Blitz Brigade: Rival Tactics

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)
    return target%2 != 0
  end
end
a = Object.new
a.extend(FlyingCreature)
a.move_to(10)
a.extend(AfraidOfEvens)
a.move_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 Polymorphism with Mixins


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

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

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

DynaMix: The Gist

- Building blocks
  - `dynamix::object` – just an empty object
  - `Mixins` – classes that you've written which actually implement messages
  - `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
```

Inevitable Boilerplate

Messages:

```
// In some header:
 DYNAMIX_MESSAGE_0(string, get_sound);
 DYNAMIX_MESSAGE_0(void, play);
 DYNAMIX_MESSAGE_2(int, foo, float, f, string, s);
```

```
// In some compilation unit (.cpp):
 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

Boilerplate Continued


```
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);
```

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

DEMO TIME

When to Use DynaMix

- When you're writing software with complex polymorphic objects
- When you have subsystems which care about interface (rather than data. Otherwise use an ECS)
- 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
- Small scale projects
- Projects which have little use of polymorphism
- Existing large projects
- In performance critical code

Recap

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

Thank you!

Questions?

DynaMix is here: github.com/iboB/dynamix

Interface to Component

Object:

```
class object {
    reader* _reader = nullptr;
    player* _player = nullptr;
    // ...
};
 // compose
 object sound_player;
 auto r = new cd_reader;
 r->cd = "Led Zeppelin IV (1971)";
 sound_player.set_reader(r);
 sound_player.set_player(new headphones);
 // modify
 sound_player.set_player(new speakers);
 // use
 sound_player.get_player()->play();
```

Interface to Component cont…

Component:

```
class component {
    object* self;
};

class player : public component {
    virtual void play() = 0;
};

struct headphones : public player {
    virtual void play() override {
        cout << "Playing " << self->get_reader()->get_sound()
            << " through headphones\n";
    }
};
```

Interface to Component cont…

- A pretty decent solution
- Not to be confused with entity-component-system
- Many games and CAD systems use it
- You can recreate almost every feature of DynaMix. But:
- In a concrete and nonreusable way
- Every new type of interface needs to be explicitly added to the huge object class
- Interfaces are limiting
- No new interfaces in plugins

2017 CPP-Summit

In C++17, templates just got a whole lot easier (and a little bit harder)

Michael Spertus

Fellow/VP, Symantec
University of Chicago

Templates are the heart of C++

- Templates have always been one of the defining features of C++
- They offer power unmatched by any language to avoid writing boilerplate, making compile-time decisions, implementing metaprogrammed design patterns, etc.
  - Look at *Modern C++ Design* by 2016 keynoter Alex Alexandrescu for an idea of just how powerful this can be (Note that you will want to adapt the code for more modern language versions)
- Unfortunately, they are a big part of why C++ is regarded as complicated to use
- A major strain in C++ design has been “How do we make templates easier to use”?
- In C++20, we are expecting Concepts, which is a new framework that will radically simplify templates
- But there are still some goodies in C++17 to simplify templates and help prepare your code for concepts in C++20
- Let’s get started

My passion: Constructor Template Argument Deduction

- "CTAD? Sounds complicated!
  What does it mean in simple terms?"

Constructor Template Argument Deduction (CTAD) is simply a fancy name for saying that C++17 will deduce class template arguments for constructors similarly to what C++ has always done with function templates. This can really simplify creating object from class templates, such as `vector`:

Before C++17

```
vector<int> v = {1, 2, 3}; // Why do I need to say this is a vector of ints?
lock_guard<shared_mutex> lck{smtx}; // Doesn't the compiler know the smtx is a shared_mutex?
shared_lock<shared_mutex> lck2{smtx2}; // Same here
scoped_lock<shared_mutex, shared_lock<shared_mutex>> assign_lock{smtx, lck2}; // Yuck!
```

By contrast, C++17 uses Constructor Template Argument Deduction to deduce the class template arguments of new objects without having to specify them explicitly:

C++17

```
vector v = {1, 2, 3}; // Deduces vector<int>, etc.
lock_guard lck{smtx};
shared_lock lck2{smtx2};
scoped_lock assign_lock{smtx, lck2};
```

Wow! That looks useful and cool! How does it work?

In its simplest form, CTAD just treats every constructor in the primary class template (i.e., the main class template, specializations are not considered) as if it were a template function and then applies ordinary deduction:

```
template<typename T> struct A { A(T); };
A a(7);  // Deduces A<int>
```

Can it really be that simple? The original version [2] of the proposal back in 2007(!) proposed simply that. Unfortunately, six more versions of the proposal were needed to nail down the language specification!

One problem is that common cases lead to "non-deducible" template contexts. To see what can go wrong, suppose you want to initialize a vector from two iterators:

```
// b and e are iterators
vector v2(b, e); // How does the compiler know it is supposed to deduce the iterator's value_type?
```

After looking at examples like these (see the Best Practices for more examples), we realized that there needed to be some way for the programmer to guide template argument deduction when the default behavior was insufficient. Fortunately, C++17 provides a *deduction guide* facility to create custom deduction rules:

```
// Inside <vector>
template<typename Iter> // Explain how to deduce from iterator pair!
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>;
// Outside <vector>
vector v2(b, e); // Works now!
```

Lessons from the standard library

- Whenever we do a new C++ language feature, we should check how it affects the library
- CTAD creates many opportunities for the standard library, so we spent a lot of time integrating it with all of the standard library classes
- This was helpful for the users of the standard library
- But it was at least as useful to us!
  - In the course of doing this, we found some flaws in the language feature that we were able to fix in C++17
  - By getting a lot of hands-on experience with using CTAD with a complex library, we were able to develop a set of best practices for using Constructor Template Argument Deduction that I can share with you today
  - They will also make their way to the C++ best practices repository that Kate Gregory will tell you about in her talk

First, a word about the standard process

- The German leader Otto von Bismarck once said about sausages and laws, that it's "best not to know how they're made"!!
- Well, that applies to standards processes too
  - So I will not be insulted if you cover your eyes on this slide
- CTAD went through at least 14 papers over 10 years before it got into the language!!
- That's a whole lot of sausage making!
- While it might have been nice to get out earlier, the benefit to you is that it is much more mature, comprehensive, and better understood feature that is deeply integrated into the standard library than any of the earlier versions
- In fact, we had to rush to get things done even in the final C++17 meeting, where I made presentations to all 4 major working groups (Evolution, Core, Library, and Library Evolution)
- So maybe there is something to say for the process ☺

Best Practices

- In this presentation, I will share some of the Best Practices we learned to make the most of this feature
- The good news is that if you are just using someone else' class templates, it should pretty much just work for you
- However, if you are designing your own classes, they may work out of the box, but sometimes you can make things easier for the user by thinking about CTAD when designing your constructors and by customizing them with deduction guides.
- First there will be a set of Basic Best Practices that class template authors should learn
- Then there will be some more Advanced Best Practices for the interested

Basic Best Practice 1: When to use CTAD

Let's start at the beginning with the Best Practice for when Constructor Template Argument Deduction should be used in the first place. In C++14, constructor template argument deduction was often simulated via "factory functions" such as:

```
auto tup = make_tuple(1, 2.3); // Deduces tuple<int, double>
auto bi = back_inserter(v); // Deduces back_insert_iterator<decltype(v)>
```

Now that we have CTAD, should we get rid of such factory functions? It is certainly tempting, as they are inconsistently-named ugly boilerplate. Just constructing the class seems so much easier to write and clearer to use. Indeed, this is usually exactly what you should do. Factory functions like `back_inserter` and `make_boyer_moore_searcher` don't really serve any purpose in C++17, and `make_boyer_moore_searcher` was removed from the standard library when the string search utilities were incorporated into C++17 from the Library Fundamentals TS.

However, there are cases where you should consider sticking with a factory function. Consider `make_tuple`, which does more than just call the tuple constructor, it also unwraps `reference_wrapper` arguments into references. While it is easy enough to write a deduction guide to emulate this behavior, this could silently lead to unexpected behavior since CTAD looks exactly like a constructor call. Therefore, our Best Practice is to only use CTAD for straightforward deduction of template arguments and use factory functions to call out special behavior: `make_tuple` should be retained as a factory function.

Basic Best Practice 2: Copying vs Wrapping

For classes that can be constructed from an initializer list, it can be confusing whether a braced initializer with a single element means copying with the copy constructor or wrapping with the initializer list constructor:

```
vector v{1, 2, 3, 4}; // Deduces vector<int>
vector v2{v, v, v}; // Deduces vector<vector<int>>
vector v3{v}; // What is v3? vector<int>? vector<vector<int>>
```

Indeed, this is not so obvious, but after extensive committee discussion [6], consensus was reached that copying should always take preference over wrapping when both are possible, so `v3` will be a `vector<int>`.

While this is usually what you want, you can still force list initialization (e.g. in generic code) by invoking your constructor in a way that could not possibly be a copy, such as:

```
vector v4({v}); // Make clear you want vector<vector<int>>
```

Basic Best Practice 3: Parameter Packs

A funny thing happened when we tried using Constructor Template Argument Deduction with `scoped_lock`, which had a constructor that takes a parameter pack of mutex types followed by an `adopt_lock_t`. This is perfectly legal for class templates, but in a function template, the variadic parameter pack needs to be the last argument for deduction to work. Since CTAD generates function templates from class template constructors, this can result in function templates with awkwardly placed parameter packs:

```
mutex m1, m2;
scoped_lock l{m1, m2, adopt_lock}; // Oops, adopt_lock is interpreted as a mutex
```

The solution is simple. We just fixed `scoped_lock` to take the `adopt_lock_t` before the parameter pack:

```
scoped_lock l{adopt_lock, m1, m2}; // Works with the C++17 version of scoped_lock
```

Advanced Best Practice: Deducible Constructors

Different ways of writing classes that behave the same in C++14 can behave differently in C++17! For example, the standard defines `vector` like:

```
template<class T, class Allocator = allocator<T>>
struct vector { vector(initializer_list<T>, Allocator = Allocator()); /* ... */ };
/* ... */
vector v = {1, 2, 3}; // Correctly deduces vector<int>
```

However, suppose a compiler implemented `vector` something like:

```
template<class T, class Allocator = allocator<T>>
struct vector {
  using value_type = Allocator::value_type;
  vector(initializer_list<value_type>, Allocator); /* ... */ };
/* ... */
vector v = {1, 2, 3}; // Oops, no way to deduce T!
```

In C++14, these two definitions are equivalent, so this was allowed by the *as-if* rule, but when doing CTAD they are different because only knowing `Allocator::value_type` in no way tells you what `T` is (especially because we don't even know what `Allocator` is), leading to a *non-deducible context*.

Deducible Constructors (cont)

Note that the indirection through `Allocator` is what causes the problem, as CTAD has special language to deduce from direct aliases like "`using value_type = T;`"

This example could easily have been avoided by simply writing `vector` without the redundant indirection in the first place in accordance with this best practice. However, what do you do if you have legacy classes that can't be rewritten or constructors that are intrinsically non-deducible, like constructing a `vector` from two iterators? Such cases can still be handled by explicitly writing deduction guides

```
template<class T, class Allocator = allocator<T>>
vector(initializer_list<T>, Allocator = Allocator()) -> vector<T, Allocator>;
template<typename Iter> vector(Iter b, Iter e) -> vector<typename Iter::value_type>;
// Outside <vector>
vector v2{b, e}; // Works now!
```

Advanced Best Practice: Pass by value

While CTAD is carefully designed to work well with both reference and value arguments [7], passing arguments by value can avoid a few gotchas. To see a good illustration of this, let's look at `pair` from the standard library. `pair<T, U>` has a constructor `pair(T const &, U const &)`. While this usually works well, the failure to decay could lead to a few surprises:

```
// What would CTAD for pair look like without deduction guides?
pair p{7, make_shared<string>("foo")}; // Nearly everything works great, like in this example
// But there are problems for types that can be "decayed"
void f() {}
int a[5];
pair p2(a, 3); // Deduces pair<int[5], int>, where pair<int *, int> would probably be better
pair p3(f, 3); // Deduces pair<void(), int>, which doesn't make sense
```

There are several choices for how to handle this.

- The first is simply to ignore it. How often do you need array-to-pointer and function-to-pointer conversion anyway ? Even if it happens, you can always explicitly give the template arguments if deduction fails (Just like in C++14)
- Have your constructor take arguments by value. While doing this when inappropriate would be the tail wagging the dog, in many cases, either by-value or by-reference is reasonable. In those cases, consider taking constructor arguments by value
- If the constructor takes a reference, creating an equivalent "by-value" deduction guide will force decay, which is exactly what the standard library does with `pair`:

```
template<class T, class U> pair(T, U) -> pair<T, U>;
```

Coolest Best Practice: Prepare for Concepts

- While this Best Practice is a little more abstract than the previous ones, it is my favorite because it is a new Best Practice that is showing up in multiple language features, suggesting a deeper unifying meaning that tells you that you are on the right track. More concretely, we had to put many constrained deduction guides in `unordered_set` to avoid constructor ambiguity. For example, it is unclear in an expression like `unordered_set(5, x)` whether you mean to invoke `unordered_set(size_type, Hash)` or `unordered_set(size_type, Allocator)`.

- Reflecting on the ambiguity, the real point here is that since `Hash` and `Allocator` are not constrained, they are just names. If only the compiler understood that Allocator is an allocator and Hash is a hashing function, there would be no ambiguity. But this is also one of the main ideas behind Concepts!

Preparing for Concepts (cont)

To make the point more clearly, imagine that there were `is_allocator_v` and `is_hash_v` type traits. Suppose we then added constraints to the definition of `unordered_set` as follows

```
template<class Key,
  class Hash = hash<Key>, class KeyEqual = equal_to<Key>, class Alloc = allocator<Key>,  // Usual
  typename = enable_if_t<is_hash_v<Hash>>, typename = enable_if_t<is_allocator_v<Alloc>>> // New constraints
> class unordered_set { /* ... */ };
```

Now, there is no longer a need to write a deduction guide to select the right constructor. It is right there in the constraints!

Of course, once Concepts become part of C++, you will use concepts and the above becomes

```
template<class Key, class Hash = hash<Key>, class KeyEqual = equal_to<Key>, class Alloc = allocator<Key>>
  requires is_hash_v<Hash> && is_allocator_v<Alloc>
class unordered_set { /* ... */ };
```

In fact, if you use a Concepts-enabled compiler today, code like the above correctly guides Constructor Template Argument Deduction without needing to manually tweak with deduction guides.

By following the new Best Practice of constraining your template classes, not only will Constructor Template Argument Deduction work better today, your code will be positioned to take advantage of Concepts in the future

template<auto>

- C++ has another great template feature that is as simple as CTAD is intricate
- Non-type template parameters can now be `auto`, just like you would expect
- Suppose we wanted to create a `constant` template that subsumes `integer_constant`, `bool_constant`, etc.
- It's as simple as

```
template <auto x> constexpr auto constant = x;
auto ci = constant<5>; // constant<int>
auto cd = constant<'c'>; // constant<char>
```

- It even handles cases which were impossible in C++14 where you can't write down the type of an object, like a lambda

Variadic folding

- Working with template parameter packs can be a mess in C++14, because it is awkward to expand them
- That got a whole lot better in C++17 (and should get even better in C++20)
- C++17 allows folding of parameter packs with binary operators
- This is easier to look at an example (from cppreference) and understand than explain the technicalities

```
template<typename... Args>
bool all(Args... args) { return (... && args); }

bool b = all(true, true, true, false);
 // within all(), the unary left fold expands as
 //  return ((true && true) && true) && false;
 // b is false
```

If constexpr

- In C++14, one often as to write many auxiliary functions to dispatch at compile-time
- Consider the famous optimized_copy example

```
template<typename I1, typename I2>
I2 optimized_copy(I1 first, I1 last, I2 out)
{
    while (first != last) {
        *out = *first;  ++out;++first;
    }
    return out;
}

template<typename T,
         enable_if_t<is_trivially_copy_constructible<T>::value> * = nullptr>
remove_const_t<T>* optimized_copy(T* first, T* last, remove_const_t<T>* out)
{
    memcpy(out, first, (last - first) * sizeof(T));
    return out + (last - first);
}
```