

Callicore atacamana manava

Callicore pygas cylindrus

Callicore hydaspes

Callicore toxus nainana

Callicore pygas eucala

Callicore lycia aegina

Callicore volderi cajetani

Callicore salina

Callicore toxus thana

Madrid, November 2017

Before the show begins: some news from the Boost front



Before the show begins: some news from the Boost front

- 145 libraries (1.65.1) and counting, 5 new last year
 - Fiber, QVM, Process, PolyCollection, Stacktrace
- Trends
 - C++11 or later increasingly required by new libs
 - Postmodern (meta)functional: Hana, Fit, Mp11
 - OS and communications: Beast, Fiber, Process, Stacktrace
- Building Boost
 - Boost.Build challenged by CMake
 - Bincrafters now provides Boost packages through **conan**
- You have no excuses for not using Boost!

Let 's dive into Boost.PolyCollection



Type hierarchy for a role playing game

```
struct sprite
{
    sprite(int id):id(id){}
    virtual ~sprite()=default;
    virtual void render(std::ostream& os) const=0;

    int id;
};
```

```
struct warrior:sprite
{
    using sprite::sprite;
    warrior(std::string rank,int id):
        sprite{id},rank{std::move(rank)}{}
    void render(std::ostream& os) const override
    {os<<rank<<" "<<id;}
    std::string rank="warrior";
};
```



```
struct juggernaut:warrior
{
    juggernaut(int id):
        warrior{"juggernaut",id}{}
};
```



```
struct goblin:sprite
{
    using sprite::sprite;
    void render(std::ostream& os)
    const override
    {os<<"goblin "<<id;}
};
```



Populate and render



vector.cpp

```
std::vector<std::unique_ptr<sprite>> c;

std::mt19937 gen{92748}; // some arbitrary random seed
std::discrete_distribution<> rnd{{1,1,1}};
for(int i=0;i<8;++i){ // assign each type with 1/3 probability
    switch(rnd(gen)){
        // watch out: std::make_unique requires C++14
        case 0: c.push_back(std::make_unique<warrior>(i));break;
        case 1: c.push_back(std::make_unique<juggernaut>(i));break;
        case 2: c.push_back(std::make_unique<goblin>(i));break;
    }
}

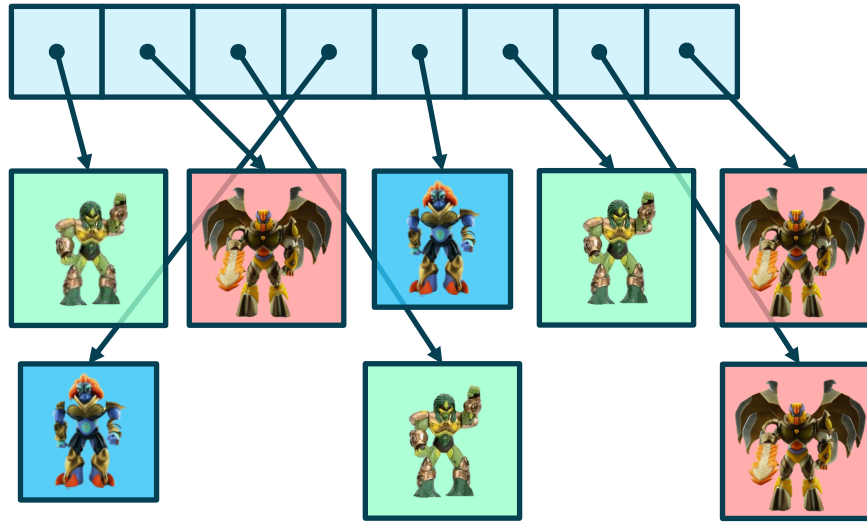
const char* comma="";
for(const auto& p:c){
    std::cout<<comma;
    p->render(std::cout);
    comma=",";
}
std::cout<<"\n";
```


Piece of cake



GORMITI is a trademark of Giochi Preziosi S.p.A.

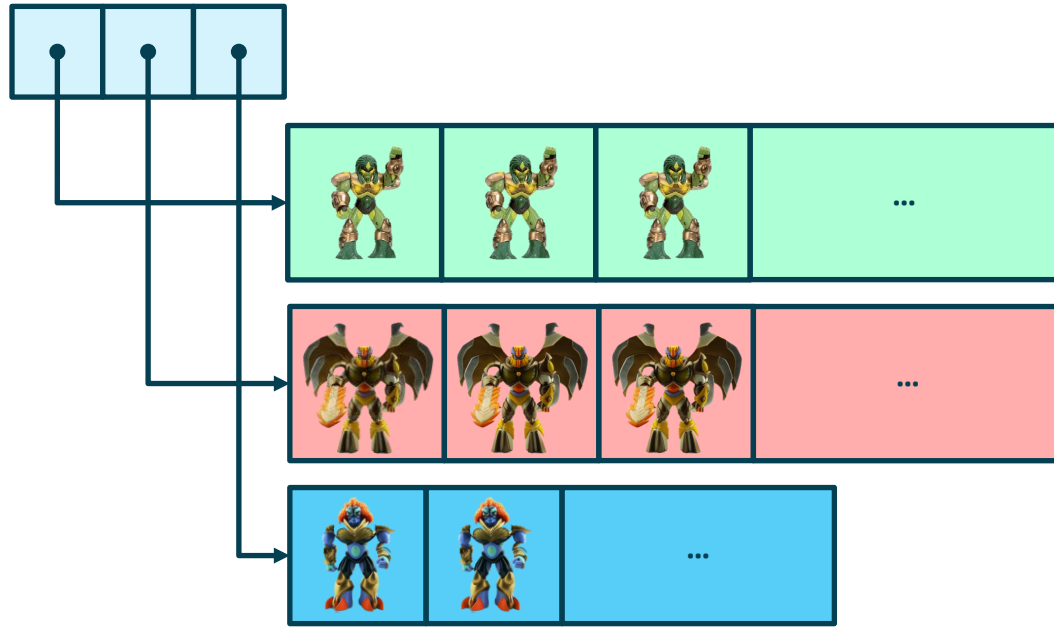
There are hidden inefficiencies, though



- Objects are scattered through memory → CPU cache misses
- Types are mixed up along traversal → CPU branch prediction fails
 - We could sort the vector on type*, object scattering is harder to fix though
- All boils down to OOP requiring pointer indirections

*See my “Mind the cache” talk at github.com/joaquintides/usingstdcpp2015

Packing things together



- `boost::base_collection` stores elements in per-type dedicated **segments**
- `[begin(), end())` traverses each segment in succession
 - So, free ordering of elements is lost (except within segment)
- In return, performance increases spectacularly



```
boost::base_collection<sprite> c;

std::mt19937 gen{92748}; // some arbitrary random seed
std::discrete_distribution<> rnd{{1,1,1}};
for(int i=0;i<8;++i){ // assign each type with 1/3 probability
    switch(rnd(gen)){
        case 0: c.insert(warrior{i});break;
        case 1: c.insert(juggernaut{i});break;
        case 2: c.insert(goblin{i});break;
    }
}

const char* comma="";
for(const sprite& s:c){
    std::cout<<comma;
    s.render(std::cout);
    comma=", ";
}
std::cout<<"\n";
```

- Spot the differences with the `std::vector` code

Interface at a glance



`base_collection_interface.cpp`

Segments can be targeted individually

Insertion interface mimics `std::multi_set` rather than `std::vector`

Emplacement requires that type of element be specified

New segments are created automatically; otherwise, use `register_types`



Segments can be targeted individually

```
std::sort( // sort warriors in descending id order
    c.begin<warrior>(), c.end<warrior>(),
    [](const warrior& x, const warrior& y){return x.id > y.id;});

for(const warrior& w : c.segment<warrior>()) // print warrior ids
    std::cout << w.id << "\n";

c.erase(c.begin(typeid(warrior))); // erase first warrior

c.clear<warrior>(); // erase all warriors
```

- Watch out: `begin(typeid(warrior))` is not the same as `begin<warrior>()`
- Two kinds of local iterators (prefer the latter)

Insertion interface mimics `std::multi_set` rather than `std::vector`

Emplacement requires that type of element be specified

New segments are created automatically; otherwise, use `register_types`

Interface at a glance



base_collection_interface.cpp

Segments can be targeted individually

Insertion interface mimics `std::multi_set` rather than `std::vector`

```
c.insert(juggernaut{8}); // at the end of juggernaut segment  
  
c.insert(c.begin(), juggernaut{9}); // hint only useful if  
                                   // juggernauts are the first segment  
c.insert(c.begin<juggernaut>(), juggernaut{10}); // 1st of juggernauts
```

Emplacement requires that type of element be specified

New segments are created automagically; otherwise, use `register_types`

Interface at a glance



base_collection_interface.cpp

Segments can be targeted individually

Insertion interface mimics `std::multi_set` rather than `std::vector`

Emplacement requires that type of element be specified

```
c.emplace<goblin>(11);
```

```
c.emplace_pos<goblin>(c.begin<goblin>(),12); // 1st of goblins
```

New segments are created automagically; otherwise, use `register_types`

Interface at a glance



base_collection_interface.cpp

Segments can be targeted individually

Insertion interface mimics `std::multi_set` rather than `std::vector`

Emplacement requires that type of element be specified

New segments are created automatically; otherwise, use `register_types`

```
// new assortment of sprites
std::array<std::unique_ptr<sprite>,3> a{{
    std::make_unique<elf>(13),
    std::make_unique<ghoul>(14),
    std::make_unique<amazon>(15)
}};

c.register_types<elf,ghoul,amazon>(); // otherwise exception is thrown below

for(const auto&p:a)c.insert(*p);
```



Segments can be targeted individually

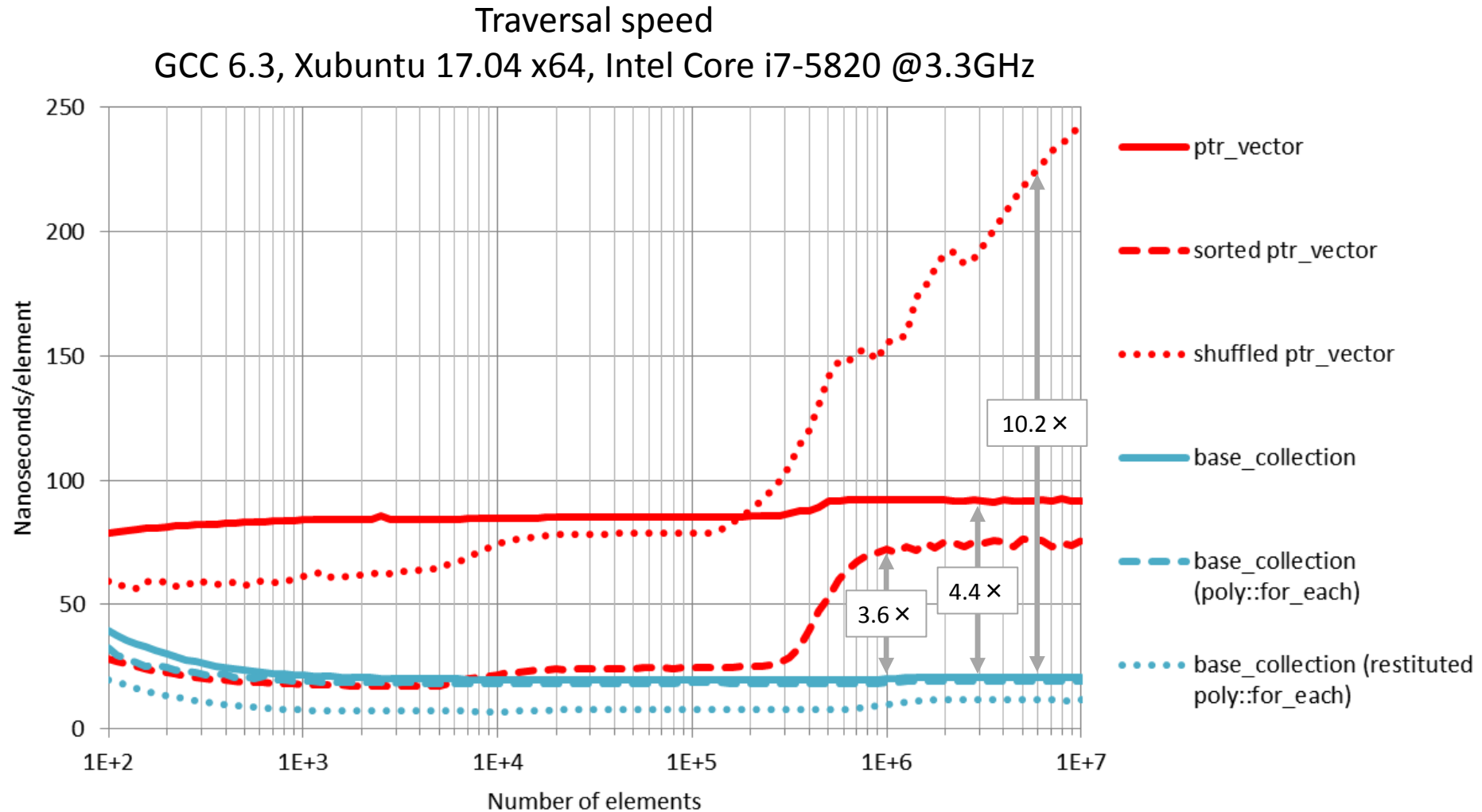
Insertion interface mimics `std::multi_set` rather than `std::vector`

Emplacement requires that type of element be specified

New segments are created automatically; otherwise, use `register_types`

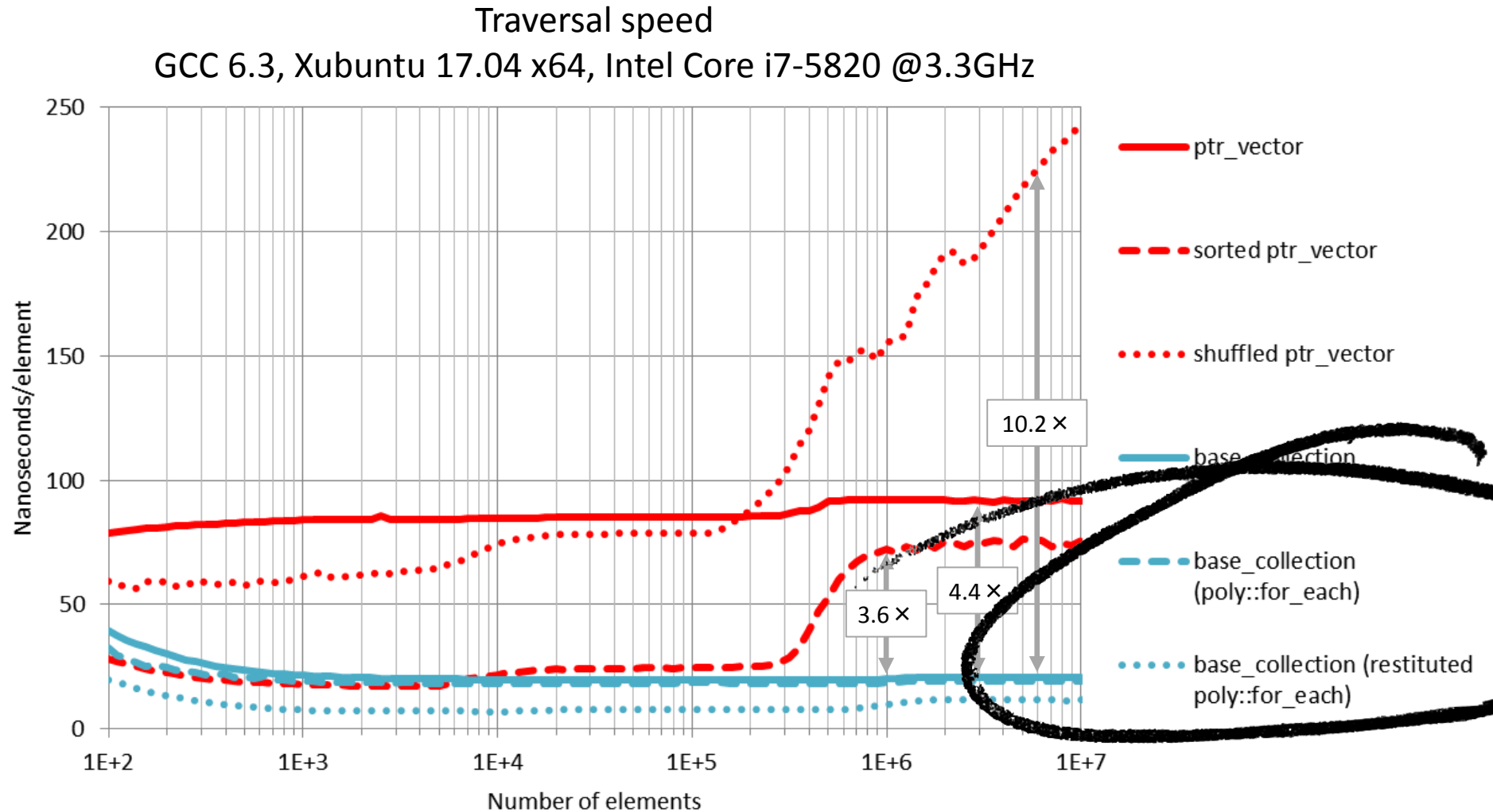
- All of this is well, but is it really faster?

Performance



■ Full results at
boost.org/doc/html/poly_collection/performance.html

Performance



■ Full results at
boost.org/doc/html/poly_collection/performance.html



■ Which is fastest?

```
for(const sprite& s:c){  
    std::cout<<comma;  
    s.render(std::cout);  
    comma=",";  
}
```

```
for(const auto& seg_info:c.segment_traversal()){  
    for(const sprite& s:seg_info){  
        std::cout<<comma;  
        s.render(std::cout);  
        comma=",";  
    }  
}
```

■ Double loop the basis of Boost.PolyCollection dedicated algorithms

```
boost::poly_collection::for_each(c.begin(),c.end(),[&](const sprite& s){  
    std::cout<<comma;  
    s.render(std::cout);  
    comma=",";  
});
```

■ 31 algorithms adapted from <algorithm>



```
boost::poly_collection::for_each<warrior,juggernaut,goblin>(
    c.begin(),c.end(),[&](const auto& s){
        std::cout<<comma;
        s.render(std::cout);
        comma=",";
    }
);
```

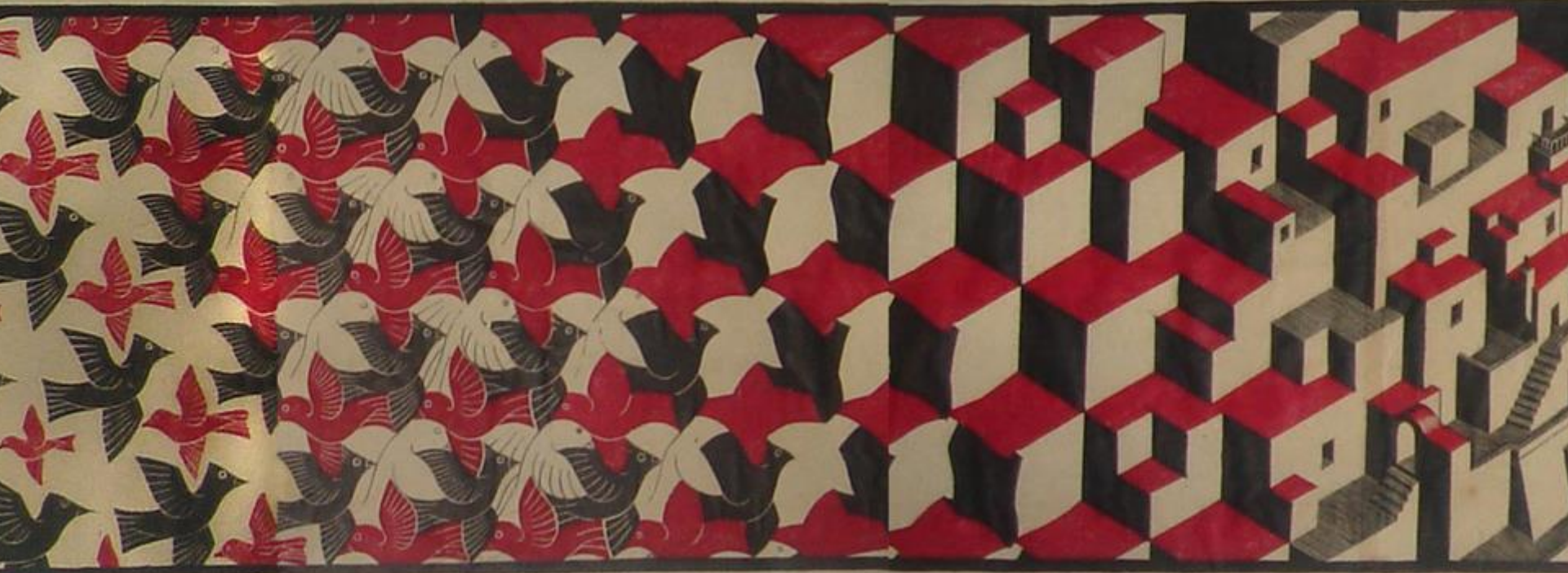
- Note the generic lambda

- **Type restitution** is the inverse of type abstraction

```
warrior* pw=new warrior{16};
sprite* ps=pw; // abstraction
warrior* pw2=static_cast<warrior*>(ps); // restitution
```

- Opportunities for **devirtualization** and inlining → more performance

A new look on dynamic polymorphism

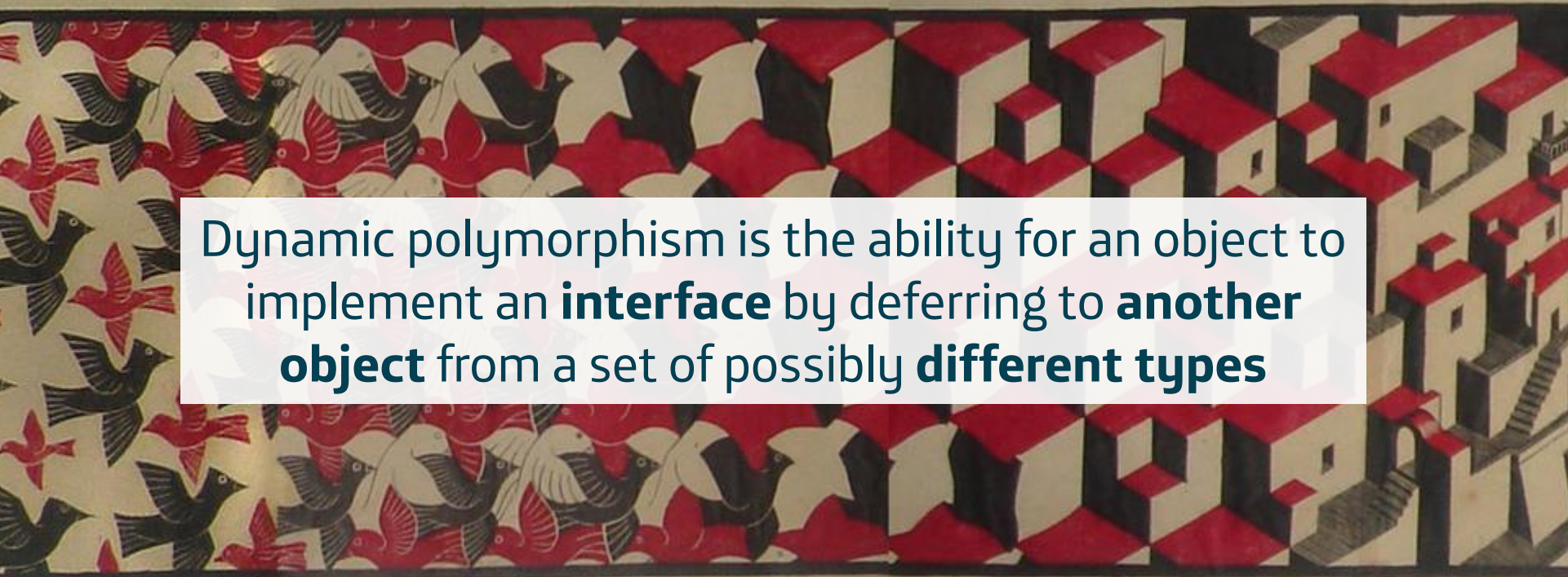


A new look on dynamic polymorphism

A horizontal strip of a M.C. Escher woodcut, likely from the 'Sky and Water' series. The image shows a transition from a pattern of birds on the left to a cityscape on the right. The birds are rendered in black, white, and red, with their forms gradually transforming into the geometric shapes of buildings. The buildings are also in black, white, and red, creating a seamless transition. The background is a light beige color.

What is dynamic polymorphism, anyway?

A new look on dynamic polymorphism



Dynamic polymorphism is the ability for an object to implement an **interface** by deferring to **another object** from a set of possibly **different types**

OOP is *just one model* of dynamic polymorphism...

```
sprite(int id);  
virtual ~sprite();
```

```
virtual void render(std::ostream& os) const;
```

interface

```
int id;
```

superobject

```
warrior(std::string rank, int id);
```

```
void render(std::ostream& os) const override;
```

```
std::string rank;
```

subobject

...and not a particularly flexible one at that

- Implementation and subtyping are coupled together by inheritance
- Implementation is intentional (again, through inheritance)
- Intersecting interfaces require forethought and virtual inheritance
 - `dynamic_cast` comes into play
 - Ever heard of the diamond problem?*
 - Believe me, you just don't want to enter this hell

*Nothing to do with choosing a wedding ring

Type erasure is the new cool...

- ...and `std::function` its most accessible example

```
int foo(int x){return x;}
struct bar{int operator()(int x)const{return 2*x;}};

using signature=int(int);
std::function<signature> f=&foo,g=bar{},h=[](int x){return 3*x;};

std::cout<<f(1)+g(2)+h(3)<<"\n";
```

- This is a new model of dynamic polymorphism
 - Interface: `signature`
 - Superobject: `std::function<signature>`
 - Subobjects: `int (*)(int)`, `bar`, compatible lambdas
- Interface compliance is checked at **compile time**



```
boost::function_collection<int(int)> c;

std::mt19937 gen{92748}; // some arbitrary random seed
std::discrete_distribution<> rnd{{1,1,1}};
for(int i=0;i<8;++i){ // assign each type with 1/3 probability
    switch(rnd(gen)){
        case 0: c.insert(&foo);break;
        case 1: c.insert(bar{});break;
        case 2: c.insert([](int x){return 3*x;});break;
    }
}

int res=0;
for(const auto& f:c)res+=f(1);
std::cout<<res<<"\n";
```

- Same interface as `boost::base_collection`
- How's this different from `std::vector<std::function<int(int)>>`?

Duck typing: type erasure on steroids

```
std::ostream& operator<<(std::ostream& os, const sprite& s)
{s.render(os); return os;}
```

```
using concept_=boost::mpl::vector<
    boost::type_erasure::ostreamable<>,
    boost::type_erasure::copy_constructible<>
>;
boost::type_erasure::any<concept_> a=5, b=std::string{"hello"}, c=warrior{16};

std::cout<<a<<"", "<<b<<"", "<<c<<"\n";
```

- Interface: `concept_`
- Superobject: `boost::type_erasure::any<concept_>`
- Subobjects: anything satisfying `concept_`
- Boost.TypeErasure not the only duck typing framework: see for instance Louis Dionne's **Dyno**
- Ultimate solution: native run-time C++ concepts (some day)



```
boost::any_collection<boost::type_erasure::ostreamable<>> c;

std::mt19937 gen{92748}; // some arbitrary random seed
std::discrete_distribution<> rnd{{1,1,1,1,1}};
for(int i=0;i<12;++i){ // assign each type with 1/5 probability
    switch(rnd(gen)){
        case 0: c.insert(warrior{i});break;
        case 1: c.insert(juggernaut{i});break;
        case 2: c.insert(goblin{i});break;
        case 3: c.insert(boost::format{"message %1%"}%i);break;
        case 4: c.insert(i);break;
    }
}

const char* comma="";
for(const auto& x:c){
    std::cout<<comma<<x;
    comma=", ";
}
std::cout<<"\n";
```

- Same interface as before... you get the idea

So many butterflies, so little time



So many butterflies, so little time

- Boost.PolyCollection packs elements of the same type together for maximum performance
 - Classic OOP `boost::base_collection`
 - Callable entities `boost::function_collection`
 - Duck typing `boost::any_collection`
- Remember: order of elements no longer free
- Squeeze more speed: dedicated algorithms and type restitution
- Dynamic polymorphism goes well beyond OOP
 - Read, play, use
- Drop me a note when you use Boost.PolyCollection

An intro to Boost.PolyCollection

Thank you

boost.org/libs/poly_collection
github.com/joaquintides/usingstdcpp2017

`using std::cpp 2017`

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>

Madrid, November 2017