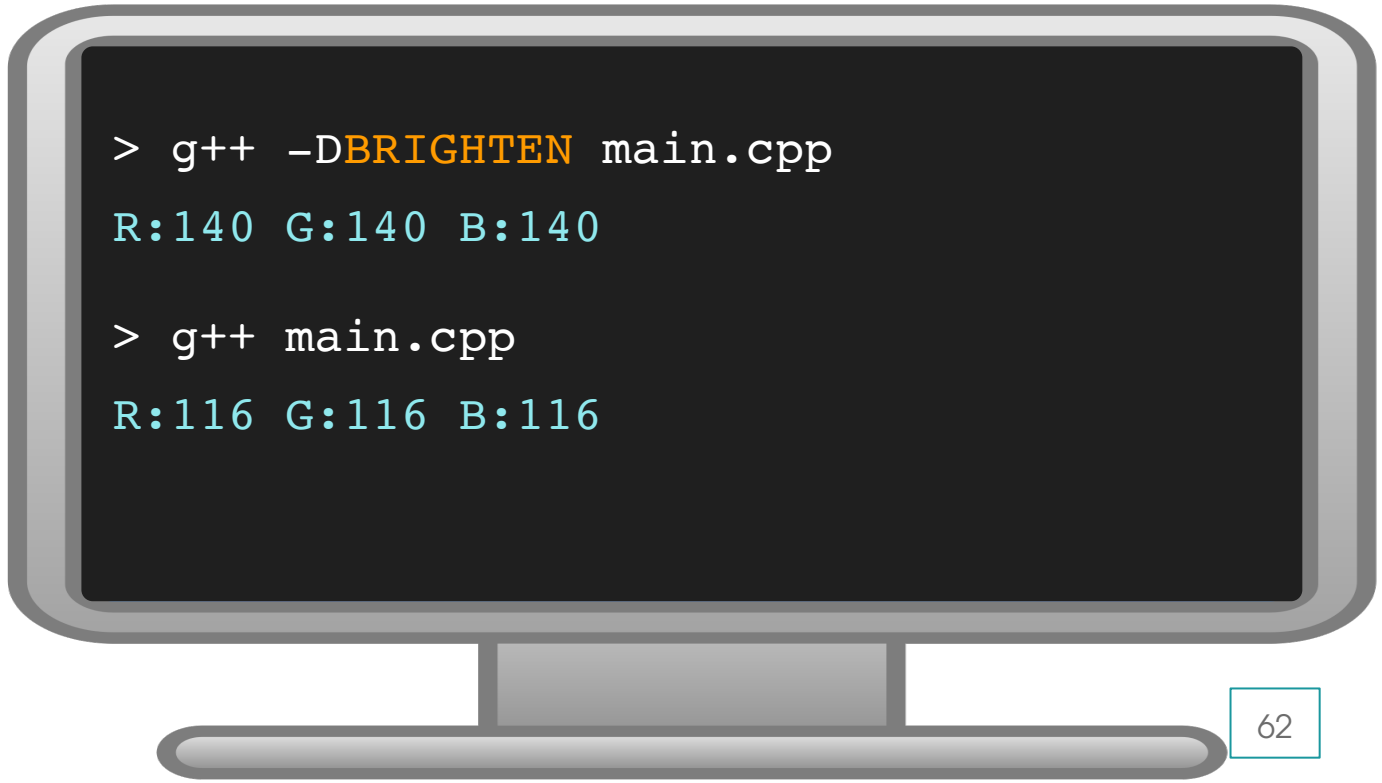
- Function?

```cpp
RGB brighten(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = std::min(255., inputColor.r * 1.1);
    outputColor.g = std::min(255., inputColor.g * 1.1);
    outputColor.b = std::min(255., inputColor.b * 1.1);

    return outputColor;
}
```

```cpp
RGB darken(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = inputColor.r / 1.1;
    outputColor.g = inputColor.g / 1.1;
    outputColor.b = inputColor.b / 1.1;

    return outputColor;
}
```

```cpp
#ifdef BRIGHTEN
    #define computeColor brighten
#else
    #define computeColor darken
#endif
```

```cpp
int main()
{
    auto const color = RGB{128,128,128};
    std::cout << computeColor(color) << '\n';
}
```



Nature of polymorphism

Pre-processing

- Value of various types

```
#define type X
```

- Function

```
#define function f
```

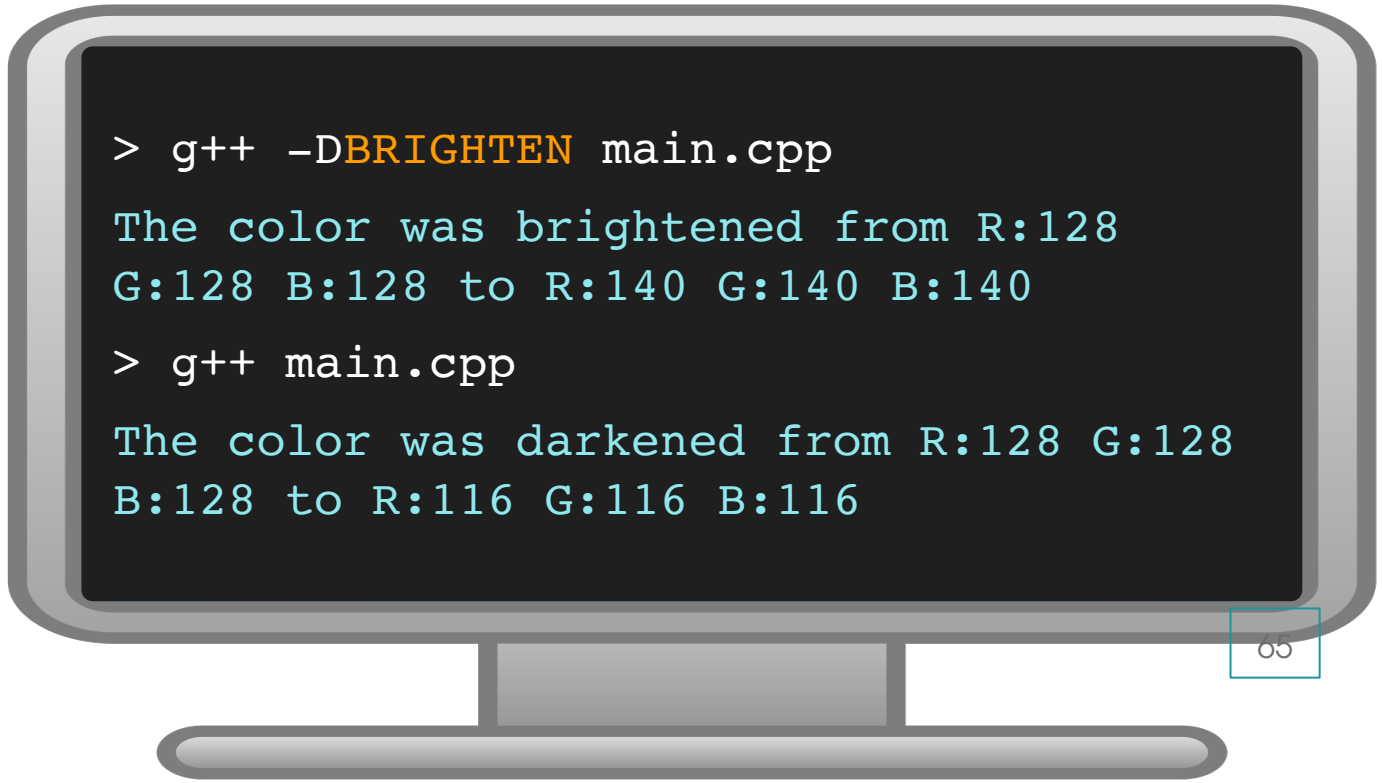
- Group of functions?

A group of functions

```
struct Brighten
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = std::min(255., inputColor.r * 1.1);
        outputColor.g = std::min(255., inputColor.g * 1.1);
        outputColor.b = std::min(255., inputColor.b * 1.1);

        return outputColor;
    }

    static void log(RGB intputColor, RGB outputColor)
    {
        std::cout << "The color was brightened from "
                  << to_string(intputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
struct Darken
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = inputColor.r / 1.1;
        outputColor.g = inputColor.g / 1.1;
        outputColor.b = inputColor.b / 1.1;

        return outputColor;
    }

    static void log(RGB intputColor, RGB outputColor)
    {
        std::cout << "The color was darkened from "
                  << to_string(intputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
#ifdef BRIGHTEN
    #define ColorChange Brighten
#else
    #define ColorChange Darken
#endif

int main()
{
    auto const input = RGB{128,128,128};
    ColorChange::log(input, ColorChange::computeColor(input));
}
```



A group of functions

```
struct Brighten
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = std::min(255., inputColor.r * 1.1);
        outputColor.g = std::min(255., inputColor.g * 1.1);
        outputColor.b = std::min(255., inputColor.b * 1.1);

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was brightened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

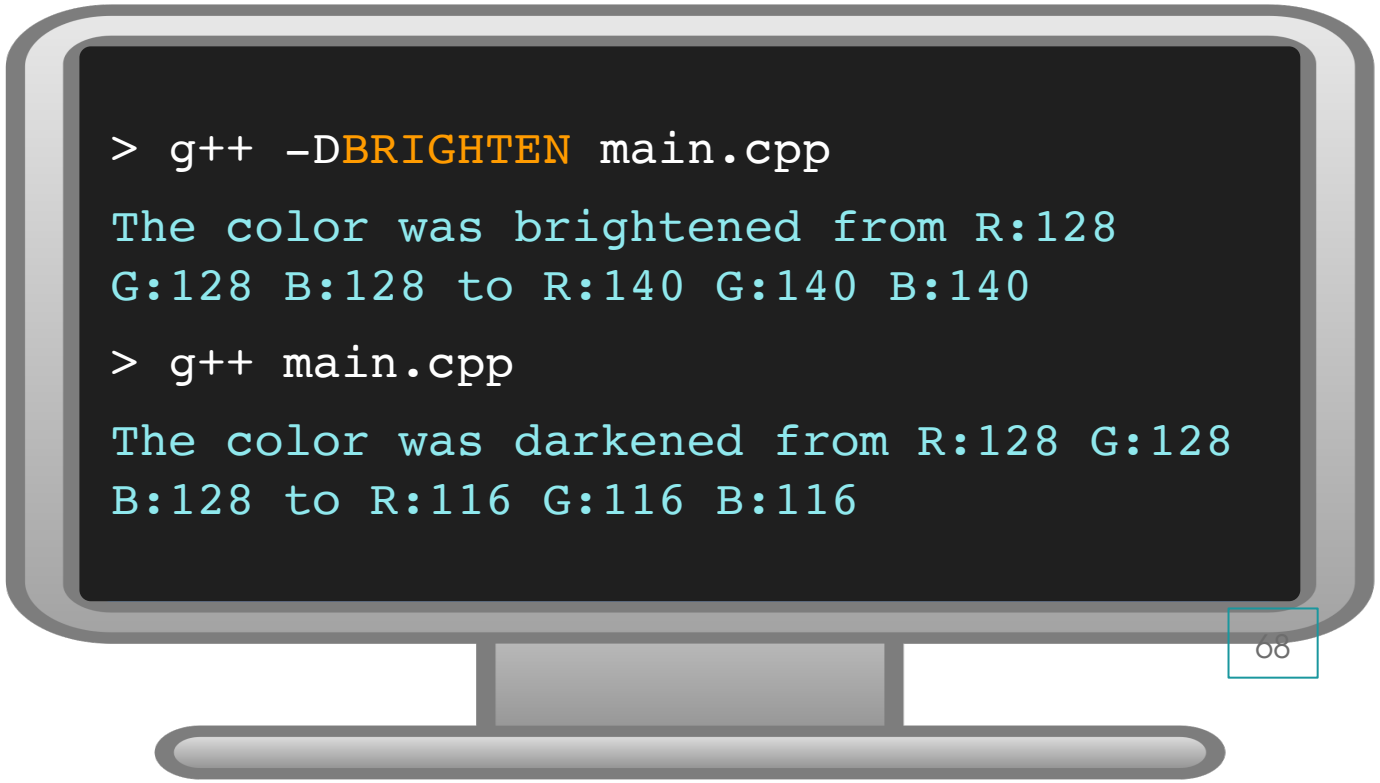
```
struct Darken
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = inputColor.r / 1.1;
        outputColor.g = inputColor.g / 1.1;
        outputColor.b = inputColor.b / 1.1;

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was darkened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

A group of functions

```
namespace Brighten
{
    RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = std::min(255., inputColor.r * 1.1);
        outputColor.g = std::min(255., inputColor.g * 1.1);
        outputColor.b = std::min(255., inputColor.b * 1.1);

        return outputColor;
    }

    void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was brightened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
namespace Darken
{
    RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = inputColor.r / 1.1;
        outputColor.g = inputColor.g / 1.1;
        outputColor.b = inputColor.b / 1.1;

        return outputColor;
    }

    void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was darkened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
#ifdef BRIGHTEN
    #define ColorChange Brighten
#else
    #define ColorChange Darken
#endif

int main()
{
    auto const input = RGB{128,128,128};
    ColorChange::log(input, ColorChange::computeColor(input));
}
```



Nature of polymorphism

Pre-processing

- Value of various types

```
#define type X
```

- Function

```
#define function f
```

- Group of functions

```
#define type X
#define Namespace N
```

- Value of various types

  Template types

```
struct Square
{
    double side_;

    void draw() const
    {
        std::cout << "This is a square.\n";
    }
};

struct LightningBolt
{
    void draw() const
    {
        std::cout << "This is a lightning bolt.\n";
    }
};
```

```
template<typename Shape>
void f(Shape const& shape)
{
    g(shape);
}
```

```
void g(Square const& square)
{
    std::cout << square.side_ << '\n';
    square.draw();
}

void g(LightningBolt const& lightningBolt)
{
    lightningBolt.draw();
}
```

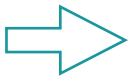
Nature of polymorphism

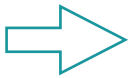
Compilation

- Value of various types
  - Template type
  - Overloading
- Function?

```
RGB brighten(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = std::min(255., inputColor.r * 1.1);
    outputColor.g = std::min(255., inputColor.g * 1.1);
    outputColor.b = std::min(255., inputColor.b * 1.1);

    return outputColor;
}

RGB darken(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = inputColor.r / 1.1;
    outputColor.g = inputColor.g / 1.1;
    outputColor.b = inputColor.b / 1.1;

    return outputColor;
}
```

```
template<int(* computeColor)(int)>
int f()
{
    return computeColor(RGB{128,128,128});
}
```

`f<brighten>()`  R:140 G:140 B:140

`f<darken>()`  R:116 G:116 B:116

Nature of polymorphism

Compilation

- Value of various types
  - Template type
  - Overloading
- Function
  - Template function parameter
- Group of functions?

```cpp
struct Brighten
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = std::min(255., inputColor.r * 1.1);
        outputColor.g = std::min(255., inputColor.g * 1.1);
        outputColor.b = std::min(255., inputColor.b * 1.1);

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was brightened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```cpp
struct Darken
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = inputColor.r / 1.1;
        outputColor.g = inputColor.g / 1.1;
        outputColor.b = inputColor.b / 1.1;

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was darkened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```cpp
template<typename ColorChange>
void f()
{
    auto const input = RGB{128,128,128};
    auto const output = ColorChange::compute(input);
    ColorChange::log(input, output);
}
```

`f<Brighten>()` ⇨ R:140 G:140 B:140

`f<Darken>()` ⇨ R:116 G:116 B:116

Nature of polymorphism

Pre-processing

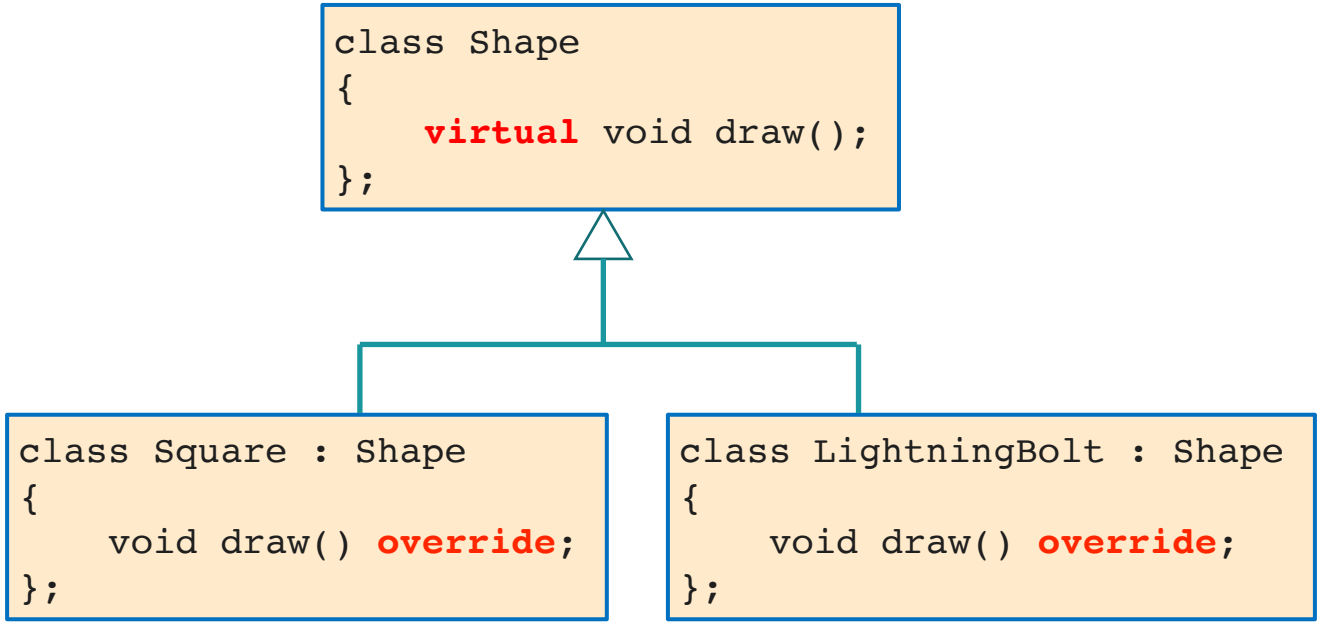
Compilation

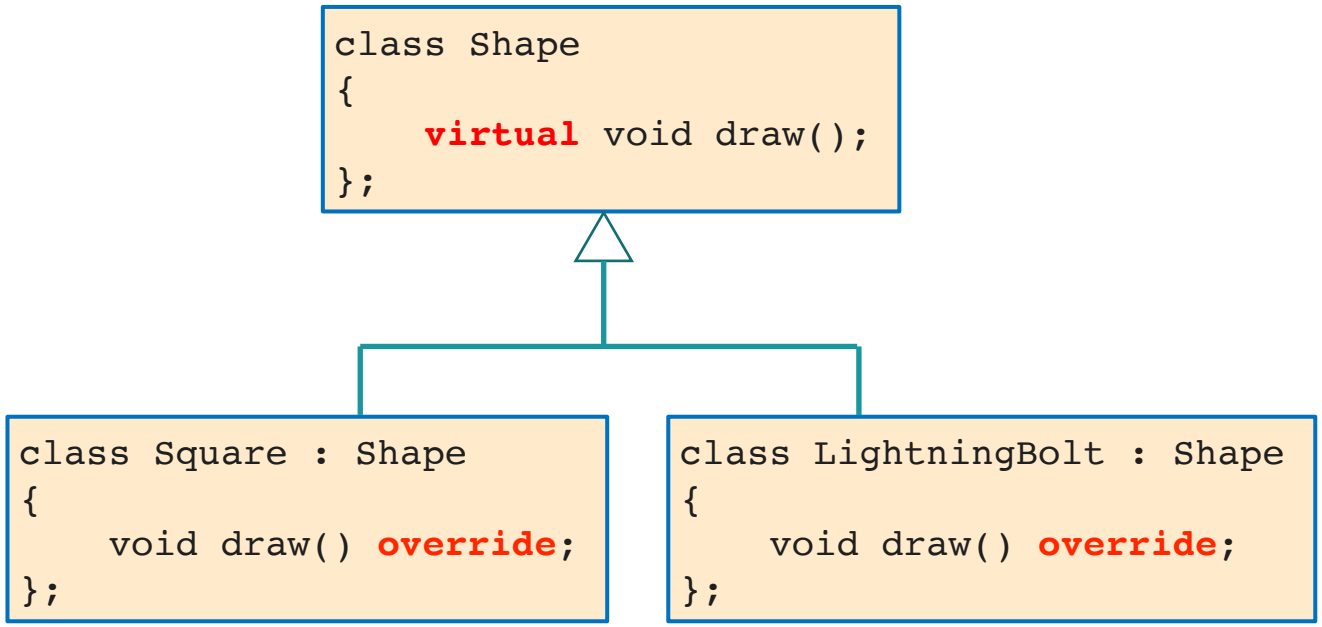
- Value of various types
  - Template type
  - Overloading
- Function
  - Template function parameter
- Group of functions
  - Template type

Runtime

Linking with other files

- Value of various types
  - Inheritance + virtual


class Shape
{
virtual void draw();
};
class Square : Shape
{
void draw() override;
};
class LightningBolt : Shape
{
void draw() override;
};


class Shape
{
virtual void draw();
};
class Square : Shape
{
void draw() override;
};
class LightningBolt : Shape
{
void draw() override;
};

```
struct Shape
{
    virtual void draw() const = 0;
    virtual ~Shape();
};

struct Square : Shape
{
    void draw() const override;
};

struct LightningBolt : Shape
{
    void draw() const override;
};
```

```
std::unique_ptr<Shape> createShape()
{
    return std::make_unique<Square>();
}
```

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

shapes.push_back(std::make_unique<Square>());
shapes.push_back(std::make_unique<LightningBolt>());
```

```
struct IShape
{
    virtual void draw() const;
    virtual ~Shape();
};

struct Square : IShape
{
    void draw() const override;
};

struct LightningBolt : IShape
{
    void draw() const override;
};
```

```
class Shape
{
public:
    template<typename ConcreteShape>
    Shape(ConcreteShape&& shape)
    : storage{std::forward<ConcreteShape>(shape)}
    , getter{ [](std::any &storage) -> IShape&
              {
                  return std::any_cast<ConcreteShape&>(storage);
              }
            }
    {}

    IShape *operator->() { return &getter(storage); }

private:
    std::any storage;
    IShape& (*getter)(std::any&);
};
```

```
class Shape
{
public:
    template<typename ConcreteShape>
    Shape(ConcreteShape&& shape)
    : storage{std::forward<ConcreteShape>(shape)}
    , getter{ [](std::any &storage) -> IShape&
              {
                  return std::any_cast<ConcreteShape&>(storage);
              }
            }
    {}

    IShape *operator->() { return &getter(storage); }

private:
    std::any storage;
    IShape& (*getter)(std::any&);
};
```

```
class Shape
{
public:
    template<typename ConcreteShape>
    Shape(ConcreteShape&& shape)
    : storage{std::forward<ConcreteShape>(shape)}
    , getter{ [](std::any &storage) -> IShape&
              {
                  return std::any_cast<ConcreteShape&>(storage);
              }
            }
    {}

    IShape *operator->() { return &getter(storage); }

private:
    std::any storage;
    IShape& (*getter)(std::any&);
};
```

```
class Shape
{
public:
    template<typename ConcreteShape>
    Shape(ConcreteShape&& shape)
    : storage{std::forward<ConcreteShape>(shape)}
    , getter{ [](std::any &storage) -> IShape&
              {
                  return std::any_cast<ConcreteShape&>(storage);
              }
            }
    {}

    IShape *operator->() { return &getter(storage); }

private:
    std::any storage;
    IShape& (*getter)(std::any&);
};
```

```
class Shape
{
public:
    template<typename ConcreteShape>
    Shape(ConcreteShape&& shape)
    : storage{std::forward<ConcreteShape>(shape)}
    , getter{ [](std::any &storage) -> IShape&
              {
                  return std::any_cast<ConcreteShape&>(storage);
              }
            }
    {}

    IShape *operator->() { return &getter(storage); }

private:
    std::any storage;
    IShape& (*getter)(std::any&);
};
```

```
struct IShape
{
    virtual void draw() const;
    virtual ~Shape();
};

struct Square : IShape
{
    void draw() const override;
};

struct LightningBolt : IShape
{
    void draw() const override;
};
```

```
class Shape
{
public:
    template<typename ConcreteShape>
    Shape(ConcreteShape&& shape)
    : storage{std::forward<ConcreteShape>(shape)}
    , getter{ [](std::any &storage) -> IShape&
              {
                  return std::any_cast<ConcreteShape&>(storage);
              }
            }
    {}

    IShape *operator->() { return &getter(storage); }

private:
    std::any storage;
    IShape& (*getter)(std::any&);
};
```

```
Shape createShape()ape> createShape()
{
    return Square{};_unique<Square>();
}
```

```
  std::vector<Shape> shapes;r<IShape>> shapes;

  shapes.push_back(Square{});unique<Square>());
  shapes.push_back(LightningBolt{});LightningBolt>());
```

```
std::unique_ptr<Shape> createShape()
{
    return std::make_unique<Square>();
}
```

```
auto shape = createShape();
shape->draw();
```

Maybe there was a crash here

POINTER SEMANTICS

?

NULLABLE

```
Shape createShape()
{
    return Square{};
}
```

```
auto shape = createShape();
shape->draw();
```

VALUE SEMANTICS

```
auto otherShape = shape;
```

NOT NULLABLE

`operator->` doesn't always mean pointer

IShape: Interface
Shape: Implementation
ConcreteShape: ConcreteType
shape: object

```
class Shape
{
public:
    template<typename ConcreteShape>
    Shape(ConcreteShape&& shape)
    : storage{std::forward<ConcreteShape>(shape)}
    , getter{ [](std::any& storage) -> IShape&
              {
                  return std::any_cast<ConcreteShape&>(storage);
              }
            }
    {}

    IShape *operator->() { return &getter(storage); }

private:
    std::any storage;
    IShape& (*getter)(std::any&);
};
```

```
template<typename Interface>
class Implementation
{
public:
    template<typename ConcreteType>
    Implementation(ConcreteType&& object)
    : storage{std::forward<ConcreteType>(object)}
    , getter{ [](std::any& storage) -> Interface&
              {
                  return std::any_cast<ConcreteType&>(storage);
              }
            }
    {}

    Interface *operator->() { return &getter(storage); }

private:
    std::any storage;
    Interface& (*getter)(std::any&);
};
```

```
using Shape = Implementation<IShape>;
```

Nature of polymorphism

Pre-processing

- Value of various types

```
#define type X
```

- Function

```
#define function f
```

- Group of functions

```
#define type X
#define Namespace N
```

Compilation

- Value of various types
  - Template type
  - Overloading
- Function
  - Template function parameter
- Group of functions
  - Template type

Link

Linking with other files

Runtime

- Value of various types
  - Inheritance + virtual
    - Pointer semantics
    - Value semantics

```
std::variant?
```

```
struct Square
{

};

struct LightningBolt
{

};
```

```
using Shape = std::variant<Square, LightningBolt>;
```

```
template<typename... Functions>
struct overload : Functions...
{
    using Functions::operator()...;
    overload(Functions... functions) : Functions(functions)...{}
};
```

```
void draw(Shape const& shape)
{
    std::visit(overload(
                [](Square const& square){ std::cout << "This is a square.\n"; },
                [](LightningBolt const& lightningBolt){ std::cout << "This is a lightning bolt.\n"; }),
                shape);
}
```

What is the interface?

```
struct Square
{
    void draw() const;
};

struct LightningBolt
{
    void draw() const;
};
```

```
using Shape = std::variant<Square, LightningBolt>;
```

```
template<typename... Functions>
struct overload : Functions...
{
    using Functions::operator()...;
    overload(Functions... functions) : Functions(functions)...{}
};
```

```
void draw(Shape const& shape)
{
    std::visit(overload(
                [](Square const& square){ square.draw(); },
                [](LightningBolt const& lightningBolt){ lightningBolt.draw() }),
                shape);
}
```

Independent client code?

```
struct Square
{
    void draw() const;
};

struct LightningBolt
{
    void draw() const;
};
```

```
using Shape = std::variant<Square, LightningBolt>;
```

```
template<typename... Functions>
struct overload : Functions...
{
    using Functions::operator()...;
    overload(Functions... functions) : Functions(functions)...{}
};
```

```
void draw(Shape const& shape)
{
    std::visit([](auto&& shape){ shape.draw(); }, shape);
}
```

Nature of polymorphism

Pre-processing

- Value of various types

```
#define type X
```

- Function

```
#define function f
```

- Group of functions

```
#define type X
#define Namespace N
```

Compilation

- Value of various types
  - Template type
  - Overloading
- Function
  - Template function parameter
- Group of functions
  - Template type

Link

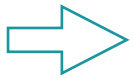
Linking with other files

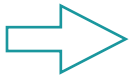
Runtime

- Value of various types
  - Inheritance + virtual
    - Pointer semantics
    - Value semantics
  - `std::variant`
- Function?

```
template<typename Function>
RGB f(RGB(* computeColor)(RGB))
{
    auto const input = RGB{128,128,128};
    return computeColor(input);
}
```

```
RGB brighten(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = std::min(255., inputColor.r * 1.1);
    outputColor.g = std::min(255., inputColor.g * 1.1);
    outputColor.b = std::min(255., inputColor.b * 1.1);

    return outputColor;
}

RGB darken(RGB inputColor)
{
    RGB outputColor{};
    outputColor.r = inputColor.r / 1.1;
    outputColor.g = inputColor.g / 1.1;
    outputColor.b = inputColor.b / 1.1;

    return outputColor;
}
```

`f(brighten)`  R:140 G:140 B:140

`f(darken)`  R:116 G:116 B:116

Nature of polymorphism

Pre-processing

- Value of various types

```
#define type X
```

- Function

```
#define function f
```

- Group of functions

```
#define type X
#define Namespace N
```

Compilation

- Value of various types
  - Template type
  - Overloading
- Function
  - Template function parameter
- Group of functions
  - Template type

Link

Linking with other files

Runtime

- Value of various types
  - Inheritance + virtual
    - Pointer semantics
    - Value semantics
  - `std::variant`
- Function
  - Function parameter
- Group of functions?

```
struct Brighten
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = std::min(255., inputColor.r * 1.1);
        outputColor.g = std::min(255., inputColor.g * 1.1);
        outputColor.b = std::min(255., inputColor.b * 1.1);

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was brightened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
struct Darken
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = inputColor.r / 1.1;
        outputColor.g = inputColor.g / 1.1;
        outputColor.b = inputColor.b / 1.1;

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was darkened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
void f(ColorChange const& colorChange)
{
    auto const input = RGB{128,128,128};
    auto const output = colorChange.computeColor(input);
    colorChange.log(input, output);
}
```

```
struct ColorChange
{
    RGB (*computeColor)(RGB input);
    void (*log)(RGB input, RGB output);
};
```

```
struct Brighten
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = std::min(255., inputColor.r * 1.1);
        outputColor.g = std::min(255., inputColor.g * 1.1);
        outputColor.b = std::min(255., inputColor.b * 1.1);

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was brightened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
struct Darken
{
    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = inputColor.r / 1.1;
        outputColor.g = inputColor.g / 1.1;
        outputColor.b = inputColor.b / 1.1;

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was darkened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
void f(ColorChange const& colorChange)
{
    auto const input = RGB{128,128,128};
    auto const output = colorChange.computeColor(input);
    colorChange.log(input, output);
}
```

```
struct ColorChange
{
    RGB (*computeColor)(RGB input);
    void (*log)(RGB input, RGB output);

    template<typename ColorChangeImplementation>
    static ColorChange createFrom()
    {
        return ColorChange{ &ColorChangeImplementation::computeColor,
                            &ColorChangeImplementation::log };
    }
};
```

```
auto const colorChange = ColorChange::createFrom<Brighten>();
f(colorChange);
```

```
struct Brighten
{
    static bool handles(RGB input)
    {
        return input.r < 128 && input.g < 128 && input.b < 128;
    }

    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = std::min(255., inputColor.r * 1.1);
        outputColor.g = std::min(255., inputColor.g * 1.1);
        outputColor.b = std::min(255., inputColor.b * 1.1);

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was brightened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }
};
```

```
struct Darken
{
    static bool handles(RGB input)
    {
        return input.r > 128 && input.g > 128 && input.b > 128;
    }

    static RGB computeColor(RGB inputColor)
    {
        RGB outputColor{};
        outputColor.r = inputColor.r / 1.1;
        outputColor.g = inputColor.g / 1.1;
        outputColor.b = inputColor.b / 1.1;

        return outputColor;
    }

    static void log(RGB inputColor, RGB outputColor)
    {
        std::cout << "The color was darkened from "
                  << to_string(inputColor)
                  << " to "
                  << to_string(outputColor);
    }};
```

```
struct ColorChange
{
    RGB (*computeColor)(RGB input);
    void (*log)(RGB input, RGB output);
    bool (*handles) (RGB input);

    template<typename ComputeColorImplementation>
    static ColorChange createFrom()
    {
        return ColorChange { &ComputeColorImplementation::computeColor,
                             &ComputeColorImplementation::log,
                             &ComputeColorImplementation::handles };
    }
};
```

```
std::vector<ColorChange> getColorChanges()
{
   return {
       ColorChange::createFrom<Brighten>(),
       ColorChange::createFrom<Darken>()
       };
}
```

```
auto const input = RGB{50,50,50};

auto const colorChanges = getColorChanges();
auto const colorChange = std::ranges::find_if(colorChanges,
                [&input](auto&& colorChange){ return colorChange.handles(input); });

if (colorChange != end(colorChanges))
{
    auto const output = colorChange->computeColor(input);
    colorChange->log(input, output);
}
```

```
The color was brightened from R:50 G:50 B:50 to R:55 G:55 B:55
```

```
struct ColorChange
{
    RGB (*computeColor)(RGB input);
    void (*log)(RGB input, RGB output);
    bool (*handles) (RGB input);

    template<typename ComputeColorImplementation>
    static ComputeColor createFrom()
    {
        return ComputeColor{ &ComputeColorImplementation::computeColor,
                             &ComputeColorImplementation::log,
                             &ComputeColorImplementation::handles };
    }
};
```

```
std::vector<ColorChange> getColorChanges()
{
   return {
       ColorChange::createFrom<Brighten>(),
       ColorChange::createFrom<Darken>()
       };
}
```

```
auto const input = RGB{200,200,200};

auto const colorChanges = get colorChanges();
auto const colorChange = std::ranges::find_if(colorChanges,
                [&input](auto&& colorChange){ return colorChange.handles(input); });

if (colorChange != end(colorChanges))
{
    auto const output = colorChange->computeColor(input);
    colorChange->log(input, output);
}
```

```
The color was darkened from R:200 G:200 B:200 to R:181 G:181 B:181
```

No heap allocation

Semantics of group of function

Nature of polymorphism

Pre-processing

- Value of various types

```
#define type X
```

- Function

```
#define function f
```

- Group of functions

```
#define type X
#define Namespace N
```

Compilation

- Value of various types
  - Template type
  - Overloading
- Function
  - Template function parameter
- Group of functions
  - Template type

Link

Linking with other files

Runtime

- Value of various types
  - Inheritance + virtual
    - Pointer semantics
    - Value semantics
  - `std::variant`
- Function
  - Function parameter
- Group of functions

Function pointers interface

Explicit/Implicit interface

Pre-processing

Macros:

IMPLICIT

Compilation

Templates:
(without concepts)

IMPLICIT

With concepts:

EXPLICIT

Link

Linking with other files:

IMPLICIT

Runtime

Inheritance + virtual:

EXPLICIT

`std::variant`

IMPLICIT

Function pointer:

EXPLICIT

Function pointers interface:

EXPLICIT

Closed/Open interface

Pre-processing

Macros:
CLOSED

Compilation

Templates:
OPEN

Overloading:
OPEN

Link

Linking with other files:
OPEN

Runtime

Inheritance + virtual:
OPEN

`std::variant`
CLOSED

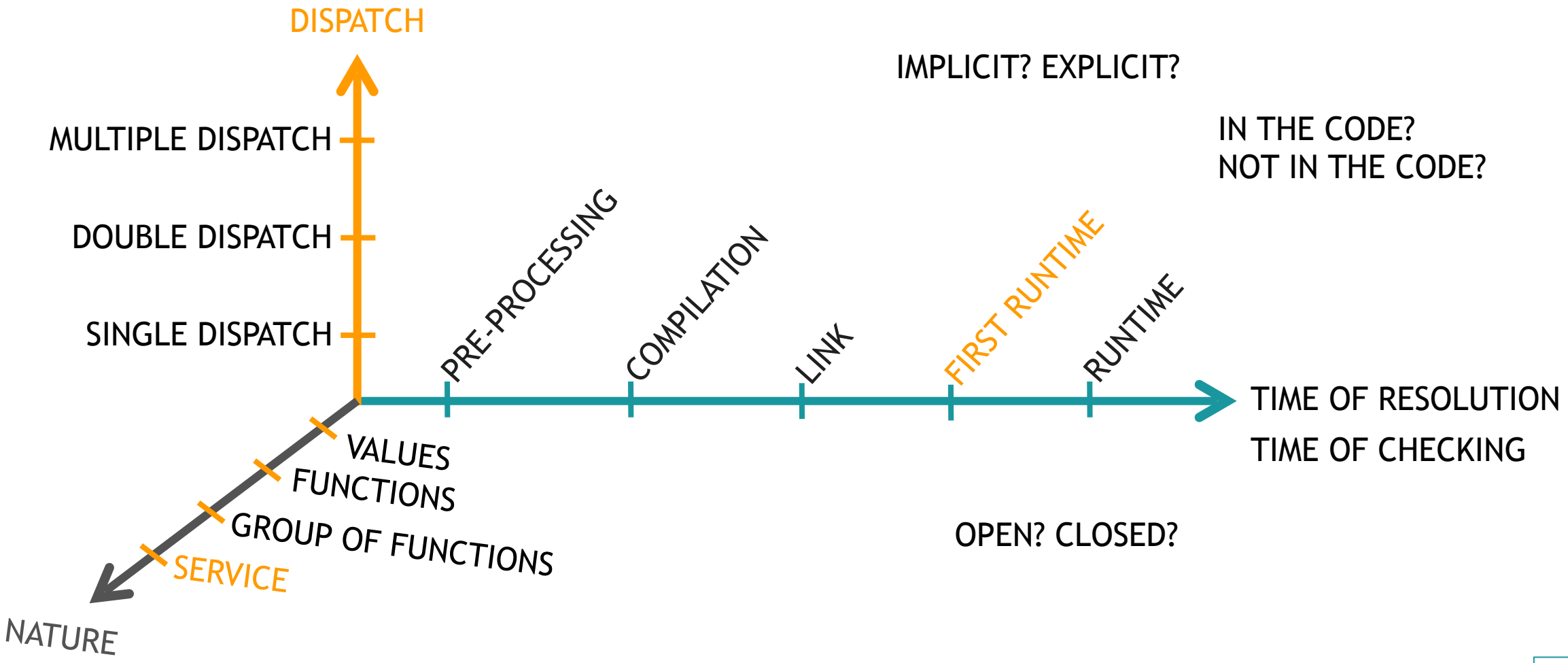
Function pointer:
OPEN

Function pointers interface:
OPEN

Time of checking

Pre-processing	Compilation	Link	Runtime
Compile time	Compile time	Compile time	Compile time

The bigger picture of polymorphism



There is more than one type of polymorphism. Think about which one to use.

@JoBoccara jonathan@fluentcpp.com

META
POLYMORPHISM

Special thanks
Reviewers
Benjamin Wagner
Rajaram Nowakgr
Fred Tingaud
Matthew Guidry
Andreas Buhr