Object-Oriented Programming in Modern C++

by Borislav Stanimirov / @stanimirovb

Hello, World

```cpp
#include <iostream>

int main()
{
    std::cout << "Hi, I'm Borislav!\n";
    std::cout << "These slides are here: https://is.gd/o
    return 0;
}
```

Hello, World

```cpp
#include <iostream>

int main()
{
    std::cout << "Hi, I'm Borislav!\n";
    std::cout << "These slides are here: https://is.gd/o
    return 0;
}
```

Bulgaria


Europe
International Boundary
Road
River
National Capital
City or Town
0 250 500 KM
0 250 500 Miles
© 2007 Geology.com
BULGARIA
Sofia

Borislav Stanimirov

- Mostly a **C++** programmer
- Mostly a **game** programmer
- Recently working on **medical software**
- **Open-source** programmer
- **github.com/iboB**

Business Logic

A part of a program which deals with the real world rules that determine how data is obtained, stored and processed.

The part which deals with the software's purpose

```
int a;
int b;
cin >> a >> b;
cout << a + b << '\n';
```

Business logic: adding two integers

The part which deals with the software's purpose

```
int a;
int b;
cin >> a >> b;
cout << a + b << '\n';
```

Business logic: adding two integers

The part which deals with the software's purpose

```
FirstInteger a;
SecondInteger b;
input.obtainValues(a, b);
gui.display(a + b);
```

Business logic: adding two integers

The part which deals with the software's purpose

```
FirstInteger a;
SecondInteger b;
input.obtainValues(a, b);
gui.display(a + b);
```

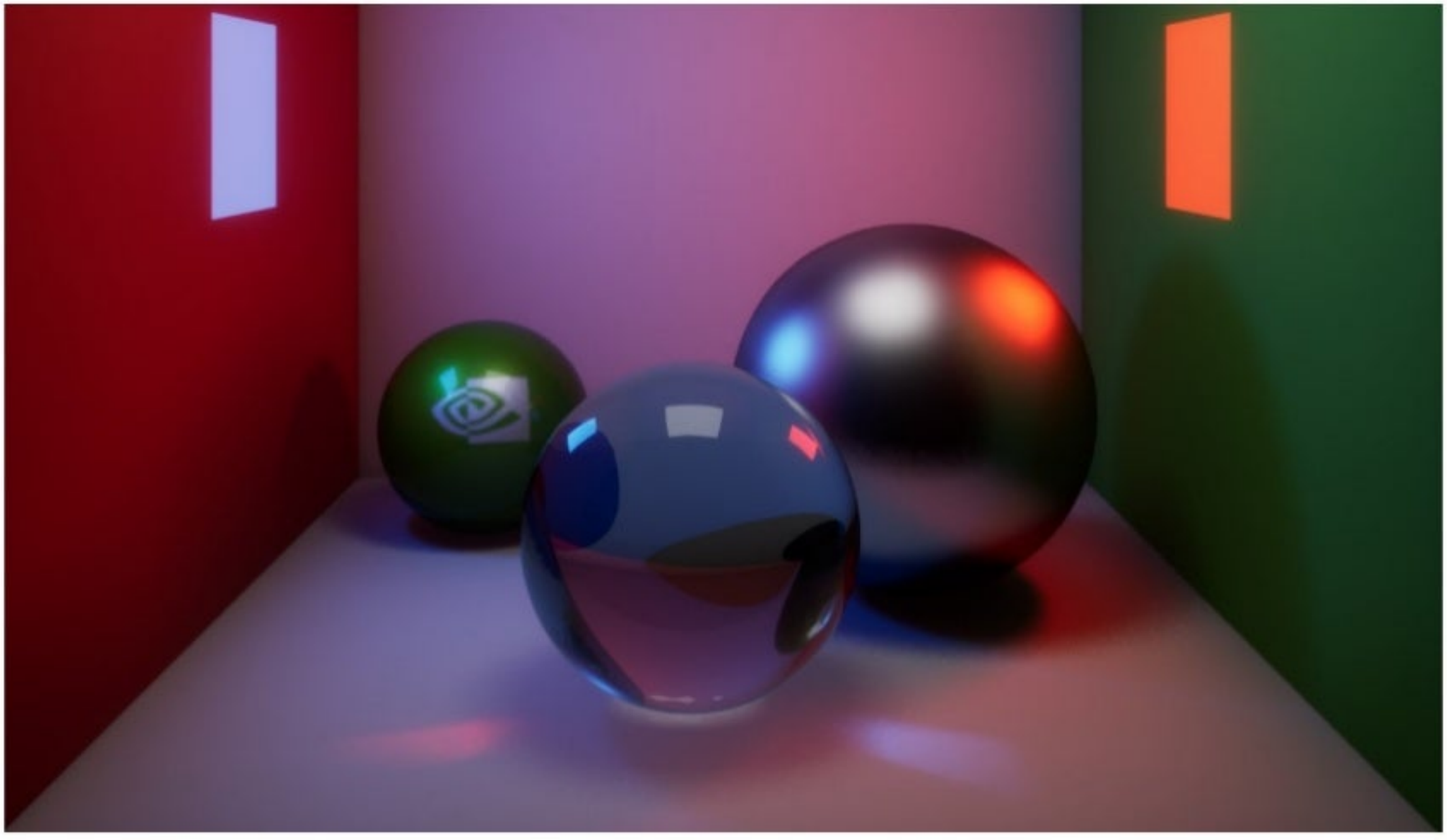
Business logic: adding two integers

Presentation, i/o...

```
FirstInteger a;
SecondInteger b;
input.obtainValues(a, b);
gui.display(a + b);
```

Non-business logic

Complex software doesn't mean complex business logic.



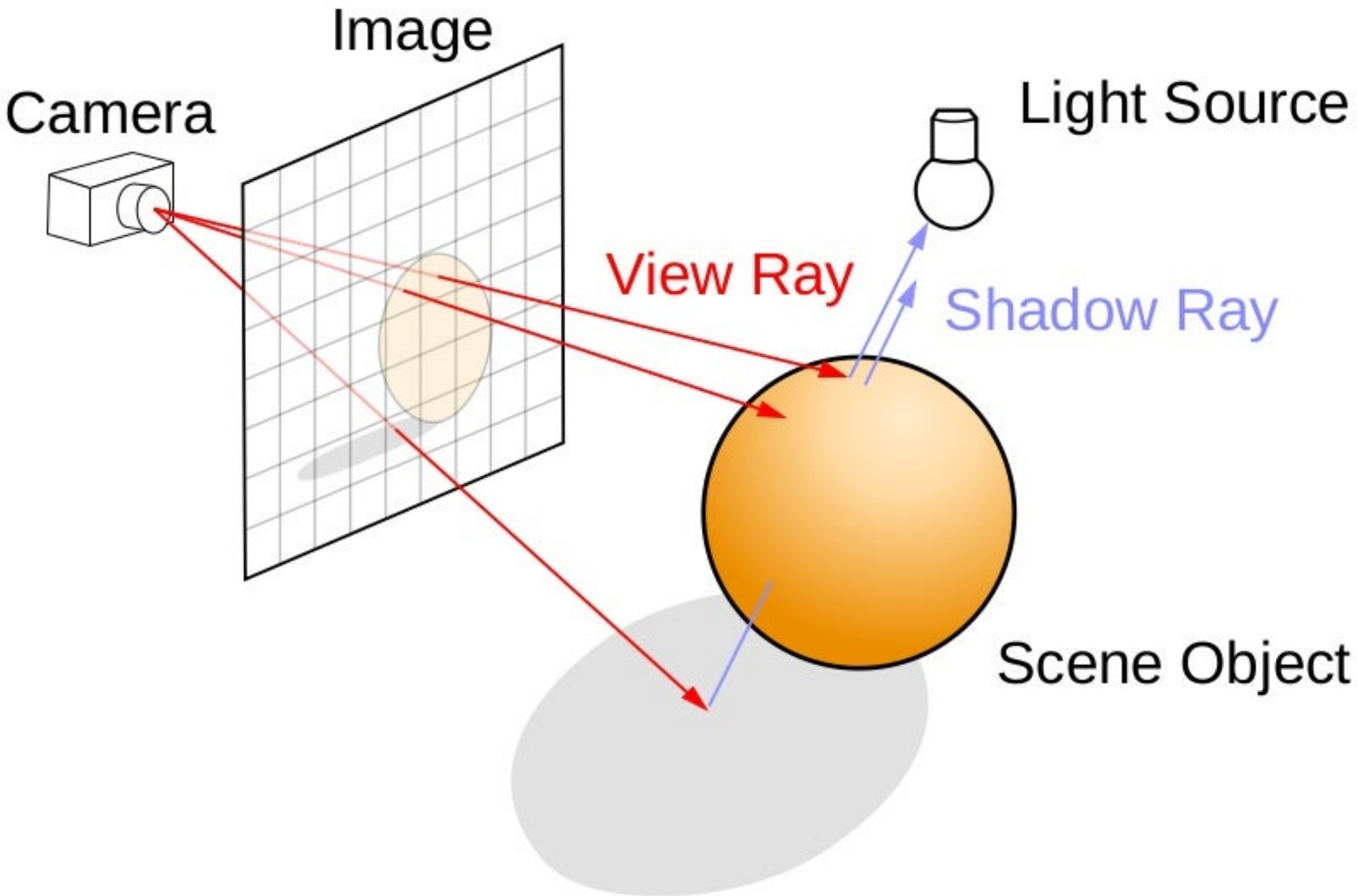
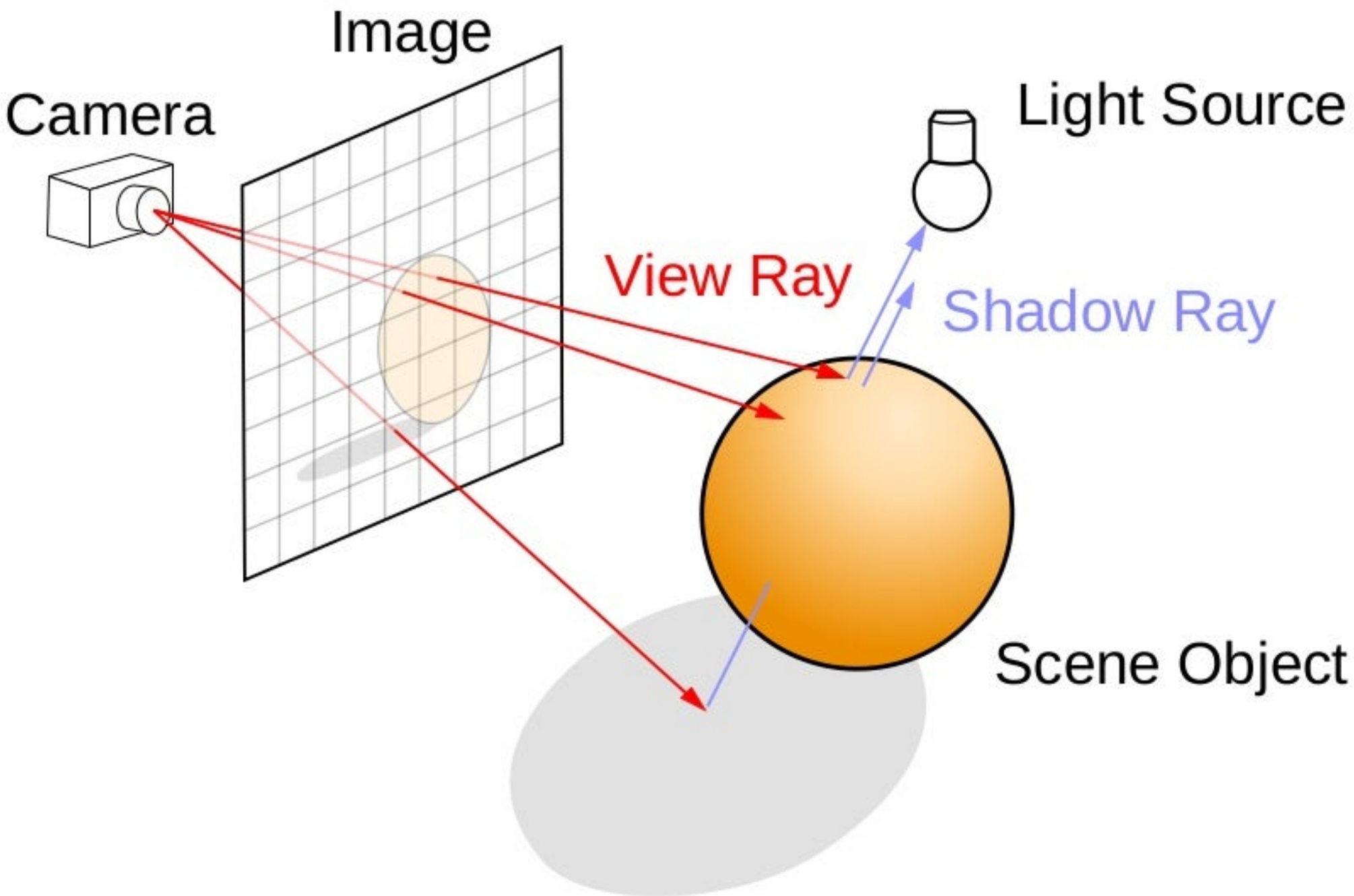


Image
Camera
Light Source
View Ray
Shadow Ray
Scene Object

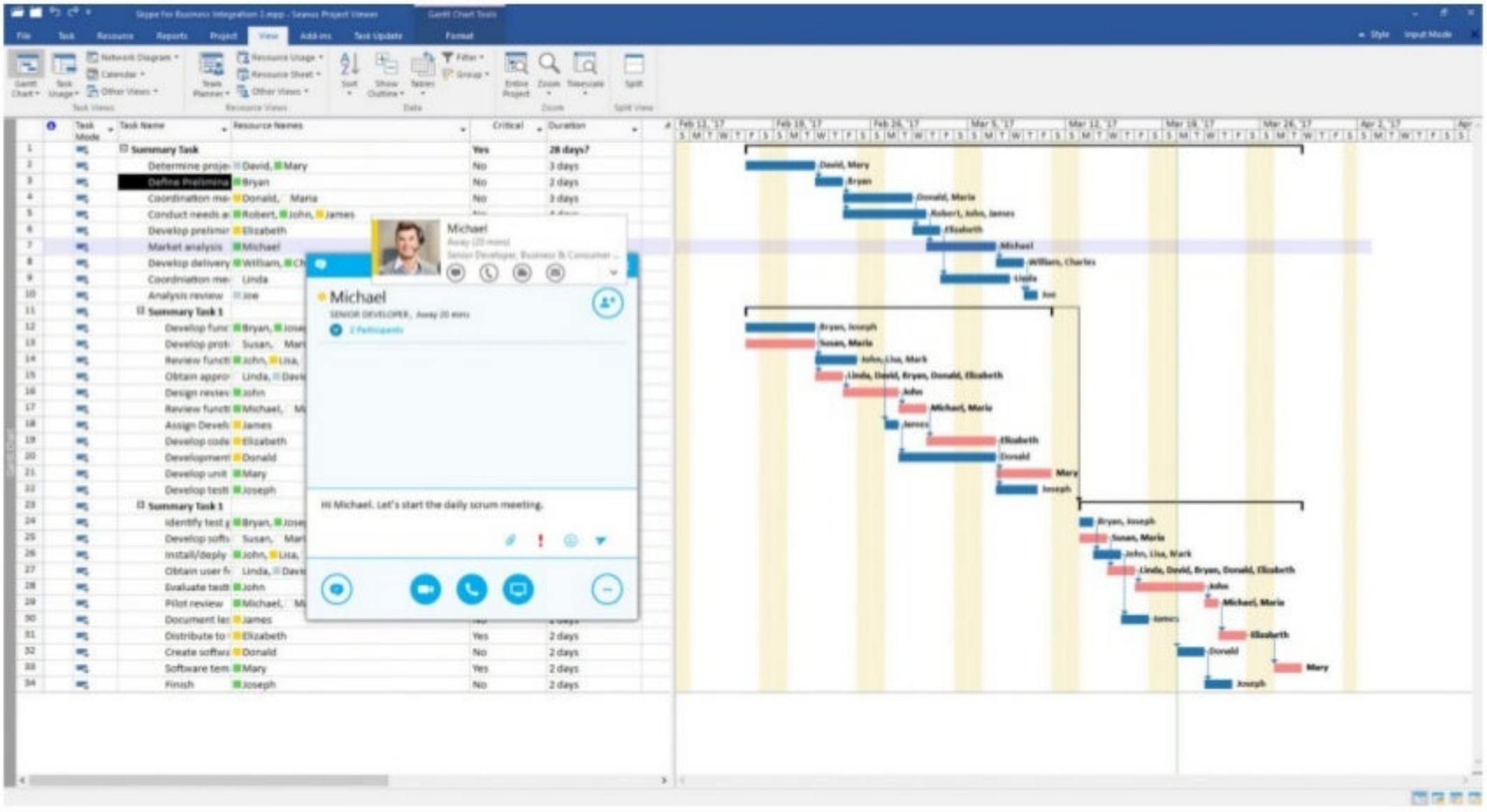
–



Simple. Right?


C++ User Group Sofia
Borislav
Home
Create
C++ User Group Sofia
Public Group
About
Discussion
Chats
Members
Events
Photos
Group Insights
Moderate Group
Search this group
Shortcuts
DevBG
Програмиране Бълг...
C++ User Group Sofia
View 7 more comments
Asen Alexandrov Книгите са хубаво нещо но от тях C++ ще научиш само повърхностно. Напиши си някакво малко приложение, например базов чат със стъи който поддържа множество клиенти, сървър, работа с база данни, базов UI на всеки клиент (примерно форми и бутони с QT), о... See More
Like · Reply · 2d
Ilian Zapryanov replied · 3 Replies
Stanimir Lukanov Отговора на такъв въпрос винаги е само един http://www.stroustrup.com/4th.html всичко друго е компромис...
STROUSTRUP.COM
Stroustrup: The C++ Programming Language (4th Edition)
Like · Reply · Remove Preview · 2d
Write a comment...
OLDER
Mitko Vasilev shared an event.
October 19 at 9:53 AM
Бързо напомняне за събитието другата седмица
24.10. - 19:30 ч.
Groups are separate spaces where you can share photos and make plans with just the people you want.
Create a Group
RECENT GROUP PHOTOS
See All
Suggested Groups
See All
Let's turn Ideas to Action Together!


CINEMA 4D Studio
Soubor Úpravy Objekty Nástroje Výběr Struktura Funkce Animace Simulace Rendering MoGraph Charakter Plugin Python Okno Nápověda
Úpravy Kamery Zobrazení Filtr zobrazení Pohled
Perspektiva
Objekty
Struktura
Soubor Úpravy Pohled Objekty Vlastnosti
Pozadí
Kamery
Kamera_predek_levo
Kamera_zadek_pravo
Kamera_zadek_levo
Kamera_predek
Kamera_predek_2
Kamera_levy_bok
Kamera_video
Kamera_video_bckp
světla
podstavec
osvetlena_vnitrni_plocha
BMW_M3_E90
Nastavení
Režim Úpravy Uživatelská data
CINEMA 4D dokument
Nastavení projektu
Info
Dynamika
Nastavení projektu
Měřítko dokumentu
Snímky za vteřinu
Čas dokumentu
Min. čas
Max. čas
Min. čas náhledu
Max. čas náhledu
Úroveň detailů
Úroveň detailů pro rendering v e
Použít animace
Použít chování
Použít generátory
Použít deformátory
Použít systém pohybu
Výchozí materiál
Barva
Materiály
Soubor Úpravy Funkce Textura
Souřadnice
Pozice
Velikost
Rotace
X 33.513 cm
Y 28.269 cm
Z -1222.446 cm
X 0 cm
Y 0 cm
Z 0 cm
H 16.4 °
P -10.3 °
B -8 °
Počátek
Velikost
Použít


Michael
SENIOR DEVELOPER, Away 20 mins
2 Participants
Hi Michael. Let's start the daily scrum meeting.

What do we have here?

- Websites - **JS, C#, Java, Ruby...**
- Gameplay - **lua, C#, Python...**
- CAD - **Python, C#...**
- Enterprise - **Everything but C++**

People don't seem to want to write business logic in C++

Note that there still is a lot of underlying C++ in such projects

Well... is there a problem with that?

Problems with that

- The code is **slower**
- There is **more complexity** in the binding layer
- There are **duplicated functionalities** (which means duplicated bugs)

Why don't people use C++?

Some reasons

- Buld times
- Hotswap
- Testing
- But (I think) most importantly...

OOP

... with dynamic polymorphism

Criticism of OOP

- You want a banana, but with it you also get the gorilla and the entire jungle
- It dangerously **couples the data with the functionality**
- It's **not reusable**
- It's **slow**

People forget that **C++ is an OOP language**

Defense of OOP

- Many cricisims target **concrete implementations**
- Many are about **misuse of OOP**
- Some are about **misconceptions of OOP**

But most importantly...

Software is written by human beings

Human beings think in terms of objects

Which languages thrive in fields with heavy business logic?

Almost all are object-oriented ones.

Powerful OOP

```
function f(shape) {
    // Everything that has a draw method works
    shape.draw();
}
Square = function () {
    this.draw = function() {
        console.log("Square");
    }
};
Circle = function () {
    this.draw = function() {
        console.log("Circle");
    }
};
f(new Square);
f(new Circle);
```

Vanilla C++ OOP

```
struct Shape {
    virtual void draw(ostream& out) const = 0;
}
void f(const Shape& s) {
    s.draw(cout);
}
struct Square : public Shape {
    virtual void draw(ostream& out) const override {
};
struct Circle : public Shape {
    virtual void draw(ostream& out) const override {
};
int main() {
    f(Square{});
    f(Circle{});
}
```

OOP isn't modern C++

However modern C++ is a pretty powerful language.

Polymorphic type-erasure wrappers

Boost.TypeErasure, Dyno, Folly.Poly, [Boost].TE

```
using Shape = Library_Magic(void, draw, (ostream&));
void f(const Shape& s) {
    s.draw(cout);
}
struct Square {
    void draw(ostream& out) const { out << "Square\n"; }
};
struct Circle {
    void draw(ostream& out) const { out << "Circle\n"; }
};
int main() {
    f(Square{});
    f(Circle{});
}
```

Polymorphic type-erasure wrappers

Boost.TypeErasure, Dyno, Folly.Poly, [Boost].TE

```
using Shape = Library_Magic(void, draw, (ostream&));
void f(const Shape& s) {
    s.draw(cout);
}
struct Square {
    void draw(ostream& out) const { out << "Square\n"; }
};
struct Circle {
    void draw(ostream& out) const { out << "Circle\n"; }
};
int main() {
    f(Square{});
    f(Circle{});
}
```

How does this work?

```
struct Shape {
    // virtual table
    std::function<void(ostream&)> draw;
    std::function<int()> area;

    // fill the virtual table when constructing
    template <typename T>
    Shape(const T& t) {
        draw = std::bind(&T::draw, &t);
        area = std::bind(&T::area, &t);
    }
};
```

How does this work?

```
struct Shape {
    // virtual table
    std::function<void(ostream&)> draw;
    std::function<int()> area;

    // fill the virtual table when constructing
    template <typename T>
    Shape(const T& t) {
        draw = std::bind(&T::draw, &t);
        area = std::bind(&T::area, &t);
    }
};
```

How does this work?

```
struct Shape {
    // virtual table
    std::function<void(ostream&)> draw;
    std::function<int()> area;

    // fill the virtual table when constructing
    template <typename T>
    Shape(const T& t) {
        draw = std::bind(&T::draw, &t);
        area = std::bind(&T::area, &t);
    }
};
```

This, of course, is a simple and naive implementation.
There's lots of room for optimization

Faster dispatch

```
struct Shape {
    // virtual table
    const void* m_obj;
    void (*m_draw)(const void*, ostream&);

    void draw() const {
        return m_draw(m_obj);
    }

    template <typename T>
    Shape(const T& t) {
        m_obj = &t;
        m_draw = [](const void* obj, ostream& out) {
            auto t = reinterpret_cast<const T*>(obj);
            t->draw(out);
        };
    }
};
```

Faster dispatch

```
struct Shape {
    // virtual table
    const void* m_obj;
    void (*m_draw)(const void*, ostream&);

    void draw() const {
        return m_draw(m_obj);
    }

    template <typename T>
    Shape(const T& t) {
        m_obj = &t;
        m_draw = [](const void* obj, ostream& out) {
            auto t = reinterpret_cast<const T*>(obj);
            t->draw(out);
        };
    }
};
```

Faster dispatch

```
struct Shape {
    // virtual table
    const void* m_obj;
    void (*m_draw)(const void*, ostream&);

    void draw() const {
        return m_draw(m_obj);
    }

    template <typename T>
    Shape(const T& t) {
        m_obj = &t;
        m_draw = [](const void* obj, ostream& out) {
            auto t = reinterpret_cast<const T*>(obj);
            t->draw(out);
        };
    }
};
```

Powerful OOP

```
Square.prototype.area = function() {
    return this.side * this.side;
}

Circle.prototype.area = function() {
    return this.radius * this.radius * Math.PI;
}
```

```
s = new Square;
s.side = 5;
console.log(s.area()); // 25
```

Vanilla C++ OOP

```
struct Shape {
    virtual void draw(ostream& out) const = 0;
}
struct Square : public Shape {
    virtual void draw(ostream& out) const override {
    int side;
};
int main()
{
    shared_ptr<Shape> ptr = make_shared<Square>();
    cout << ptr->area() << '\n'; // ??
}
```

Vanilla C++ OOP

```
struct Shape {
    virtual void draw(ostream& out) const = 0;
    virtual double area() const = 0;
}
struct Square : public Shape {
    virtual void draw(ostream& out) const override {
    virtual double area() const override { return si
    double side;
};
int main()
{
    shared_ptr<Shape> ptr = make_shared<Square>();
    cout << ptr->area() << '\n';
}
```

And what if we can't just edit the classes?

Open methods

Not to be confused with extension methods or UCS

[Boost].TE, yomm2

```
declare_method(double, area, (virtual_<const Shape&> s));

define_method(double, area, (const Square& s)
{
    return s.area * s.area;
}

int main() {
    shared_ptr<Shape> ptr = make_shared<Square>();
    cout << area(ptr) << '\n';
}
```

Open methods

Not to be confused with extension methods or UCS

[Boost].TE, yomm2

```
declare_method(double, area, (virtual_<const Shape&> s));

define_method(double, area, (const Square& s)
{
    return s.area * s.area;
}

int main() {
    shared_ptr<Shape> ptr = make_shared<Square>();
    cout << area(ptr) << '\n';
}
```

How does this work?

```
// declare method
using area_func = double (*)(const Shape*);
using area_func_getter = area_func (*)(const Shape*);
std::vector<area_func_getter> area_getters;
double area(Shape* s) {
    for(auto g : area_getters) {
        auto func = g();
        if(func) return func(s);
    }
    throw error("Object doesn't implement area");
}

// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
area_func area_for_square_getter(const Shape* s) {
    return dynamic_cast<const Square*>(s) ? area_for_squ
}
auto register_area_for_square = []() {
```

How does this work?

```
// declare method
using area_func = double (*)(const Shape*);
using area_func_getter = area_func (*)(const Shape*);
std::vector<area_func_getter> area_getters;
double area(Shape* s) {
    for(auto g : area_getters) {
        auto func = g();
        if(func) return func(s);
    }
    throw error("Object doesn't implement area");
}

// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
area_func area_for_square_getter(const Shape* s) {
    return dynamic_cast<const Square*>(s) ? area_for_squ
}
auto register_area_for_square = []() {
```

How does this work?

```
// declare method
using area_func = double (*)(const Shape*);
using area_func_getter = area_func (*)(const Shape*);
std::vector<area_func_getter> area_getters;
double area(Shape* s) {
    for(auto g : area_getters) {
        auto func = g();
        if(func) return func(s);
    }
    throw error("Object doesn't implement area");
}

// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
area_func area_for_square_getter(const Shape* s) {
    return dynamic_cast<const Square*>(s) ? area_for_squ
}
auto register_area_for_square = []() {
```

How does this work?

```
// declare method
using area_func = double (*)(const Shape*);
using area_func_getter = area_func (*)(const Shape*);
std::vector<area_func_getter> area_getters;
double area(Shape* s) {
    for(auto g : area_getters) {
        auto func = g();
        if(func) return func(s);
    }
    throw error("Object doesn't implement area");
}

// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
area_func area_for_square_getter(const Shape* s) {
    return dynamic_cast<const Square*>(s) ? area_for_squ
}
auto register_area_for_square = []() {
```

How does this work?

```
// declare method
using area_func = double (*)(const Shape*);
using area_func_getter = area_func (*)(const Shape*);
std::vector<area_func_getter> area_getters;
double area(Shape* s) {
    for(auto g : area_getters) {
        auto func = g();
        if(func) return func(s);
    }
    throw error("Object doesn't implement area");
}

// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
area_func area_for_square_getter(const Shape* s) {
    return dynamic_cast<const Square*>(s) ? area_for_squ
}
auto register_area_for_square = []() {
```

Optimize the dispatch

```
// declare method
using area_func = double (*)(const Shape*);
std::unordered_map<std::type_index, area_func> area_gett
double area(Shape* s) {
    auto f = area_getters.find(std::type_index(typeid(*s
    if (f == area_getters.end()) throw error("Object doe
    return f->second(s);
}
```

```
// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
auto register_area_for_square = []() {
    area_getters[std::type_index(typeid(Square))] = area
    return area_getters.size();
}();
double area_for_square(const Square&) // user defines af
```

Optimize the dispatch

```
// declare method
using area_func = double (*)(const Shape*);
std::unordered_map<std::type_index, area_func> area_gett
double area(Shape* s) {
    auto f = area_getters.find(std::type_index(typeid(*s
    if (f == area_getters.end()) throw error("Object doe
    return f->second(s);
}
```

```
// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
auto register_area_for_square = []() {
    area_getters[std::type_index(typeid(Square))] = area
    return area_getters.size();
}();
double area_for_square(const Square&) // user defines af
```

Optimize the dispatch

```
// declare method
using area_func = double (*)(const Shape*);
std::unordered_map<std::type_index, area_func> area_gett
double area(Shape* s) {
    auto f = area_getters.find(std::type_index(typeid(*s
    if (f == area_getters.end()) throw error("Object doe
    return f->second(s);
}
```

```
// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
auto register_area_for_square = []() {
    area_getters[std::type_index(typeid(Square))] = area
    return area_getters.size();
}();
double area_for_square(const Square&) // user defines af
```

Optimize the dispatch

```
// declare method
using area_func = double (*)(const Shape*);
std::unordered_map<std::type_index, area_func> area_gett
double area(Shape* s) {
    auto f = area_getters.find(std::type_index(typeid(*s
    if (f == area_getters.end()) throw error("Object doe
    return f->second(s);
}
```

```
// define method
double area_for_square(const Square&);
double area_for_square_call(const Shape* s) {
    auto square = static_cast<const Square*>(s);
    return area_for_square(*square);
}
auto register_area_for_square = []() {
    area_getters[std::type_index(typeid(Square))] = area
    return area_getters.size();
}();
double area_for_square(const Square&) // user defines af
```

Powerful OOP

```
multi sub collide(Square $s, Square $c) {
    collide_square_square($s, $c);
}
multi sub collide(Circle $s, Circle $c) {
    collide_circle_circle($s, $c);
}
multi sub collide(Square $s, Circle $c) {
    collide_square_circle($s, $c);
}
multi sub collide(Circle $c, Square $s) {
    collide_square_circle($s, $c);
}

my $s1 = get_random_shape;
my $s2 = get_random_shape;
say "Collision: " ~ collide($s1, $s2);
```

Vanilla C++ OOP

```
struct Shape {
    virtual bool collide(const Shape* other) const =
    virtual bool collideImpl(const Square*) const =
    virtual bool collideImpl(const Circle*) const =
    virtual bool collideImpl(const Triangle*) const
}
struct Square : public Shape {
    virtual bool collide(const Shape* other) const o
        return other->collideImpl(this);
    }
    virtual bool collideImpl(const Square*) const ov
    virtual bool collideImpl(const Circle*) const ov
    virtual bool collideImpl(const Triangle*) const
};
// ...
shared_ptr<Shape> s1 = get_random_shape();
shared_ptr<Shape> s2 = get_random_shape();
cout << "Collision: " << s1->collide(s2.get()) << '\
```

Vanilla C++ OOP

```
struct Shape {
    virtual bool collide(const Shape* other) const =
    virtual bool collideImpl(const Square*) const =
    virtual bool collideImpl(const Circle*) const =
    virtual bool collideImpl(const Triangle*) const
}
struct Square : public Shape {
    virtual bool collide(const Shape* other) const o
        return other->collideImpl(this); /* copy thi
    }
    virtual bool collideImpl(const Square*) const ov
    virtual bool collideImpl(const Circle*) const ov
    virtual bool collideImpl(const Triangle*) const
};
// ...
shared_ptr<Shape> s1 = get_random_shape();
shared_ptr<Shape> s2 = get_random_shape();
cout << "Collision: " << s1->collide(s2.get()) << '\
```

Vanilla C++ OOP

```
struct Shape {
    virtual bool collide(const Shape* other) const =
    virtual bool collideImpl(const Square*) const =
    virtual bool collideImpl(const Circle*) const =
    virtual bool collideImpl(const Triangle*) const
}
struct Square : public Shape {
    virtual bool collide(const Shape* other) const o
        return other->collideImpl(this);
    }
    virtual bool collideImpl(const Square*) const ov
    virtual bool collideImpl(const Circle*) const ov
    virtual bool collideImpl(const Triangle*) const
};
// ...
shared_ptr<Shape> s1 = get_random_shape();
shared_ptr<Shape> s2 = get_random_shape();
cout << "Collision: " << s1->collide(s2.get()) << '\
```

Multiple dispatch (Multimethods)

yomm2, Folly.Poly

```
declare_method(bool, collide,
    (virtual_<const Shape&> s1, (virtual_<const Shape&>

define_method(bool, collide, (const Square& s, const Cir
{
    return collide_square_circle(s, c);
}
define_method(bool, collide, (const Circle& c, const Squ
{
    return collide_square_circle(s, c);
}
// ...
shared_ptr<Shape> s1 = get_random_shape();
shared_ptr<Shape> s2 = get_random_shape();
cout << "Collision: " << collide(*s1, *s2) << '\n';
```

How does this work?

```
struct collide_index {
    size_t first;
    size_t second;
};


size_t hash(collide_index); // implement as you wish
```

```
collide_index circle_square = {
    std::type_index(typeid(Circle).hash_code(),
    std::type_index(typeid(Square).hash_code(),
};
```

Powerful OOP

```ruby
module FlyingCreature
  def move_to(target)
    puts can_move_to?(target) ?
      "flying to #{target}"
      : "can't fly to #{target}"
  end
  def can_move_to?(target)
    true # flying creatures don't care
  end
end
module AfraidOfEvens
  def can_move_to?(target)
    target % 2 != 0
  end
end
a = Object.new
a.extend(FlyingCreature)
a.move_to(10) # -> flying to 10
a.extend(AfraidOfEvens)
a.move_to(10) # -> can’t fly to 10
```

Powerful OOP

```ruby
module FlyingCreature
  def move_to(target)
    puts can_move_to?(target) ?
      "flying to #{target}"
      : "can't fly to #{target}"
  end
  def can_move_to?(target)
    true # flying creatures don't care
  end
end
module AfraidOfEvens
  def can_move_to?(target)
    target % 2 != 0
  end
end
a = Object.new
a.extend(FlyingCreature)
a.move_to(10) # -> flying to 10
a.extend(AfraidOfEvens)
a.move_to(10) # -> can’t fly to 10
```

Powerful OOP

```ruby
module FlyingCreature
  def move_to(target)
    puts can_move_to?(target) ?
      "flying to #{target}"
      : "can't fly to #{target}"
  end
  def can_move_to?(target)
    true # flying creatures don't care
  end
end
module AfraidOfEvens
  def can_move_to?(target)
    target % 2 != 0
  end
end
a = Object.new
a.extend(FlyingCreature)
a.move_to(10) # -> flying to 10
a.extend(AfraidOfEvens)
a.move_to(10) # -> can’t fly to 10
```

Powerful OOP

```ruby
module FlyingCreature
  def move_to(target)
    puts can_move_to?(target) ?
      "flying to #{target}"
      : "can't fly to #{target}"
  end
  def can_move_to?(target)
    true # flying creatures don't care
  end
end
module AfraidOfEvens
  def can_move_to?(target)
    target % 2 != 0
  end
end
a = Object.new
a.extend(FlyingCreature)
a.move_to(10) # -> flying to 10
a.extend(AfraidOfEvens)
a.move_to(10) # -> can’t fly to 10
```

Powerful OOP

```ruby
module FlyingCreature
  def move_to(target)
    puts can_move_to?(target) ?
      "flying to #{target}"
      : "can't fly to #{target}"
  end
  def can_move_to?(target)
    true # flying creatures don't care
  end
end
module AfraidOfEvens
  def can_move_to?(target)
    target % 2 != 0
  end
end
a = Object.new
a.extend(FlyingCreature)
a.move_to(10) # -> flying to 10
a.extend(AfraidOfEvens)
a.move_to(10) # -> can’t fly to 10
```

Powerful OOP

```ruby
module FlyingCreature
  def move_to(target)
    puts can_move_to?(target) ?
      "flying to #{target}"
      : "can't fly to #{target}"
  end
  def can_move_to?(target)
    true # flying creatures don't care
  end
end
module AfraidOfEvens
  def can_move_to?(target)
    target % 2 != 0
  end
end
a = Object.new
a.extend(FlyingCreature)
a.move_to(10) # -> flying to 10
a.extend(AfraidOfEvens)
a.move_to(10) # -> can’t fly to 10
```

Vanilla C++ OOP

```
// ???
```

Vanilla C++ OOP

```
struct Object
{
    shared_ptr<ICanMoveTo> m_canMoveTo;
    shared_ptr<IMoveTo> m_moveTo;
    // ... every possible method ever
    bool canMoveTo(int target) const {
        return m_canMoveTo->call(target);
    } // ... every possible method ever (again)

    void extend(shared_ptr<Creature> c) {
        c->setObject(this);
        m_canMoveTo = c;
        m_moveTo = c;
    }
    void extend(shared_ptr<ICanMoveTo> cmt) {
        cmt->setObject(this);
        m_canMoveTo = cmt;
    } // ... every possible extension ever
};
```

Vanilla C++ OOP

```
// ... components
// ... virtual inheritence
// ... A LOT OF CODE
Object o;
o.extend(make_shared<FlyingCreature>());
o.moveTo(10); // flying to 10
o.extend(make_shared<AfraidOfEvens>());
o.moveTo(10); // can't fly to 10
```

Dynamic Mixins

DynaMix

```
DYNAMIX_MESSAGE_1(void, moveTo, int, target);
DYNAMIX_CONST_MESSAGE_1(bool, canMoveTo, int, target);
struct FlyingCreature {
    void moveTo(int target) {
        cout << (::canMoveTo(dm_this, target) ?
            "flying to " : "can't fly to ")
            << target << '\n';
    }
    bool canMoveTo(int target) const {
        return true;
    }
}; DYNAMIX_DEFINE_MIXIN(FlyingCreature, moveTo_msg & canMoveTo_msg)
struct AfraidOfEvens {
    bool canMoveTo(int target) const {
        return target % 2 != 0;
    }
}; DYNAMIX_DEFINE_MIXIN(AfraidOfEvens, priority(1, canMoveTo_msg));
int main() {
    object o;
    mutate(o).add<FlyingCreature>();
    moveTo(o, 10); // flying to 10
    mutate(o).add<AfraidOfEvens>();
    moveTo(o, 10); // can't fly to 10
}
```

Dynamic Mixins

DynaMix

```
DYNAMIX_MESSAGE_1(void, moveTo, int, target);
DYNAMIX_CONST_MESSAGE_1(bool, canMoveTo, int, target);
struct FlyingCreature {
    void moveTo(int target) {
        cout << (::canMoveTo(dm_this, target) ?
            "flying to " : "can't fly to ")
            << target << '\n';
    }
    bool canMoveTo(int target) const {
        return true;
    }
}; DYNAMIX_DEFINE_MIXIN(FlyingCreature, moveTo_msg & canMoveTo_msg)
struct AfraidOfEvens {
    bool canMoveTo(int target) const {
        return target % 2 != 0;
    }
}; DYNAMIX_DEFINE_MIXIN(AfraidOfEvens, priority(1, canMoveTo_msg));
int main() {
    object o;
    mutate(o).add<FlyingCreature>();
    moveTo(o, 10); // flying to 10
    mutate(o).add<AfraidOfEvens>();
    moveTo(o, 10); // can't fly to 10
}
```

Dynamic Mixins

DynaMix

```
DYNAMIX_MESSAGE_1(void, moveTo, int, target);
DYNAMIX_CONST_MESSAGE_1(bool, canMoveTo, int, target);
struct FlyingCreature {
    void moveTo(int target) {
        cout << (::canMoveTo(dm_this, target) ?
            "flying to " : "can't fly to ")
            << target << '\n';
    }
    bool canMoveTo(int target) const {
        return true;
    }
}; DYNAMIX_DEFINE_MIXIN(FlyingCreature, moveTo_msg & canMoveTo_msg)
struct AfraidOfEvens {
    bool canMoveTo(int target) const {
        return target % 2 != 0;
    }
}; DYNAMIX_DEFINE_MIXIN(AfraidOfEvens, priority(1, canMoveTo_msg));
int main() {
    object o;
    mutate(o).add<FlyingCreature>();
    moveTo(o, 10); // flying to 10
    mutate(o).add<AfraidOfEvens>();
    moveTo(o, 10); // can't fly to 10
}
```

Dynamic Mixins

DynaMix

```
DYNAMIX_MESSAGE_1(void, moveTo, int, target);
DYNAMIX_CONST_MESSAGE_1(bool, canMoveTo, int, target);
struct FlyingCreature {
    void moveTo(int target) {
        cout << (::canMoveTo(dm_this, target) ?
            "flying to " : "can't fly to ")
            << target << '\n';
    }
    bool canMoveTo(int target) const {
        return true;
    }
}; DYNAMIX_DEFINE_MIXIN(FlyingCreature, moveTo_msg & canMoveTo_msg)
struct AfraidOfEvens {
    bool canMoveTo(int target) const {
        return target % 2 != 0;
    }
}; DYNAMIX_DEFINE_MIXIN(AfraidOfEvens, priority(1, canMoveTo_msg));
int main() {
    object o;
    mutate(o).add<FlyingCreature>();
    moveTo(o, 10); // flying to 10
    mutate(o).add<AfraidOfEvens>();
    moveTo(o, 10); // can't fly to 10
}
```

Dynamic Mixins

DynaMix

```
DYNAMIX_MESSAGE_1(void, moveTo, int, target);
DYNAMIX_CONST_MESSAGE_1(bool, canMoveTo, int, target);
struct FlyingCreature {
    void moveTo(int target) {
        cout << (::canMoveTo(dm_this, target) ?
            "flying to " : "can't fly to ")
            << target << '\n';
    }
    bool canMoveTo(int target) const {
        return true;
    }
}; DYNAMIX_DEFINE_MIXIN(FlyingCreature, moveTo_msg & canMoveTo_msg)
struct AfraidOfEvens {
    bool canMoveTo(int target) const {
        return target % 2 != 0;
    }
}; DYNAMIX_DEFINE_MIXIN(AfraidOfEvens, priority(1, canMoveTo_msg));
int main() {
    object o;
    mutate(o).add<FlyingCreature>();
    moveTo(o, 10); // flying to 10
    mutate(o).add<AfraidOfEvens>();
    moveTo(o, 10); // can't fly to 10
}
```

Dynamic Mixins

DynaMix

```
DYNAMIX_MESSAGE_1(void, moveTo, int, target);
DYNAMIX_CONST_MESSAGE_1(bool, canMoveTo, int, target);
struct FlyingCreature {
    void moveTo(int target) {
        cout << (::canMoveTo(dm_this, target) ?
            "flying to " : "can't fly to ")
            << target << '\n';
    }
    bool canMoveTo(int target) const {
        return true;
    }
}; DYNAMIX_DEFINE_MIXIN(FlyingCreature, moveTo_msg & canMoveTo_msg)
struct AfraidOfEvens {
    bool canMoveTo(int target) const {
        return target % 2 != 0;
    }
}; DYNAMIX_DEFINE_MIXIN(AfraidOfEvens, priority(1, canMoveTo_msg));
int main() {
    object o;
    mutate(o).add<FlyingCreature>();
    moveTo(o, 10); // flying to 10
    mutate(o).add<AfraidOfEvens>();
    moveTo(o, 10); // can't fly to 10
}
```

Eye candy time!

MixQuest: github.com/iboB/mixquest

How does this work?

```
// message_registry
struct message_info {}
std::vector<message_info*> msg_infos;
// register mixin
auto moveTo_id = []() {
    msg_infos.emplace_back();
    return msg_infos.size();
}();

template <typename Mixin>
void* moveTo_caller() {
    return [](void* mixin, int target) {
        auto m = reintepret_cast<Mixin*>(mixin);
        m->moveTo(target);
    };
}
void moveTo(object& o, int target);
```

How does this work?

```
// message_registry
struct message_info {}
std::vector<message_info*> msg_infos;
// register mixin
auto moveTo_id = []() {
    msg_infos.emplace_back();
    return msg_infos.size();
}();

template <typename Mixin>
void* moveTo_caller() {
    return [](void* mixin, int target) {
        auto m = reintepret_cast<Mixin*>(mixin);
        m->moveTo(target);
    };
}
void moveTo(object& o, int target);
```

How does this work?

```
// message_registry
struct message_info {}
std::vector<message_info*> msg_infos;
// register mixin
auto moveTo_id = []() {
    msg_infos.emplace_back();
    return msg_infos.size();
}();

template <typename Mixin>
void* moveTo_caller() {
    return [](void* mixin, int target) {
        auto m = reintepret_cast<Mixin*>(mixin);
        m->moveTo(target);
    };
}
void moveTo(object& o, int target);
```

How does this work?

```
// message_registry
struct message_info {}
std::vector<message_info*> msg_infos;
// register mixin
auto moveTo_id = []() {
    msg_infos.emplace_back();
    return msg_infos.size();
}();

template <typename Mixin>
void* moveTo_caller() {
    return [](void* mixin, int target) {
        auto m = reintepret_cast<Mixin*>(mixin);
        m->moveTo(target);
    };
}
void moveTo(object& o, int target);
```

How does this work?

```
// mixin_registry
struct mixin_info {
    std::vector<void*> funcs;
}
std::vector<mixin_info*> mixin_infos;

// register mixin
auto FlyingCreature_id = []() {
    auto info = new mixin_type_info;
    info->funcs.resize(registered_messages.size());
    info->funcs[moveTo_id] = moveTo_caller<FlyingCreatur
}();
mixin_type_info* info_for(FlyingCreature*) {
    return mixin_infos[FlyingCreature_id];
}
```

How does this work?

```
// mixin_registry
struct mixin_info {
    std::vector<void*> funcs;
}
std::vector<mixin_info*> mixin_infos;

// register mixin
auto FlyingCreature_id = []() {
    auto info = new mixin_type_info;
    info->funcs.resize(registered_messages.size());
    info->funcs[moveTo_id] = moveTo_caller<FlyingCreatur
}();
mixin_type_info* info_for(FlyingCreature*) {
    return mixin_infos[FlyingCreature_id];
}
```

How does this work?

```
// mixin_registry
struct mixin_info {
    std::vector<void*> funcs;
}
std::vector<mixin_info*> mixin_infos;

// register mixin
auto FlyingCreature_id = []() {
    auto info = new mixin_type_info;
    info->funcs.resize(registered_messages.size());
    info->funcs[moveTo_id] = moveTo_caller<FlyingCreatur
}();
mixin_type_info* info_for(FlyingCreature*) {
    return mixin_infos[FlyingCreature_id];
}
```

How does this work?

```
struct object {
    vector<std::unique_ptr<void>> mixins;

    struct vtable_entry {
        void* mixin;
        void* caller;
    };
    vector<vtable_entry> vtable;

    template <typename Mixin>
    void add() {
        mixins.emplace_back(make_unique<Mixin>());
        auto info = info_for((Mixin*)nullptr);
        for(size_t i=0; i<registered_messages.size(); ++
            if (info->funcs[i]) vtable[i] =
                {mixins.back().get(), info->funcs[i]};
        }
    }
};
```

How does this work?

```
struct object {
    vector<std::unique_ptr<void>> mixins;

    struct vtable_entry {
        void* mixin;
        void* caller;
    };
    vector<vtable_entry> vtable;

    template <typename Mixin>
    void add() {
        mixins.emplace_back(make_unique<Mixin>());
        auto info = info_for((Mixin*)nullptr);
        for(size_t i=0; i<registered_messages.size(); ++
            if (info->funcs[i]) vtable[i] =
                {mixins.back().get(), info->funcs[i]};
        }
    }
};
```

How does this work?

```
struct object {
    vector<std::unique_ptr<void>> mixins;

    struct vtable_entry {
        void* mixin;
        void* caller;
    };
    vector<vtable_entry> vtable;

    template <typename Mixin>
    void add() {
        mixins.emplace_back(make_unique<Mixin>());
        auto info = info_for((Mixin*)nullptr);
        for(size_t i=0; i<registered_messages.size(); ++
            if (info->funcs[i]) vtable[i] =
                {mixins.back().get(), info->funcs[i]};
        }
    }
};
```

How does this work?

```
struct object {
    vector<std::unique_ptr<void>> mixins;

    struct vtable_entry {
        void* mixin;
        void* caller;
    };
    vector<vtable_entry> vtable;

    template <typename Mixin>
    void add() {
        mixins.emplace_back(make_unique<Mixin>());
        auto info = info_for((Mixin*)nullptr);
        for(size_t i=0; i<registered_messages.size(); ++
            if (info->funcs[i]) vtable[i] =
                {mixins.back().get(), info->funcs[i]};
        }
    }
};
```

How does this work?

```
struct object {
    vector<std::unique_ptr<void>> mixins;

    struct vtable_entry {
        void* mixin;
        void* caller;
    };
    vector<vtable_entry> vtable;

    template <typename Mixin>
    void add() {
        mixins.emplace_back(make_unique<Mixin>());
        auto info = info_for((Mixin*)nullptr);
        for(size_t i=0; i<registered_messages.size(); ++
            if (info->funcs[i]) vtable[i] =
                {mixins.back().get(), info->funcs[i]};
        }
    }
};
```

How does this work?

```
void moveTo(object& o, int target) {
    auto& entry = o.vtable[moveTo_id];
    auto caller = reintepret_cast<void(*)(void*, int)>(e
    caller(entry.mixin, target);
}
```

The beast is back! - Jon Kalb

End

Questions?

Borislav Stanimirov / ibob.github.io / @stanimirovb

Link to these slides: http://ibob.github.io/slides/oop-in-cpp/