DynaMix

A New Take on Polymorphism

by Borislav Stanimirov / @stanimirovb

Hello, World

```
#include <iostream>

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

Hello, World

```
#include <iostream>

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

Bulgaria


Europe
International Boundary
Road
River
National Capital
City or Town
0 250 500 KM
0 250 500 Miles
© 2007 Geology.com
Norwegian Sea
North Sea
Atlantic Ocean
Baltic Sea
Black Sea
Mediterranean Sea
Caspian Sea
Bay of Biscay
RUSSIA
SWEDEN
FINLAND
NORWAY
U.K.
IRELAND
DENMARK
NETHERLANDS
BELGIUM
GERMANY
POLAND
FRANCE
SPAIN
PORTUGAL
ITALY
AUSTRIA
HUNGARY
ROMANIA
UKRAINE
BELARUS
BULGARIA
Sofia
GREECE
TURKEY
KAZAKHSTAN
IRAN
Moscow
Paris
London
Berlin
Rome
Madrid
Bucharest
Belgrade
Skopje
Athens
Istanbul
Ankara

Borislav Stanimirov

- Mostly a C++ programmer
- Mostly a game programmer
- Open-source programmer
- github.com/iboB

DynaMix: A New Take on Polymorphism

DynaMix: A New Take on Polymorphism

Polymorphism

Many forms

The same code does multiple things

Static Polymorphism

The differentiation is done by the compiler

```
abs(5);
abs(5.); // function overloads


template <typename Vector> // templates
auto dot_product(const Vector& a, const Vector& b) {
    assert(a.size() == b.size());
    typename Vector::value_type sum = 0;
    for (size_t i=0; i<a.size(); ++i) {
        sum += a[i]*b[i]; // operator overloads
    }
    return sum;
}
```

Static polymorphism is modern C++

image source

We're not going to talk about modern C++

Dynamic polymorphism and Object oriented programming (OOP)

Dynamic Polymorphism

- OOP has come to imply dynamic polymorphism
- Dynamic polymorphism is when the compiler can see a function call but can't know which actual piece of code will be executed next
  - Function pointers
  - `std::function`
  - Virtual functions
- It's in the category of things which are slower and can't have good compilation errors

Totally anti modern C++

State of OOP

- OOP has been criticized a lot
- People forget that C++ is an OOP language
- OOP can be useful for business logic (gameplay)

image source

C++ and OOP

Out of the box in an OOP context C++ only gives us virtual functions for polymorphism

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

C++ and Business Logic

- Is C++ is a bad choice for business logic?
- Many projects have chosen other languages: Lua, Python, JavaScript, Ruby...
  - C++ has poor OOP capabilities
  - You can hotswap
  - You can delegate to non-programmers
- However:
  - The code is slower
  - There is more complexity in the binding layer
  - There are duplicated functionalities (which means duplicated bugs)

DynaMix: A New Take on Polymorphism

DynaMix: A New Take on Polymorphism

Polymorphism in Modern C++

- Polymorphic type-erasure wrappers
  - Boost.TypeErasure, Dyno, Folly.Poly

```
using Drawable = Library_Magic(void, draw, (ostream&));
void f(const Drawable& d) {
    d.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{});
}
```

Polymorphism in Modern C++

- Polymorphic type-erasure wrappers
  - Boost.TypeErasure, Dyno, Folly.Poly

```
using Drawable = Library_Magic(void, draw, (ostream&));
void f(const Drawable& d) {
    d.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 Wrappers

- Better than classic virtual functions
  - Information hiding (PIMPL)
  - Non-intrusive
  - More extensible
  - Potentially faster

- … but more or less the same
  - Interface types
  - Implementation types
  - Basically improved virtual functions
  - Don’t seem compelling enough to ditch scripting languages

Other C++ polymorphism

- Signals/slots (Multicasts)
  - Very popular
  - Especially in GUI libraries (say Qt)
  - Boost.Signals2, FastDelegate, ...
- Multiple dispatch
  - `collide(obj1, obj2);`
  - Obscure feature
  - Relatively easy to mimic
  - Folly.Poly, yomm2
- Functional programming libraries

DynaMix: A New Take on Polymorphism

DynaMix: A New Take on Polymorphism

DynaMix

- Open source, MIT license, C++ library
  - github.com/iboB/dynamix

- This talk is an introduction to the library
  - Focus on the what and why
  - Hardly even mention the “how”
  - There will also be a small demo

- History
  - 2007: Interface. Zahary Karadjov
  - 2013: Rebirth as Boost.Mixin
  - 2016: Bye, Boost. Hello, DynaMix

Earthrise

Epic Pirate Story

Blitz Brigade: Rival Tactics

War Planet Online

Two more mobile games in development

What is DynaMix?

- Not a physics library
- Not even a game library
- A new take on polymorphism

Compose and modify polymorphic objects at run time

Some Inspirational Ruby

```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
```

Some Inspirational Ruby

```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
```

Some Inspirational Ruby

```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
```

Some Inspirational Ruby

```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
```

Some Inspirational Ruby

```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
```

Some Inspirational Ruby

```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
```

DynaMix means "Dynamic Mixins"

Static (CRTP) Mixins

```
struct cd_reader {
    string get_sound() const {
        return cd.empty() ? "silence" : ("cd: " + cd);
    }
    string cd;
};
template <typename Self>
struct headphones {
    const Self* self() const {
        return static_cast<const Self*>(this);
    }
    void play() {
        cout << "Playing " << self()->get_sound()
            << " through headphones\n";
    }
};
struct diskman : public cd_reader, public headphones<diskman>
struct boombox : public cd_reader, public speakers<boombox> {}
struct ipod : public mp3_reader, public headphones<ipod> {};
```

Static (CRTP) Mixins

```
struct cd_reader {
    string get_sound() const {
        return cd.empty() ? "silence" : ("cd: " + cd);
    }
    string cd;
};
template <typename Self>
struct headphones {
    const Self* self() const {
        return static_cast<const Self*>(this);
    }
    void play() {
        cout << "Playing " << self()->get_sound()
            << " through headphones\n";
    }
};
struct diskman : public cd_reader, public headphones<diskman>
struct boombox : public cd_reader, public speakers<boombox> {}
struct ipod : public mp3_reader, public headphones<ipod> {};
```

Static (CRTP) Mixins

```
struct cd_reader {
    string get_sound() const {
        return cd.empty() ? "silence" : ("cd: " + cd);
    }
    string cd;
};
template <typename Self>
struct headphones {
    const Self* self() const {
        return static_cast<const Self*>(this);
    }
    void play() {
        cout << "Playing " << self()->get_sound()
            << " through headphones\n";
    }
};
struct diskman : public cd_reader, public headphones<diskman>
struct boombox : public cd_reader, public speakers<boombox> {}
struct ipod : public mp3_reader, public headphones<ipod> {};
```

Static (CRTP) Mixins

```
struct cd_reader {
    string get_sound() const {
        return cd.empty() ? "silence" : ("cd: " + cd);
    }
    string cd;
};
template <typename Self>
struct headphones {
    const Self* self() const {
        return static_cast<const Self*>(this);
    }
    void play() {
        cout << "Playing " << self()->get_sound()
            << " through headphones\n";
    }
};
struct diskman : public cd_reader, public headphones<diskman>
struct boombox : public cd_reader, public speakers<boombox> {}
struct ipod : public mp3_reader, public headphones<ipod> {};
```

Static (CRTP) Mixins

```
struct cd_reader {
    string get_sound() const {
        return cd.empty() ? "silence" : ("cd: " + cd);
    }
    string cd;
};
template <typename Self>
struct headphones {
    const Self* self() const {
        return static_cast<const Self*>(this);
    }
    void play() {
        cout << "Playing " << self()->get_sound()
            << " through headphones\n";
    }
};
struct diskman : public cd_reader, public headphones<diskman>
struct boombox : public cd_reader, public speakers<boombox> {}
struct ipod : public mp3_reader, public headphones<ipod> {};
```

Static (CRTP) Mixins

```
struct cd_reader {
    string get_sound() const {
        return cd.empty() ? "silence" : ("cd: " + cd);
    }
    string cd;
};
template <typename Self>
struct headphones {
    const Self* self() const {
        return static_cast<const Self*>(this);
    }
    void play() {
        cout << "Playing " << self()->get_sound()
            << " through headphones\n";
    }
};
struct diskman : public cd_reader, public headphones<diskman> {};
struct boombox : public cd_reader, public speakers<boombox> {};
struct ipod : public mp3_reader, public headphones<ipod> {};
```

Static Polymorphism with Mixins

```
template <typename SoundPlayer>
void use_player(SoundPlayer& player) {
    player.play();
}

int main() {
    diskman dm;
    dm.cd = "Led Zeppelin IV (1971)";
    use_player(dm); // -> Playing "Led Zeppelin IV (1971)"

    ipod ip;
    ip.mp3 = "Led Zeppelin - Black Dog.mp3";
    use_player(ip); // -> Playing "Led Zeppelin - Black Dog.mp
}
```

Static Polymorphism with Mixins

```
template <typename SoundPlayer>
void use_player(SoundPlayer& player) {
    player.play();
}

int main() {
    diskman dm;
    dm.cd = "Led Zeppelin IV (1971)";
    use_player(dm); // -> Playing "Led Zeppelin IV (1971)"

    ipod ip;
    ip.mp3 = "Led Zeppelin - Black Dog.mp3";
    use_player(ip); // -> Playing "Led Zeppelin - Black Dog.mp
}
```

Static Polymorphism with Mixins

```
template <typename SoundPlayer>
void use_player(SoundPlayer& player) {
    player.play();
}

int main() {
    diskman dm;
    dm.cd = "Led Zeppelin IV (1971)";
    use_player(dm); // -> Playing "Led Zeppelin IV (1971)"

    ipod ip;
    ip.mp3 = "Led Zeppelin - Black Dog.mp3";
    use_player(ip); // -> Playing "Led Zeppelin - Black Dog.mp
}
```

Static Polymorphism with Mixins

```
template <typename SoundPlayer>
void use_player(SoundPlayer& player) {
    player.play();
}

int main() {
    diskman dm;
    dm.cd = "Led Zeppelin IV (1971)";
    use_player(dm); // -> Playing "Led Zeppelin IV (1971)"

    ipod ip;
    ip.mp3 = "Led Zeppelin - Black Dog.mp3";
    use_player(ip); // -> Playing "Led Zeppelin - Black Dog.mp
}
```

DynaMix: The Gist

- Building blocks
  - `dynamix::object` – just an empty object
  - Mixins – user classes which actually implement functionality
  - Messages – function-like pieces of interface, that an object might implement

- Usage
  - Mutation – the process of adding and removing mixins from objects
  - Calling messages – like calling methods, this is where the actual business logic lies

DynaMix Sound Player

```
dynamix::object sound_player; // just an empty dynamix::object

dynamix::mutate(sound_player)
    .add<cd_reader>()
    .add<headphones_output>();

sound_player.get<cd_reader>()->insert("Led Zeppelin IV (1971)"

// play is a message
play(sound_player); // cant have sound_player.play() :(
// -> Playing CD "Led Zeppelin IV (1971)" through headphones

dynamix::mutate(sound_player)
    .remove<headphones_output>()
    .add<speakers_output>();

play(sound_player);
// -> Playing CD "Led Zeppelin IV (1971)" THROUGH SPEAKERS
```

DynaMix Sound Player

```
dynamix::object sound_player; // just an empty dynamix::object

dynamix::mutate(sound_player)
    .add<cd_reader>()
    .add<headphones_output>();

sound_player.get<cd_reader>()->insert("Led Zeppelin IV (1971)"

// play is a message
play(sound_player); // cant have sound_player.play() :(
// -> Playing CD "Led Zeppelin IV (1971)" through headphones

dynamix::mutate(sound_player)
    .remove<headphones_output>()
    .add<speakers_output>();

play(sound_player);
// -> Playing CD "Led Zeppelin IV (1971)" THROUGH SPEAKERS
```

DynaMix Sound Player

```
dynamix::object sound_player; // just an empty dynamix::object

dynamix::mutate(sound_player)
    .add<cd_reader>()
    .add<headphones_output>();

sound_player.get<cd_reader>()->insert("Led Zeppelin IV (1971)"

// play is a message
play(sound_player); // cant have sound_player.play() :(
// -> Playing CD "Led Zeppelin IV (1971)" through headphones

dynamix::mutate(sound_player)
    .remove<headphones_output>()
    .add<speakers_output>();

play(sound_player);
// -> Playing CD "Led Zeppelin IV (1971)" THROUGH SPEAKERS
```

DynaMix Sound Player

```
dynamix::object sound_player; // just an empty dynamix::object

dynamix::mutate(sound_player)
    .add<cd_reader>()
    .add<headphones_output>();

sound_player.get<cd_reader>()->insert("Led Zeppelin IV (1971)"

// play is a message
play(sound_player); // cant have sound_player.play() :(
// -> Playing CD "Led Zeppelin IV (1971)" through headphones

dynamix::mutate(sound_player)
    .remove<headphones_output>()
    .add<speakers_output>();

play(sound_player);
// -> Playing CD "Led Zeppelin IV (1971)" THROUGH SPEAKERS
```

DynaMix Sound Player

```
dynamix::object sound_player; // just an empty dynamix::object

dynamix::mutate(sound_player)
    .add<cd_reader>()
    .add<headphones_output>();

sound_player.get<cd_reader>()->insert("Led Zeppelin IV (1971)"

// play is a message
play(sound_player); // cant have sound_player.play() :(
// -> Playing CD "Led Zeppelin IV (1971)" through headphones

dynamix::mutate(sound_player)
    .remove<headphones_output>()
    .add<speakers_output>();

play(sound_player);
// -> Playing CD "Led Zeppelin IV (1971)" THROUGH SPEAKERS
```

DynaMix Sound Player

```
dynamix::object sound_player; // just an empty dynamix::object

dynamix::mutate(sound_player)
    .add<cd_reader>()
    .add<headphones_output>();

sound_player.get<cd_reader>()->insert("Led Zeppelin IV (1971)"

// play is a message
play(sound_player); // cant have sound_player.play() :(
// -> Playing CD "Led Zeppelin IV (1971)" through headphones

dynamix::mutate(sound_player)
    .remove<headphones_output>()
    .add<speakers_output>();

play(sound_player);
// -> Playing CD "Led Zeppelin IV (1971)" THROUGH SPEAKERS
```

Inevitable Boilerplate

Messages:

```
 // Declare
DYNAMIX_MESSAGE_0(string, get_sound);
DYNAMIX_MESSAGE_0(void, play);
DYNAMIX_MESSAGE_2(int, foo, float, arg1, string, arg2);

 // Define
DYNAMIX_DEFINE_MESSAGE(get_sound);
DYNAMIX_DEFINE_MESSAGE(play);
DYNAMIX_DEFINE_MESSAGE(foo);
```

We fully separate the interface from the implementation.

Message vs Method

Late binding and Smalltalk

```
struct Foo {
    void bar();
};
Foo foo;

foo.bar(); // message
void Foo::bar() {} // method
```

Boilerplate Continued

Mixins:

```
DYNAMIX_DECLARE_MIXIN(cd_reader);
DYNAMIX_DECLARE_MIXIN(headphones_output);
// That's all we need to mutate

class cd_reader {
public:
    string get_sound() {
        return _cd.empty() ? "silence" : ("CD " + cd);
    }
    void insert(const string& cd) {
        _cd = cd;
    }
    string _cd;
};

DYNAMIX_DEFINE_MIXIN(cd_reader, get_sound_msg);
// ...
DYNAMIX_DEFINE_MIXIN(mp3_reader, get_sound_msg);
```

Referring to the Owning Object

```
class headphones_output {
public:
    void play() {
        cout << "Playing " << get_sound(dm_this)
            << " through headphones\n";
    }
};

DYNAMIX_DEFINE_MIXIN(headphones_output, play_msg);
```

Referring to the Owning Object

```
class headphones_output {
public:
    void play() {
        cout << "Playing " << get_sound(dm_this)
            << " through headphones\n";
    }
};

DYNAMIX_DEFINE_MIXIN(headphones_output, play_msg);
```

- `dm_this` is like `self`: the owning object
- No inheritance. The library is non-intrusive

Eye candy time!

MixQuest: github.com/iboB/mixquest

DynaMix vs Scripts

- Cons of using a scripting language
  - The code is slower V
  - More complexity in the binding layer V
  - There are duplicated functionalities (which means duplicated bugs) V
- Pros of using a scripting language
  - C++ has poor OOP capabilities V
  - You can hotswap V
  - You can delegate to non-programmers X
    - Not yet. But you can mix.

The library has found a niche in mobile games, which are less likely to sacrifice performance

When to Use DynaMix?

- When you're writing software with complex polymorphic objects
- When you have subsystems which care about interface (rather than data)
- When you want plugins which enable various aspects of your objects
- Such types of projects include
  - Most CAD systems
  - Some games: especially RPGs and strategies
  - Some enterprise systems

When Not to Use DynaMix?

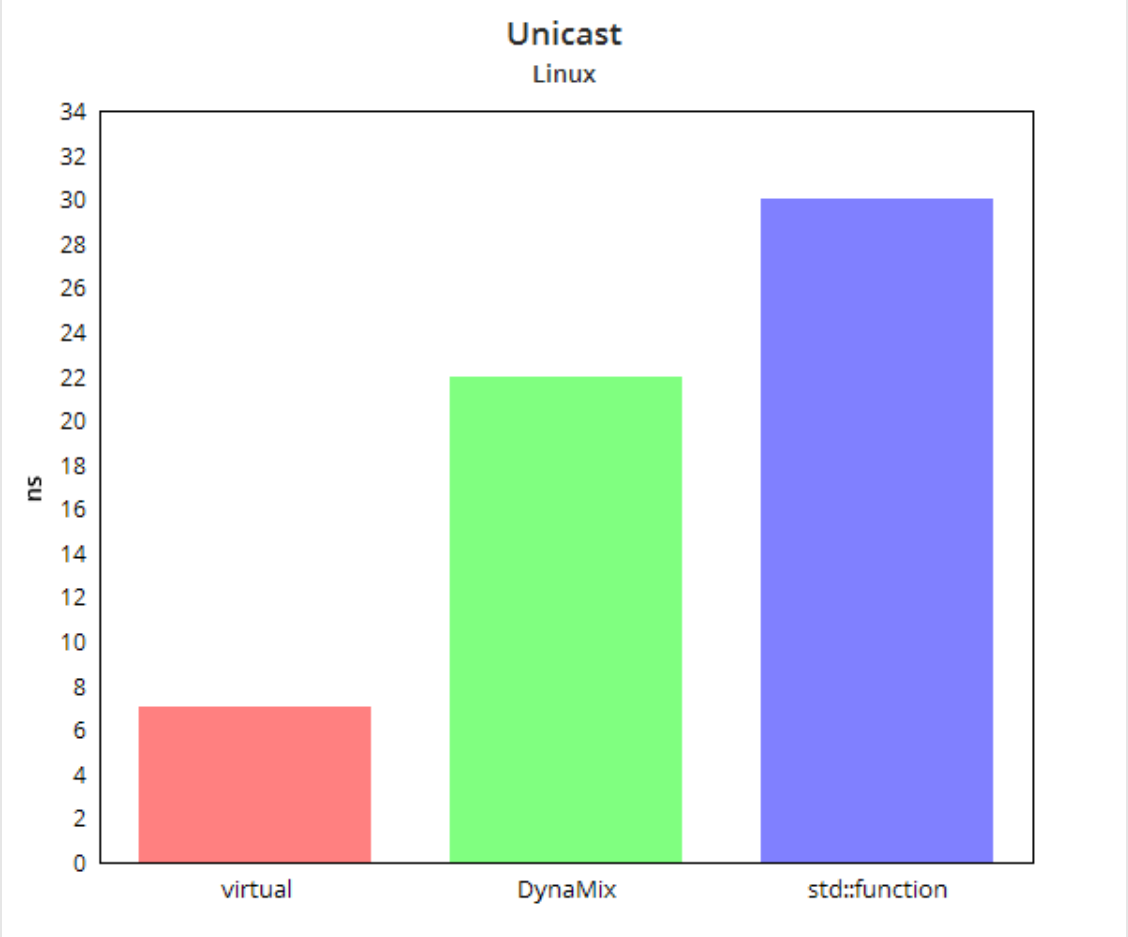
DynaMix is a means to create a project's architecture rather than achieve its purpose.

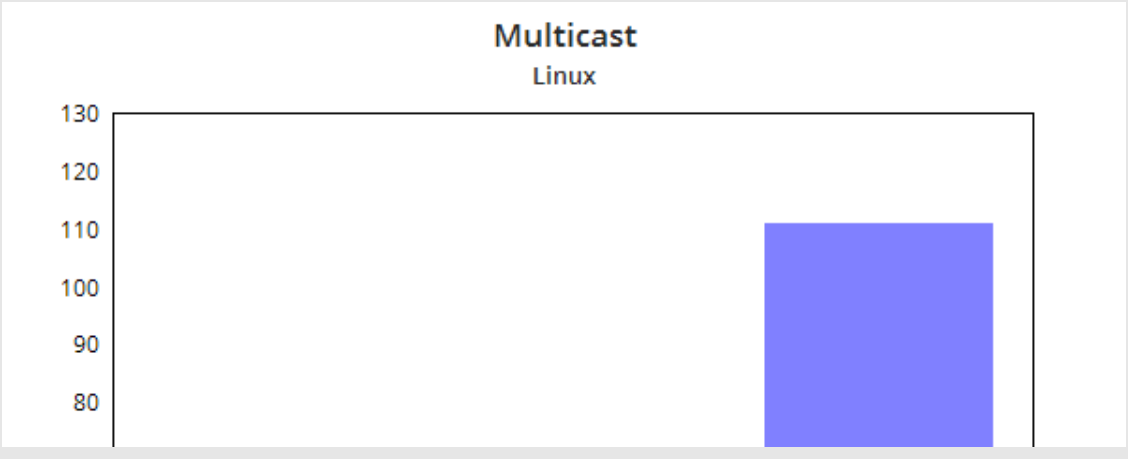
... so it's not really suitable for:

- Small scale projects
- Projects which have little use of polymorphism
- Existing large projects
- In performance critical code. As any other dynamic polymorphism.

Performance

- Message calls, as any polymorphic call, are slower than function calls
- They are comparable to `std::function`
- Mutations can be fairly slow. Internal types
- Memory overhead
  - For objects: pointers but mainly size of mixins
  - For unique types: sparse arrays of mixins
- Thread safety
  - Calling messages is safe
  - Mutating an object in one thread and calling messages on it in another is not safe
  - Mutating two objects in two threads is safe


Unicast
Linux
ns
34
32
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
0
virtual
DynaMix
std::function


Multicast
Linux
130
120
110
100
90
80

Recap

- Compose and mutate objects from mixins
- Have uni- and multicast messages
- Manage message execution with priorities
- Easily have hot-swappable or even releaseble plugins
- There was no time for:
  - Custom allocators
  - Message bids
  - Multicast result combinators
  - Implementation details

C++ allows you to have great power with OOP. Make use of it!

... but don't abuse it

End

Questions?

Borislav Stanimirov / ibob.github.io / @stanimirovb

DynaMix is here: github.com/ibob/dynamix/

Link to these slides: http://ibob.github.io/slides/dynamix-cppcon/