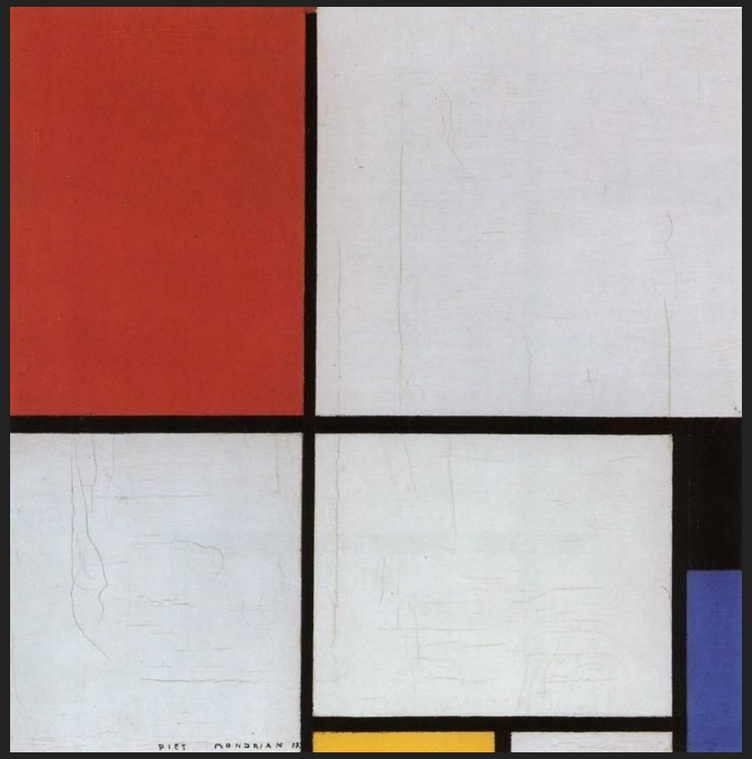
COMPOSABLE C++



BEN DEANE / @ben_deane

CPPCON 2021

IN THIS TALK

1. Composability defined
2. Composable types
3. Objects and patterns
4. Composability at compile time
5. Hierarchies and computation

COMPOSABLE?

What does "composable" mean?

Composable, reusable, extensible, flexible…?

Simple?

Do we recognize composability when we see it?

- STL algorithms?
- ranges?

THE COMPOSABILITY PROBLEM

"When I am working on a problem,
I never think about beauty.
But when I have finished,
if the solution is not beautiful,
I know it is wrong."

-- R. Buckminster Fuller

COMPOSABILITY LAID BARE

"Understanding why software fails is important, but the real challenge is understanding why software works."

-- Alexander Stepanov (via Sean Parent)

WHY SOFTWARE WORKS

Software works because of properties we can reason about.

This is the root of composability:

- Alice writes some code
- Bob takes that code
- Bob puts it to a use that Alice didn't foresee, and it works

WHAT COMPOSABILITY ISN'T

Composability is not:

WHAT COMPOSABILITY ISN'T

Composability is not:

- based on syntax

WHAT COMPOSABILITY ISN'T

Composability is not:

- based on syntax
- a product of test-driven development

WHAT COMPOSABILITY ISN'T

Composability is not:

- based on syntax
- a product of test-driven development
- one particular pattern

WHAT COMPOSABILITY ISN'T

Composability is not:

- based on syntax
- a product of test-driven development
- one particular pattern
- exclusive to one coding paradigm

NO, REALLY

NO, REALLY

Composability

NO, REALLY

Composability

is NOT

NO, REALLY

Composability
is NOT
about syntax!

COMPOSABILITY IS JOB #1

"If you want to design an allocator, you gotta make composition the first thing in your design, the first concern.

Getting composition right is getting the allocators right."

-- Andrei Alexandrescu (CppCon 2015)

As for allocators, so for everything.

PART 1: FUNCTIONS

Starting at the bottom and working our way up:

Function-level composability

THE RETURN TYPE

Choosing the wrong return type is one of the most common composability errors.

Often we don't even realise we're choosing.

bool

The simplest composable return type is `bool`.

```
auto do_a_thing() -> void;
auto do_a_thing_more_composably() -> bool;
```

Simple, but important.

WHY IS `bool` COMPOSABLE?

The simplest form of composability stems from properties of boolean algebra.

```
auto any_done = do_a_thing(1) or do_a_thing(2) or ...;
auto all_done = do_a_thing(1) and do_a_thing(2) and ...;
```

`bool` UNDERLIES SO MUCH

It seems so trivial that we often don't notice it.

Things that build on composability of `bool`:

`bool` UNDERLIES SO MUCH

It seems so trivial that we often don't notice it.

Things that build on composability of `bool`:

- control flow: almost all algorithms and loops, really

`bool` UNDERLIES SO MUCH

It seems so trivial that we often don't notice it.

Things that build on composability of `bool`:

- control flow: almost all algorithms and loops, really
- a lot of caching schemes (or idempotent calculations)

`bool` UNDERLIES SO MUCH

It seems so trivial that we often don't notice it.

Things that build on composability of `bool`:

- control flow: almost all algorithms and loops, really
- a lot of caching schemes (or idempotent calculations)
- polling/non-blocking functions

`bool` UNDERLIES SO MUCH

It seems so trivial that we often don't notice it.

Things that build on composability of `bool`:

- control flow: almost all algorithms and loops, really
- a lot of caching schemes (or idempotent calculations)
- polling/non-blocking functions
- Chain of Responsibility pattern

`bool` UNDERLIES SO MUCH

It seems so trivial that we often don't notice it.

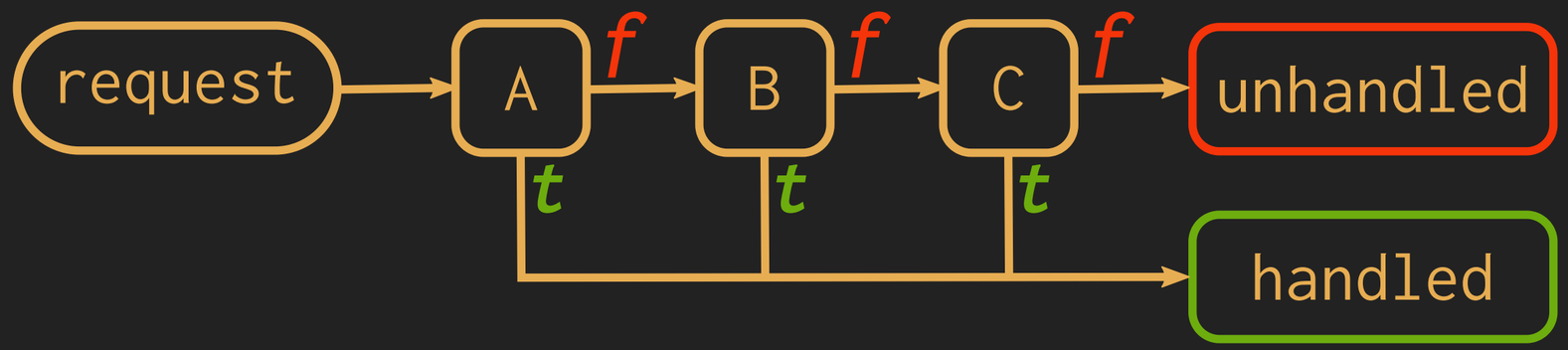
Things that build on composability of `bool`:

- control flow: almost all algorithms and loops, really
- a lot of caching schemes (or idempotent calculations)
- polling/non-blocking functions
- Chain of Responsibility pattern

Note: short-circuiting is not part of composability here;
it's just a semantic nicety of the operators.

CHAIN OF RESPONSIBILITY

A design pattern that exploits `bool`'s composability to achieve prioritization.



CASE STUDY: LOGGING

From "Easy to Use, Hard to Misuse: Declarative Style in C++" (CppCon 2018)

```
struct log_sink {
  virtual bool push(const log_entry&);
};
```

The `bool` return type is the key to composition here.

CASE STUDY: LOGGING

Given the `push` function, we can write various `log_sink` classes:

CASE STUDY: LOGGING

Given the `push` function, we can write various `log_sink` classes:

- "ordinary" sinks that send entries to different places (file, output window, etc)

CASE STUDY: LOGGING

Given the `push` function, we can write various `log_sink` classes:

- "ordinary" sinks that send entries to different places (file, output window, etc)
- a filter sink that runs a predicate on the entry and accepts it conditionally

CASE STUDY: LOGGING

Given the `push` function, we can write various `log_sink` classes:

- "ordinary" sinks that send entries to different places (file, output window, etc)
- a filter sink that runs a predicate on the entry and accepts it conditionally
- a sink that wraps another sink in an execution policy (e.g. for threaded logging)

CASE STUDY: LOGGING

Given the `push` function, we can write various `log_sink` classes:

- "ordinary" sinks that send entries to different places (file, output window, etc)
- a filter sink that runs a predicate on the entry and accepts it conditionally
- a sink that wraps another sink in an execution policy (e.g. for threaded logging)
- a sink that wraps several other sinks and:
  - sends a `log_entry` to all
  - sends a `log_entry` to the first one that accepts

CASE STUDY: LOGGING

Given the `push` function, we can write various `log_sink` classes:

- "ordinary" sinks that send entries to different places (file, output window, etc)
- a filter sink that runs a predicate on the entry and accepts it conditionally
- a sink that wraps another sink in an execution policy (e.g. for threaded logging)
- a sink that wraps several other sinks and:
  - sends a `log_entry` to all
  - sends a `log_entry` to the first one that accepts
- the null sink that accepts every entry and does nothing

BEYOND `bool`

Returning `void` gives no choice to the caller.

Returning `bool` gives one choice to the caller.

What can be returned to give more choices?

FROM `bool` TO `int`

The next simplest composable return type is `int`.

```
auto do_a_thing() -> void;
auto do_a_thing_more_composably() -> bool;
auto do_a_thing_even_more_composably() -> int;
```

FROM `bool` TO `int`

The next simplest composable return type is `int`.

```
auto do_a_thing() -> void;
auto do_a_thing_more_composably() -> bool;
auto do_a_thing_even_more_composably() -> int;
```

Spoiler: `bool` and `int` are in some sense
the *only* two composable return types we need…

WHY `int`?

Of course, `int` can represent N (32?) `bools`,
using `bitand` (`&`) and `bitor` (`|`) operators.

That's sometimes useful, but really
just an extension of the composability of `bool`…

WHY `int`?

With `int` we can say not just *if* something happened, but *how much* was taken care of.

Going from `bool` to `int` is like going from `find` to `accumulate`.

It also opens the door for more complex user-defined behaviour.

HOW MUCH WAS CALCULATED?

```
auto do_calculation(float) -> float;
```

`bool` gives a binary choice (naturally).

An arithmetic type allows incremental progress.

CASE STUDY: STEERING BEHAVIOURS

a.k.a. "boids" (by Craig Reynolds)

Boids are defined quite simply:

- mechanical parameters: position, heading, velocity etc
- a collection of behaviours

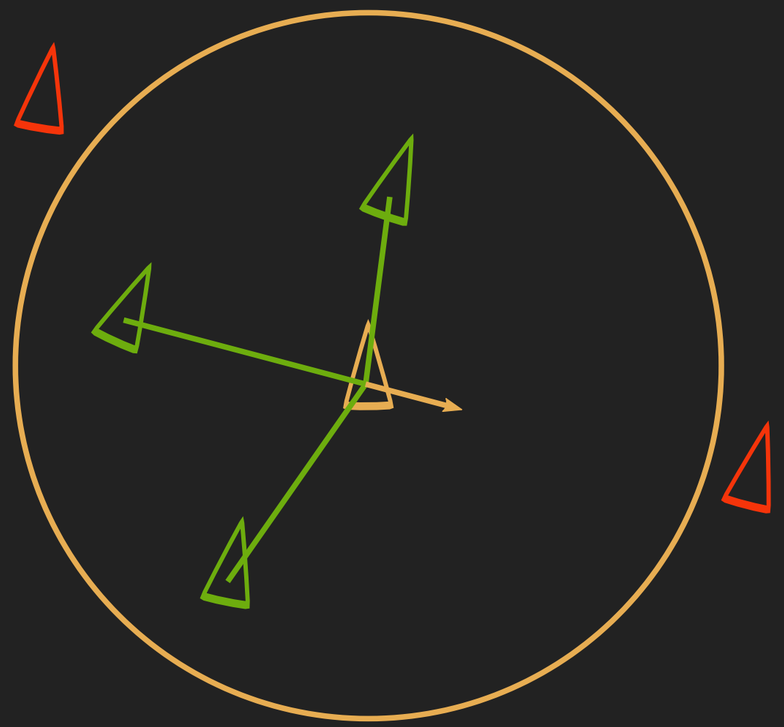
The calculation of each behaviour returns a force (arithmetic type).

The sum of forces (composition) given by all behaviours is applied to the boid.

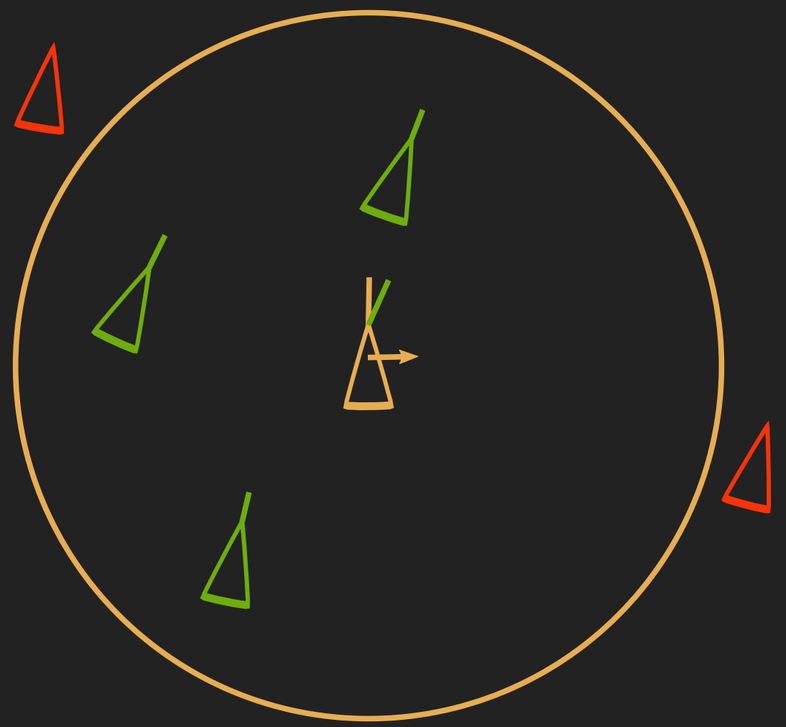
STEERING BEHAVIOURS

Classic boids flocking consists of three behaviours.

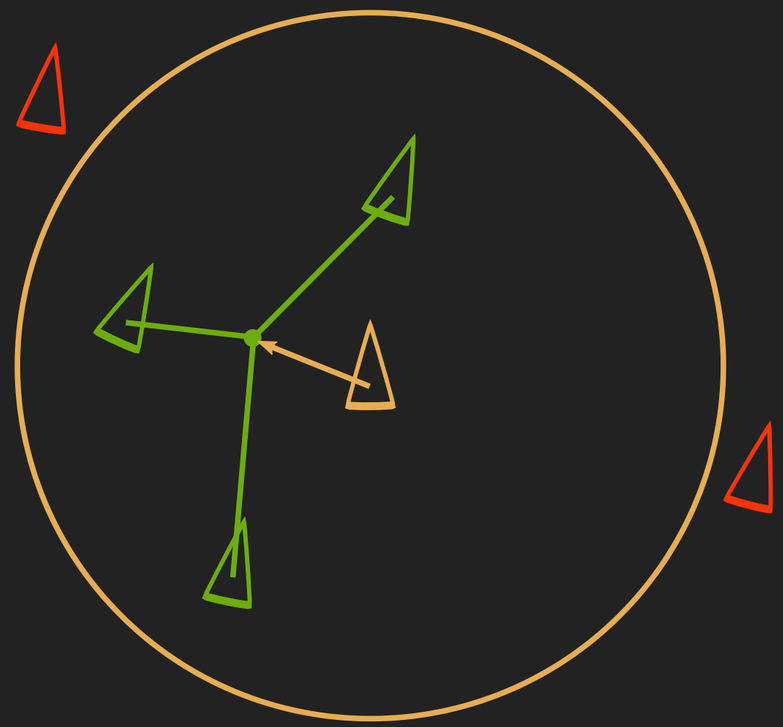
separation



alignment

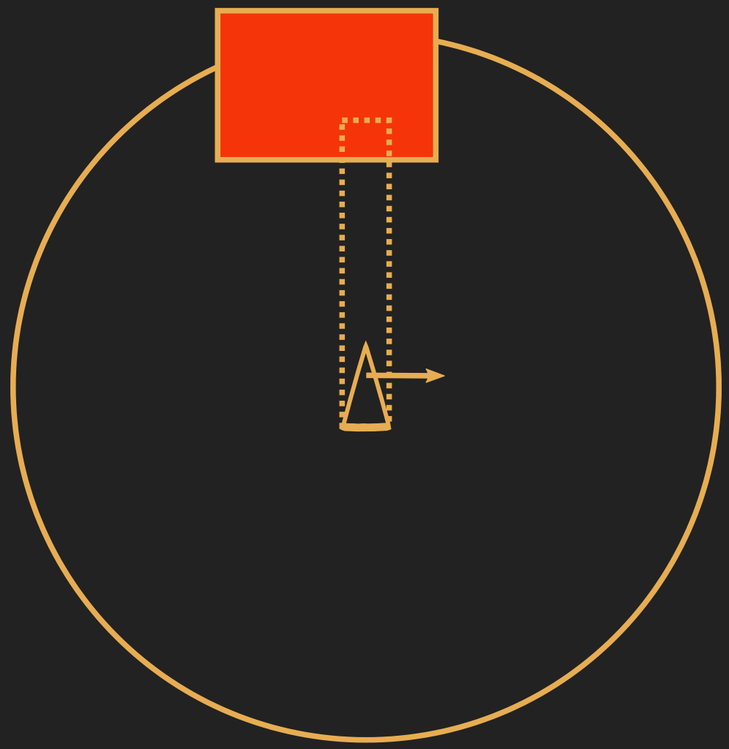


cohesion



MORE BEHAVIOURS

We can arbitrarily layer more behaviours into the calculation.



Obstacle avoidance, collision response, priority deferment, etc…

COMPOSABILITY FROM PRINCIPLES

At the core, steering behaviours is based on the composability of arithmetic types.

A single function returns a force that can be accumulated and applied.

The resulting framework has many uses:

- crowd simulations
- visualisations
- optimisation problems
- etc

MORE COMPOSABLE TYPES

So far we've seen `bool` and `int`.

The next step in composable types stems directly from here.

Instead of integral types, let's go to `T`.

MORE COMPOSABLE TYPES

`bool` is to `int`

as

`optional<T>` is to `collection<T>`

COMPOSING optional

optional can compose in the same way as bool. But it also has another trick.

```
struct T { ... };
auto combine(T x, T y) -> T;
```

When you have a T that has no "default value" for some combining operation, you can use optional to provide that value.

```
auto combine(optional<T> x, optional<T> y) -> optional<T> {
  if (x.has_value() and y.has_value()) {
    return combine(*x, *y);
  }
  return {};
}
```

COMPOSING `vector`

`optional` is like `bool`: we get presence or absence.

`vector` is like `int`: we get to combine quantities.

(It's been suggested that `optional` is just a `vector` with max size 1)

A COMMON PROBLEM

Many calculations are most simply formulated as recursive calculations.
Interesting variations often include mutual recursion.

```
auto calculate_recursively(const collection&) -> collection;
```

The most common inhibitor of recursive beauty is mixed up return types.

WE GET CONFUSED

```
auto calculate_recursively(branch_t) -> collection_t;
auto calculate_recursively(leaf1_t) -> value_t;
auto calculate_recursively(leaf2_t) -> value_t;
auto calculate_recursively(array_leaf_t) -> vector<value_t>;
```

When solving any kind of interesting recursive problem, the highest probability of confusion occurs from failing to keep return types uniform.

MATCH RETURN TYPE WITH PARAMETERS

Functions whose return type is the same as (one of) their parameter type(s) are the most composable.

```
auto do_calculation(float) -> float;
auto combine(T x, T y) -> T;
auto combine(optional<T> x, optional<T> y) -> optional<T>;
auto calculate_recursively(const collection&) -> collection;
```

And one of the easiest ways to make code ugly, if we don't line up types properly.

MATCH RETURN TYPE WITH PARAMETERS

Put more simply:

When working with a container, stay in the container!

e.g. if you're returning a `vector`, write your function to take a `vector`.

There is always a cost to exiting container-world.

- checking whether an `optional` is engaged
- checking whether a `vector` is empty
- waiting for a `future` value to materialize

COMPOSABLE FUNCTIONS

Composable functions stem from composable properties of their return types. We can:

- use `and` and `or` with `bool`
- use arithmetic, ordering operations, etc with numbers
- concatenate or merge collections

And we can take the output of a function and feed it back to another.

Almost all composability in object-pattern land builds on such type composability.

PART 2: OBJECTS & PATTERNS

The next rung on the ladder:

Object-level composability

SO WE HAVE AN INTERFACE

```
struct thing_doer {
  virtual auto frob(const widget&) -> composable_type_t;
};
```

What classes should we write to this interface, to enable higher-level composability?

FIRST: ZERO

The first object to write is the one that does nothing, composably.

FIRST: ZERO

The first object to write is the one that does nothing, composably.

- the logger that accepts an entry and produces no log

FIRST: ZERO

The first object to write is the one that does nothing, composably.

- the logger that accepts an entry and produces no log
- the allocator that always returns `nullptr`

FIRST: ZERO

The first object to write is the one that does nothing, composably.

- the logger that accepts an entry and produces no log
- the allocator that always returns `nullptr`
- the parser or validator that fails

FIRST: ZERO

The first object to write is the one that does nothing, composably.

- the logger that accepts an entry and produces no log
- the allocator that always returns `nullptr`
- the parser or validator that fails
- the visitor that uses the identity operation

FIRST: ZERO

The first object to write is the one that does nothing, composably.

- the logger that accepts an entry and produces no log
- the allocator that always returns `nullptr`
- the parser or validator that fails
- the visitor that uses the identity operation
- the request that does nothing

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state
- side-effectful behaviour

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state
- side-effectful behaviour
- world-switching

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state
- side-effectful behaviour
- world-switching

Solutions:

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state
- side-effectful behaviour
- world-switching

Solutions:

- use composable types

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state
- side-effectful behaviour
- world-switching

Solutions:

- use composable types
- apply SOLID principles

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state
- side-effectful behaviour
- world-switching

Solutions:

- use composable types
- apply SOLID principles
- avoid world-switching

THE ZERO OBJECT

If writing this is difficult, it's a good sign our composability abstraction is not right.

Possible problems:

- no default state
- coupled actions or state
- side-effectful behaviour
- world-switching

Solutions:

- use composable types
- apply SOLID principles
- avoid world-switching
- find a better abstraction?

SECOND: PASSTHROUGH

The second object to think about is the one that `delegates` to another presumed object in some way.

If your return type is `bool` or `optional`, you can write the "if-else" object:
try option A, and if it fails, employ option B.

At this point, the "if-else" object may extend to the "any-of" and "all-of" objects.

CASE STUDY: ALLOCATORS

> `std::allocator` *is to Allocation*
> *what* `std::vector` *is to Vexation*
>
> *-- Andrei Alexandrescu, CppCon 2015*

```
struct Allocator {
  auto allocate(size_t) -> Blk;
  auto owns(Blk) const -> bool;
  auto deallocate(Blk) -> void;
};
```

`Blk` contains pointer (=> `bool`) and size.

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator
- fixed size allocators (chunks of N bytes)

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator
- fixed size allocators (chunks of N bytes)
- slab/arena allocators

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator
- fixed size allocators (chunks of N bytes)
- slab/arena allocators
- freelist based allocators

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator
- fixed size allocators (chunks of N bytes)
- slab/arena allocators
- freelist based allocators
- affix allocator (metadata prefix/suffix)

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator
- fixed size allocators (chunks of N bytes)
- slab/arena allocators
- freelist based allocators
- affix allocator (metadata prefix/suffix)
- hi/lo allocator, bucket allocator, etc

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator
- fixed size allocators (chunks of N bytes)
- slab/arena allocators
- freelist based allocators
- affix allocator (metadata prefix/suffix)
- hi/lo allocator, bucket allocator, etc
- mallocator

CASE STUDY: ALLOCATORS

With just that interface, we can implement:

- fallback allocator (try A, if it fails, try B)
- stack allocator
- fixed size allocators (chunks of N bytes)
- slab/arena allocators
- freelist based allocators
- affix allocator (metadata prefix/suffix)
- hi/lo allocator, bucket allocator, etc
- mallocator
- compositions of all of the above (powered by fallback allocator)

COMPOSABILITY IN THE FALLBACK ALLOCATOR

The fallback allocator first tries a primary;
if that fails, it *falls back* to the other.

```
template <class Primary, class Fallback>
struct FallbackAllocator : Primary, Fallback {
  auto allocate(size_t n) -> Blk {
    if (const auto r = Primary::allocate(n); r.ptr) {
      return r;
    }
    return Fallback::allocate(n);
  }
};
```

This relies on the composable properties of `bool`.

N-ARY FALLBACK ALLOCATOR

We can turn a binary pattern into an n-ary pattern:

```
template <class... As>
struct FallbackAllocator : As... {
  auto allocate(size_t n) -> Blk {
    Blk r{};
    auto alloc = [&] <class A> () -> bool {
      r = static_cast<A*>(this)->allocate(n);
      return r.ptr;
    };
    (... or alloc.template operator()<As>());
    return r;
  }
};
```

"FUNCTIONAL PROGRAMMING"

Functional programming in C++? Polarising?

Higher order functions that take functions as arguments and return functions.

"A m**** is just a m***** in the c******* of…"

WHO WRITES CODE LIKE THIS?

```
auto compose(auto f1, auto f2) {
  return [f1, f2] (auto arg) {
    if (auto result = f1(arg); result) {
      return result;
    }
    return f2(arg);
  };
};
```

Who *writes* code like this?

WHO WRITES CODE LIKE THIS?

```
auto compose(auto f1, auto f2) {
  return [f1, f2] (auto arg) {
    if (auto result = f1(arg); result) {
      return result;
    }
    return f2(arg);
  };
};
```

Who *writes* code like this?

We. All. Do.

PATTERNS: COMPOSITE

"Composite lets clients treat individual objects and compositions of objects uniformly."

In classical OO-style, this is done with an abstract base class that declares operations on both composite and leaf classes.

```
struct graphic {
  virtual auto draw() -> void = 0;
  virtual auto add(graphic&) -> void = 0;
  virtual auto remove(graphic&) -> void = 0;
};

struct leaf : graphic { ... };
struct collection : graphic { ... };
```

PATTERNS: COMPOSITE

Composite is a useful pattern because it allows us to compose structure directly without conditions in the handling code.

We don't have a dichotomy between a leaf and a collection.

This means we can write functions against one interface that work with both.

WHAT IS THE LEARNING HERE?

Q. What makes composite nice to use?

WHAT IS THE LEARNING HERE?

Q. What makes composite nice to use?

A. We don't have to put conditions in the calling code.

HERE'S THE LEARNING

No raw conditions.

A *raw condition* is any binary condition that doesn't serve one of two purposes:

- intrinsic optionality present inside a data type
- a configuration or construction-time choice

Notice this includes:

- regular `if`
- the conditional operator
- `if constexpr`

BRIEF ASIDE

Remember "C++ Seasoning"?

Of course. We all remember "no raw loops".

I suggest watching that talk again. (For the rest of it.)

NO RAW CONDITIONS

Put another way -

We should strive to put `if` statements in two places only:

- at the bottom of the stack (built into data structure)
- at the top of the stack (construction time)

THE SECRET OF UNIFORMITY

Good composition happens when functions across the object space:

- take the same types
- return the same types
- take the same types as they return (or related ones)

WELL-BEHAVED FUNCTIONS

Functions or methods with the following signatures
usually make good compositional sense:

```
auto func() -> bool;
auto func(T) -> collection<T>;
auto func(collection<T>) -> collection<T>;
```

PROBLEMATIC FUNCTIONS

Functions or methods with the following signatures are often problematic:

```
auto func(int) -> bool;
auto func(optional<T>) -> T;
auto func(collection<T>) -> T;
```

As well as mixing functions that return `T` and `collection<T>`.

INTERLUDE: COMPILE-TIME

So far we've seen everything in value-space.

The same principles apply (perhaps even more so) to type-space.

COMPILE-TIME TYPE CALCULATIONS

In this form of compile-time programming:

- values become types
- functions become templates (usually class/alias)
- data structures are typically type lists or pairs

COMPILE-TIME FAIL

We have pretty much one data structure: the type list.

The same problematic function signatures apply.

```
// auto func(collection<T>) -> T;
template <typename... Ts>
struct func {
  using type = some_T;
};
```

WE WOULDN'T WRITE THIS CODE

```
if (conditionA) {
  if (subConditionX) {
  } else if (subConditionY) {
  } else {
  }
} else if (conditionB and conditionC) {
  if (not subConditionZ) {
  }
}
```

If we saw this structure in code review, we'd balk.

BUT WE CAN END UP WITH THIS

```
if constexpr (conditionA) {
  if constexpr (subConditionX) {
  } else if constexpr (subConditionY) {
  } else {
  }
} else if constexpr (conditionB and conditionC) {
  if constexpr (not subConditionZ) {
  }
}
```

"At least it's better than SFINAE."

REFLECTION IS COMING

Maybe for some of you, it's already here in some form.

We are going to get a lot of metafunctions building on the following general form:

```
template <typename T>
auto members_of(T) -> type_list<TMembers...>;
```

What should happen if `T` is an `int`? What does that mean for when T is a `struct`?

REFLECTION IS COMING

We are going to get a lot of metafunctions in the following pattern:

```
template <typename T>
auto for_each_member_recursively(T) -> /* what? */;
```

What should this function return?

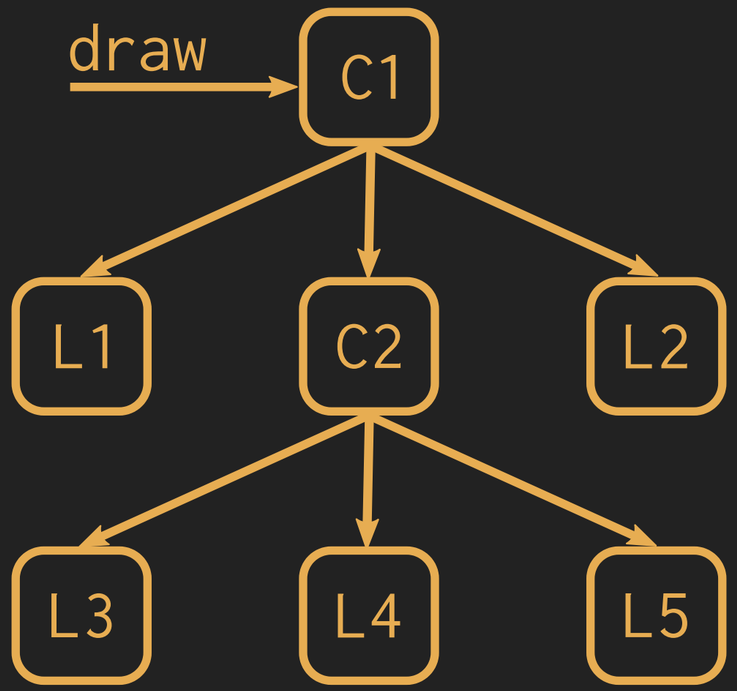
How can we compose functionality like this?

PART 3: STRUCTURES

The top rung on the ladder (for this talk):

Structural composability

TRAVERSING HIERARCHIES



FIRST PROBLEM

Many languages have object-oriented features.

They make it easy to add new types.

It's difficult to add new behaviour though.

This is (half of) the expression problem.

WHAT SOLVES THIS PROBLEM?

Another pattern, of course.

The visitor pattern.

VISITOR PATTERN

*"The **Visitor pattern** is the most widely misunderstood pattern in all of Design Patterns, which is really saying something [...] The pattern isn't about "visiting", [... it's] really about approximating the functional style within an OOP language."*

-- Bob Nystrom, Crafting Interpreters

VISITOR PATTERN

In an OO language, types (classes) are easy to add,
but adding behaviour (methods) require touching the world.

In a functional language, it's the other way around: functions are easy to add,
but adding types requires updating pattern matching everywhere.

The visitor pattern approximates the functional experience in an OO language.

WAS THAT THE ONLY PROBLEM?

Fundamentally that's still a choice of one way or the other.
And that's fine for some problems (like representing expressions).

The expression problem always remains to some degree.

We have another problem - composability means:

- structured hierarchies
- hierarchical composition and extension
- run computations over hierarchies

COMPUTATION AND TRAVERSAL

When structures are flat, we're already good at separating computation and traversal.

```
auto total_length(const vector<string>& v) -> int {
  return transform_reduce(begin(v), end(v),
                          0, plus{},
                          [] (const auto& s) { return size(s); });
}
```

```
vector<string> v = {"Hello", "CppCon", "2021"};
std::cout << total_length(v); // 15
```

We have lots of algorithms that separate where (traversal with iterators)
from what (the operation passed in).

HIERARCHICAL COMPUTATION

When structures are hierarchical, we're much less good at separating computation and traversal.

Consider how to sum the lengths in a rose tree.

```
template <typename T>
struct rose_tree : variant<T, vector<rose_tree<T>>> {
  using variant<T, vector<rose_tree<T>>>::variant;
};
```

HIERARCHICAL COMPUTATION

```
auto total_length(const rose_tree<string>& t) -> int {
  return visit(
    overloaded{
      [] (const string& leaf) -> int { return size(leaf); },
      [] (const auto& branch) -> int {
        return transform_reduce(
          begin(branch), end(branch),
          0, plus{},
          [] (const auto& t) -> int { return total_length(t); }); }},
    t);
}
```

The traversal is mixed with the operation.

SEPARATE WHAT FROM WHERE

```
template <typename T, typename TInit, typename FBranch, typename FLeaf>
auto reduce_tree(const rose_tree<T>& t, TInit i, FBranch r_op, FLeaf t_op) {
  return std::visit(overloaded{
    [&] (const T& leaf) { return t_op(leaf); },
    [&] (const auto& branch) {
      return std::transform_reduce(
          std::begin(branch), std::end(branch), i,
          r_op, [&] (const auto& t) { return reduce_tree(t, i, r_op, t_op); }); }},
  t);
}
```

Here is the "where": `reduce_tree` embodies the traversal.

SEPARATE WHAT FROM WHERE

```
auto total_length(const rose_tree<string>& t) -> int {
  return reduce_tree(t, 0, plus{},
                     [] (const auto& leaf) -> int { return size(leaf); });
}
```

Here is the "what": how to treat a branch and a leaf.

C++ IN FLATLAND

When all you have is iterators/ranges, everything looks flat.

Every standard structure is flat… many non-standard structures are not.

But they are amenable to the same kind of abstraction that leads to composability!

FROM FLATLAND TO HIGHER DIMENSIONS

```
// this is not possible in C++ but a useful structure to imagine...
struct nil{};
template <typename T>
struct vector : variant<nil, pair<T, vector<T>>> {
  using variant<nil, pair<T, vector<T>>>::variant;
};
```

```
template <typename T, typename TInit, typename Op>
auto reduce(const T& t, TInit i, Op op) {
  return visit(overloaded{
    [&] (nil) { return i; },
    [&] (const auto& v) {
      return op(v.first, reduce(v.second, i, op)); }},
  t);
}
```

GENERIC HIERARCHY TRAVERSAL

By applying a functional lens, we can see how to traverse hierarchies in higher dimensions.

This unlocks structural and computational composition by separating what from where.

We can go a little further though…

WHAT, WHERE - AND WHEN?

```
template <typename T, typename TInit, typename Op>
auto reduce(const T& t, TInit i, Op op) {
  return visit(overloaded{
    [&] (nil) { return i; },
    [&] (const auto& v) {
      return op(v.first, reduce(v.second, i, op)); }},
  t);
}
```

Where's the "when" in this picture?

WHY IS ANY OF THIS IMPORTANT?

- Alice writes some code
- Bob takes that code
- Bob puts it to a use that Alice didn't foresee, and it works

This. This is only possible with a principled approach to composability.

- types
- functions
- objects
- structures
- computation

CONCLUSIONS

- Composability is not about syntax, it's about semantics.
- Pattern/object-level composability stems from function-level composability, stems from type-level composability.
- When working in a collection, stay in the collection.
- Conditions inhibit composition: avoid raw `if` statements.
- Compile-time capability is still in its infancy (`if constexpr` is *not* better than SFINAE).
- The key to structural composition is breaking out of flatland and separating what, where and when.