Message Handling

with Boolean Algebra

Ben Deane / CppCon / 2024-09-16

Alternative Title

"The Unreasonable Effectiveness of Boolean Algebra in Software Design,
Showing the Particular Application of a Message Handling Library,
with an Excursion into the Roots of Programming"

Frontmatter

No AI/LLM was used in the creation of this talk.

Code is simplified for slides; may have some errors in translation.
Real code is at `https://github.com/intel/compile-time-init-build`

The code here is mostly east-const. I have no particular preference.
Spot a const-west and win a prize!*

*the respect of your peers

Accessibility matters

Accessibility matters, and it's for everyone.

There are many simple things that can make presentations more accessible.
Here are some things I've adopted.

- Body font: Atkinson Hyperlegible [*thanks Linus Boman*]
- Code font: Berkeley Mono
- Dark-on-light (optimised for in-the-room, not YouTube)
- Code color theme: modus-operandi-tinted (WCAG AAA) [*thanks Prot*]
- SVG diagrams for resolution-independence

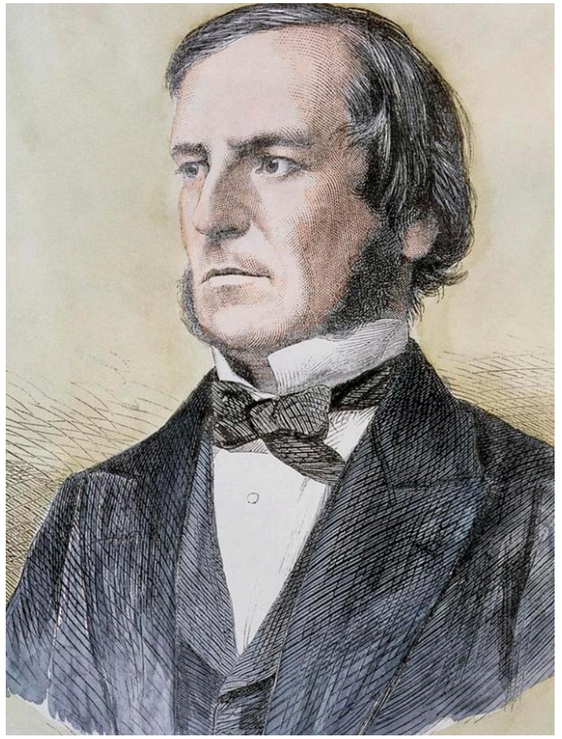
What this talk is about

The workings of a message-handling library.

- How messages and the fields in them are specified.
- Efficiently identifying (with *matchers*) a message coming off the wire.
- The role of Boolean algebra in composing matchers.
- Understanding Boolean implication and using it to simplify matchers.

This talk is a series of baby steps, each building on the previous.
And by the end, we really get somewhere.

With thanks to...



...a couple of the giants whose shoulders we stand on.
(Augustus de Morgan & George Boole)

Also to Trevor Huxtable, my high school physics & electronics teacher.

How it started

https://github.com/intel/compile-time-init-build

Issue #374

Author: lukevalenty (Luke Valenty)

Assigned to: elbeno (Ben Deane)

> Factor arbitrary matcher expressions into "sum-of-products" expressions

Part 1

Fields and messages: how they are structured and specified.

Fields

A *field* is:

- a human-readable *name*
- a *type* (almost always integral or enumeration)
- one or more *locators* that define the layout

Field *locators* are conventionally specified with 3 items:

- word index (in 32-bit words)
- most significant bit (inclusive!)
- least significant bit

IPv4 Header

Word	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19 - 23	24 - 31
0	0	**Version**				**IHL**				**DSCP**						**ECN**		Total Length				
1	32	Identification																Flags			Fragment Offset	
2	64	Time To Live								Protocol								Header Checksum				
3	96	Source IP Address																				
4	128	Destination IP Address																				
5	160	Options (if IHL > 5)																				
⋮	⋮																					
14	448																					

IPv4 Header Fields

```
 1: using version =
 2:     field<"version", std::uint8_t>::located<at{0_dw, 3_msb, 0_lsb}>;
 3:
 4: using ihl =  // internet header length
 5:     field<"ihl", std::uint8_t>::located<at{0_dw, 7_msb, 4_lsb}>;
 6:
 7: using dscp = // differentiated services code point
 8:     field<"dscp", std::uint8_t>::located<at{0_dw, 13_msb, 8_lsb}>;
 9:
10: using ecn =  // explicit congestion notification
11:     field<"ecn", std::uint8_t>::located<at{0_dw, 15_msb, 14_lsb}>;
```

Fields have 2 methods

```
1: template <range R>
2: constexpr static auto extract(R &&r) -> value_type;
3:
4: template <range R>
5: constexpr static auto insert(R &&r, value_type const &value) -> void;
```

Given a range over underlying data, a `field` can read and write its value in the layout.

Note: a `field` is an empty type! Definition, not data.

Field design

A `field` is an empty type.

It is two separate things:

- the field `spec` (name and type)
- the field `locator` (where the field lives in storage)

Separating these helps to decouple things:
the same field `spec` can be located differently in different messages.

Field locator options (1)

Word	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19 - 23	24 - 31
0	0	Version				IHL				DSCP						ECN		Total Length				
1	32	Identification																**Flags**			Fragment Offset	

One convention is to locate fields by word index, msb and lsb.

```
1: using flags =
2:     field<"flags", std::uint8_t>::located<at{1_dw, 18_msb, 16_lsb}>;
```

But another is to just use "raw" bit offsets.

```
1: using flags =
2:     field<"flags", std::uint8_t>::located<at{50_msb, 48_lsb}>;
```

Field locator options (2)

Multiple locations!

Word	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 - 23	24 - 31
0	0	split								split									
1	32																		

```
1: using split_field =
2:   field<"split", std::uint8_t>::located<at{0_dw,  3_msb, 0_lsb},
3:                                        at{0_dw, 11_msb, 8_lsb}>;
```

Yes, this is a use case.

Messages

Given this, we can define a *message layout* as
a named collection of `field`s.
This definition is also an empty type.

A message instance is a combination of the layout definition and storage.

```
1: using ipv4_header_layout = message<"ipv4", version, ihl, dscp, ecn, ...>;
2:
3: owning<ipv4_header_layout> ipv4_header; // layout + right-sized array
4:
5: auto version = ipv4_header.get("version"_field);
6: ipv4_header.set("ihl"_field = 5);
```

A `message` has `get` and `set` functions which delegate to `extract` & `insert`
functions on the appropriate `field`.

Layout definition + storage = message

When we want to send a message, we own the data.

```
1: owning<ipv4_header_layout> ipv4_header;
2: // now set some fields...
```

When receiving a message, we interpret it using a view of the data.

```
1: const_view<ipv4_header_layout> ipv4_header{recv_buffer};
2: // now test some fields...
```

Field & Message structure

We can ask about (or assert) properties of `field`s and `message`s.
For example:

- can a message's fields fit into the given storage?
- is a field's type adequately covered by its locators (or vice versa)?
- does a field occur in a message?

Of course these are all decidable at compile time
(given that storage is statically known).

Part 2

Matching on fields in a message.

Matchers

To determine the correct message type for data arrived off the wire, we use a *matcher*.

```
1: template <typename T>
2: concept matcher =
3:   requires(T const &t, message const &msg) { // a prototypical message
4:     { t(msg) } -> std::convertible_to<bool>;
5: };
```

A matcher is a predicate on a message.

A simple matcher

```
1: template <typename Field, auto Value> struct equal_to_t {
2:   constexpr auto operator()(auto const &msg) const -> bool {
3:     return Value == msg.template get<Field>();
4:   }
5: };
```

This matcher tests that a given field matches a given value.

Notice: a matcher is an empty function object.

A more generic matcher...

```
1: template <typename Op, typename Field, auto Value>
2: struct relational_matcher_t {
3:   constexpr auto operator()(auto const &msg) const -> bool {
4:     return Op{}(Value, msg.template get<Field>());
5:   }
6: };
```

And then we have convenience aliases:

```
1: template <typename Field, auto Value>
2: using equal_to_t = relational_matcher_t<std::equal_to<>, Field, Value>;
3:
4: template <typename Field, auto Value>
5: using less_than_t = relational_matcher_t<std::less<>, Field, Value>;
6:
7: // etc...
```

Matchers *are* Boolean values

We can treat a function that returns a `bool` just like a `bool`.
(They're isomorphic.)

All we need to do is to provide `and`, `or`, and `not` class templates that wrap matchers and implement `operator()` appropriately.

The resulting matcher will be an expression template.

`and` Matcher

```
1: template <matcher L, matcher R>
2: struct and_t {
3:   L lhs;
4:   R rhs;
5:
6:   constexpr auto operator()(auto const &msg) const -> bool {
7:     return lhs(msg) and rhs(msg);
8:   }
9: };
```

`or` Matcher

```
1: template <matcher L, matcher R>
2: struct or_t {
3:   L lhs;
4:   R rhs;
5:
6:   constexpr auto operator()(auto const &msg) const -> bool {
7:     return lhs(msg) or rhs(msg);
8:   }
9: };
```

`not` Matcher

```
1: template <matcher M>
2: struct not_t {
3:   M m;
4:
5:   constexpr auto operator()(auto const &msg) const -> bool {
6:     return not m(msg);
7:   }
8: };
```

The usual Boolean operators

```
 1: template <matcher L, matcher R>
 2: constexpr auto operator and(L const &lhs, R const &rhs) {
 3:     return and_t{lhs, rhs};
 4: }
 5:
 6: template <matcher L, matcher R>
 7: constexpr auto operator or(L const &lhs, R const &rhs) {
 8:     return or_t{lhs, rhs};
 9: }
10:
11: template <matcher M>
12: constexpr auto operator not(M const &m) {
13:   return not_t{m};
14: }
```

Operator overloads for `and`, `or` and `not` on matchers.

The library in context

Now we have:

- ways to define fields and messages
- ways to match on messages
- ways to combine matchers

Let's look at some usage in context.

The library in context

```
 1: // define the fields and message
 2: using type_f = field<"type", std::uint8_t>
 3:                     ::located<at{0_dw, 7_msb, 0_lsb}>;
 4: using msg_layout = message<"msg", type_f>; // and more fields...
 5:
 6: // a callback is a matcher and a function to call
 7: auto cb = callback<"cb", msg_layout>(
 8:     "type"_field == 42_c,  // a DSL that defines a matcher
 9:     [] (const_view<msg_layout> msg) { /* do something */ });
10:
11: // a handler encapsulates callbacks
12: auto h = handler(cb);
13:
14: // when we handle some data, message views are overlayed
15: // and functions are called based on which matchers match the data
16: auto data = std::array{ ... };
17: h.handle(data);
```

DSL for easy composition

```
1: auto cb = callback<"cb", msg_layout>(
2:     "type"_field == 42_c
3:     and ("subtype"_field == 17_c or "subtype"_field == 21_c)
4:     and "other"_field > 9_c,
5:     [] (const_view<msg_layout> msg) { /* do something */ });
```

We've made it easy for users to express constraints and match on messages.

The resulting matcher is an expression template.

Boolean algebra primer/refresher

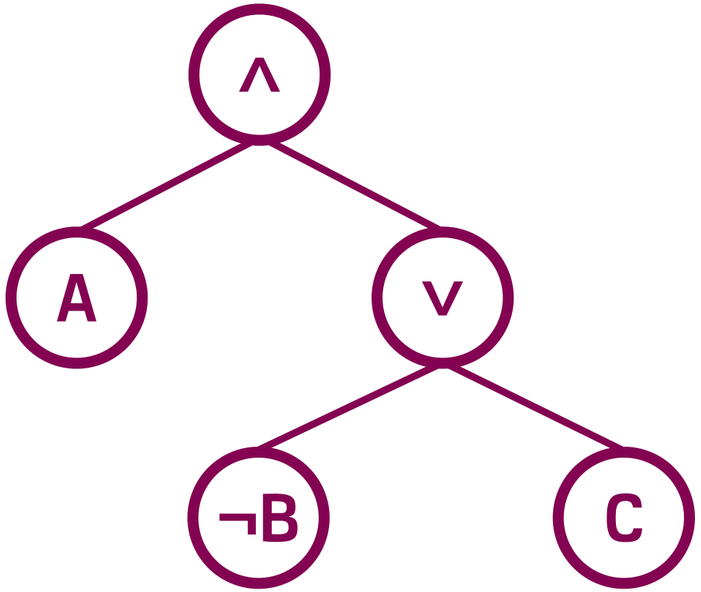
In upcoming slides you'll see some Boolean expressions (not necessarily in C++).

Just so we're all clear, I'm using this convention:

- `∧` means `and`: `A ∧ B` reads "A and B"
- `∨` means `or`: `A ∨ B` reads "A or B"
- `¬` means `not`: `¬A` reads "not A"

OK, now what?

We have an expression template
representing a combination of matchers
that can be run on a message.



We'd like to make sure that evaluating this at runtime is as simple as possible.

Example matcher problems

We've made it easy for people to write complex expressions, possibly containing tautologies or contradictions, or at least common parts...

How does the library know how to simplify things like:

```
less_than<5> or greater_equal<5>
```

```
less_than<5> and less_than<7>
```

```
not less_than<5> and greater_equal<5>
```

(Here I used the Feynman problem-solving algorithm.)

Part 3

Simplifying matchers.

OK, simplify

You're thinking: let's simplify the expression with our "normal" rules of Boolean algebra.

But... how?

What does the library need to know about matchers in order to simplify them?

And how can it know in a generic way?

Let's start easy

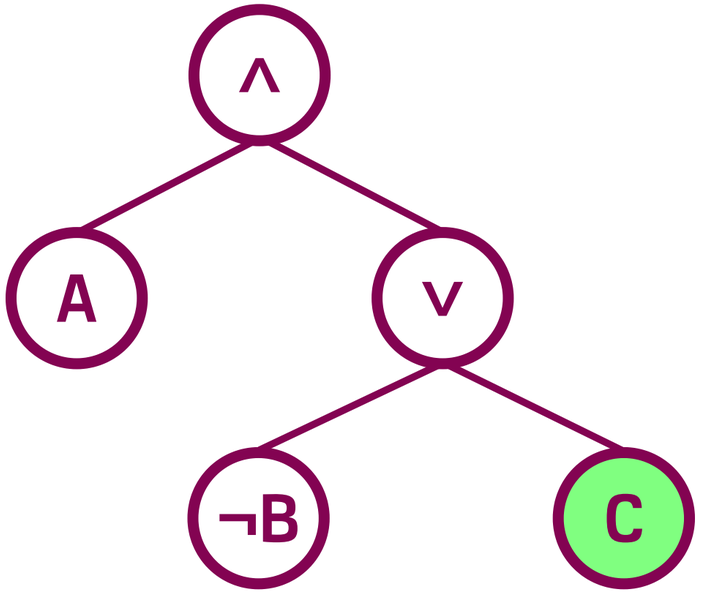
Let's take a step back and define two matchers.

```
 1: struct always_t {
 2:   constexpr auto operator()(auto const &) const -> bool {
 3:     return true;
 4:   }
 5: } always{};
 6:
 7: struct never_t {
 8:   constexpr auto operator()(auto const &) const -> bool {
 9:     return false;
10:   }
11: } never{};
```

These seem like they will be useful. When in doubt, start with the basics.

If we can simplify

If we know at compile time that `C` is `true`,
We can replace it with an `always_t`.



And if we can simplify in general, this whole tree would then collapse,
and we'd just evaluate `A` at runtime.

Let's write a simplify function

We'll make `simplify` a customization point that each matcher type can overload according to its own needs.

```
1: template <matcher M>
2: constexpr auto simplify(M const &m) -> M {
3:   return m;
4: }
```

And the default simplification is no simplification. Just the identity function.

Simplification - `and` and `or`

Now that we have `always_t` and `never_t`,
we can simplify `and_t` and `or_t`.

```
 1: template <matcher L, matcher R>
 2: constexpr auto simplify(and_t<L, R> const &m) {
 3:   auto l = simplify(m.lhs);
 4:   auto r = simplify(m.rhs);
 5:
 6:   if constexpr (/* l is a never_t or r is a never_t */) {
 7:     return never;
 8:   } else if constexpr (/* r is an always_t */) {
 9:     return l;
10:   } else if constexpr (/* l is an always_t */) {
11:     return r;
12:   } else {
13:     return and_t{l, r};
14:   }
15: }
```

Simplification, easy mode

We can code the "normal" laws of Boolean algebra.

AND-ish			OR-ish	
Expression	**Simplification**		**Expression**	**Simplification**
`A ∧ true`	`A`		`A ∨ true`	`true`
`A ∧ false`	`false`		`A ∨ false`	`A`
`A ∧ A`	`A`		`A ∨ A`	`A`
`A ∧ ¬A`	`false`		`A ∨ ¬A`	`true`

Well... hmm... this is only some of them...

This is getting wordy

We could do that...
we need to account for commutativity in every case, too...
it's quite a lot.

And we didn't even solve this problem yet:

```
less_than<5> or greater_equal<5>
```

These are matchers of different and arbitrary types.
How can the library simplify this to `always`?

Customizing `not`

```
less_than<5> or greater_equal<5>
```

If we know that these expressions are complements of each other, we could simplify this by "normal" Boolean algebra rules.

So let's customize negation!

A new `not` operator

```
1: template <matcher M>
2: constexpr auto operator not(M const &m) {
3:   return negate(m);
4: }
```

Instead of always returning a `not_t`, let's use another customization point!

The default `negate` function

```
1: template <matcher M>
2: constexpr auto negate(M const &m) -> not_t<M> {
3:   return {m};
4: }
```

Of course the default way to negate something is to put a `not_t` around it.

And an overload for `not_t`

```
1: template <matcher M>
2: constexpr auto negate(not_t<M> const &n) -> M {
3:   return n.m;
4: }
```

If we're negating something that's already a `not_t`,
we can just return the thing inside the `not_t`.

(i.e. unwrap, rather than double-wrapping.)

Customizing `negate`

To negate a relational matcher, we can use the "inverse" operation.

```
1: template <typename Op, typename Field, auto Value>
2: constexpr auto negate(relational_matcher_t<Op, Field, Value> const &) {
3:   return relational_matcher_t<
4:       inverse_op_t<Op>, Field, Value>{};
5: }
```

Where `Op` is e.g. `std::less<>` and `inverse_op_t` is defined appropriately.

OK, that's good

Now we have genericized the idea of negation/complement.
So our library understands things besides `not_t` terms.

`less_than<5> or greater_equal<5>`

With the "normal" simplifications, this problem is solved,
because we know that:

`A ∨ ¬A ≡ T`

Next problem

```
less_than<5> and less_than<7>
```

What are we going to do with this?

We'd like to simplify...

Part 4

An aside. And a quiz!

Don't worry, the questions should be easy.

Q. Can I obtain it?

```
1: // I have an A:
2: A a = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a B?
```

Q. Can I obtain it?

```
1: // I have an A:
2: A a = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a B?
```

Of course! (I just call `f(a)`.)

Q. Can I obtain it?

```
1: // I have an A:
2: A a = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a pair<A, B>?
```

Q. Can I obtain it?

```
1: // I have an A:
2: A a = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a pair<A, B>?
```

Yes! (I'll make the `B` using `f(a)`.)

Q. Can I obtain it?

```
1: // I have a variant<A, B>:
2: variant<A, B> v = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a B?
```

Q. Can I obtain it?

```
1: // I have a variant<A, B>:
2: variant<A, B> v = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a B?
```

Again yes! (If there's an `A` in the `variant`, I'll use `f`.)

Q. What is Ben talking about?

Oh no, is this is about functional programming?

It's about Boolean algebra.

Which is what you just did.

Even if it didn't seem like it.

Q. What does `->` mean?

```
1: using F = auto (*)(A) -> B;
```

Q. What does `->` mean?

```
1: using F = auto (*)(A) -> B;
```

It means "implies".

$$A \Rightarrow B$$

Q. What does `->` mean?

```
1: using F = auto (*)(A) -> B;
```

It means "implies".

$$A \Rightarrow B$$

Function arrow is *the same thing* as Boolean implication arrow!

Let's look at the questions again.

Q. Can I obtain it?

```
1: // I have an A:
2: A a = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a B?
```

C++	Boolean interpretation
`A a;`	`A` is `true`
`auto (*f)(A) -> B;`	`A` ⇒ `B`
Can I obtain a `B`?	Is `B` `true`?

Q. Can I obtain it?

```
1: // I have an A:
2: A a = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a pair<A, B>?
```

C++	Boolean interpretation
`A a;`	`A` is `true`
`auto (*f)(A) -> B;`	`A` ⇒ `B`
Can I obtain a `pair<A, B>`?	Is `A` ∧ `B` `true`?

Q. Can I obtain it?

```
1: // I have a variant<A, B>:
2: variant<A, B> v = /* from somewhere */;
3:
4: // I have a function from A to B:
5: using F = auto (*)(A) -> B;
6: F f = /* from somewhere */;
7:
8: // can I obtain a B?
```

C++	Boolean interpretation
`variant<A, B> v;`	`A` ∨ `B` is `true`
`auto (*f)(A) -> B;`	`A` ⇒ `B`
Can I obtain a `B`?	Is `B` `true`?

How did you know?

Those questions were pretty easy, right?

How did you know the answers?

You were using your (implicit?) knowledge of implication.

Curry-Howard Isomorphism

That's what it means.

Existence *is* truth. Non-existence *is* falsehood.

Function arrow *is* implication. Product types *are* `and`. Sum types *are* `or`.

You already know how implication works,
because *you know the meaning of function types*.

We can transfer that implicit understanding to actual Boolean algebra.

So, what is implication?

It's a Boolean operator just like `and`(`∧`) and `or`(`∨`).

And it has a truth table.

A	B	A ∧ B	A ∨ B	A ⇒ B
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

Notice: `A ⇒ B` is equivalent to `¬A ∨ B`. We'll use this fact later.

What is implication?

It's "natural" to think of implication as "if A then B".
But this sort of hides the fact that `A ⇒ B` is a proposition itself.

It can be simpler to think of implication as an operator (like `and` and `or`) that produces a Boolean value.

Notice a couple of propositions that are always true:

- `X ⇒ true`
- `false ⇒ X`

Fact 1: `X ⇒ true`

A	B	A ⇒ B
0	0	1
0	1	1
1	0	0
1	1	1

Fact 2: `false ⇒ X`

A	B	A ⇒ B
0	0	1
0	1	1
1	0	0
1	1	1

`false ⇒ X` - Wait, what?

Many people find this hard to intuit. But what colour is a unicorn?
Here is a true statement:

All unicorns have pink polka-dots. 🦄

(Unicorns don't exist, so any statement about them is true.
Forget about the white ones you've seen in movies.)

(also the emoji is wrong)

Another explanation

"If it's raining, the ground is wet."

Q. What would make this a false statement?

A. Only if it's raining and the ground is *not* wet.

If it's not raining, the ground might still be wet,
and the statement would still be true.

- I have sprinklers on
- It *was* raining but not now
- A water main break
- etc

Yet another explanation

"If it's raining, the ground is wet."

or

"The set of times when it's raining is a subset of the set of times when the ground is wet."

Truth tables again

A	B	A ∧ B	A ∨ B	A ⇒ B
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

Notice: when `A ⇒ B` is `true`, `(A ∧ B) ≡ A` and `(A ∨ B) ≡ B`.

Part 5

End of aside. Now: using implication to simplify.

Simplifying by implying

`less_than<5> and less_than<7>`

We know (as humans) that:

`X < 5 ⇒ X < 7`

(is a true proposition)

If we can state this in code, we have the key to simplifying the expression.

Because we already know that:

`A ⇒ B` means that `(A ∧ B) ≡ A`

Another customization point

Let's make an `implies` function.

```
1: template <matcher X, matcher Y>
2: constexpr auto implies(X const &, Y const &) -> bool {
3:   return std::same_as<X, Y>;
4: }
```

In the general case, `X ⇒ Y` is `false`.

But also, `X ⇒ X` is `true`.

Some more fundamentals

```
 1: constexpr auto implies(matcher auto &&, always_t) -> bool {
 2:   return true;
 3: }
 4:
 5: constexpr auto implies(never_t, matcher auto &&) -> bool {
 6:   return true;
 7: }
 8:
 9: constexpr auto implies(never_t, always_t) -> bool { // disambiguate!
10:   return true;
11: }
```

These are the true propositions that we noted earlier: `X ⇒ true` and `false ⇒ X`.

Implication: `and` and `or`

We can do some Boolean algebra manipulations to find out how to write `implies` for `and` terms:

```
1:     (X ∧ Y) ⇒ A
2: ≡ ¬(X ∧ Y) ∨ A          [truth table of implication]
3: ≡ (¬X ∨ ¬Y) ∨ A         [de Morgan's law]
4: ≡ ¬X ∨ A ∨ ¬Y ∨ A       [idempotency of OR]
5: ≡ X ⇒ A ∨ Y ⇒ A         [truth table of implication]
```

And similarly (slightly more easily) for `or`.

Implication: `and` and `or`

```
1: template <matcher M, matcher L, matcher R>
2: constexpr auto implies(and_t<L, R> const &a, M const &m) -> bool {
3:   return implies(a.lhs, m) or implies(a.rhs, m);
4: }
```

```
1: template <matcher M, matcher L, matcher R>
2: constexpr auto implies(M const &m, or_t<L, R> const &o) -> bool {
3:   return implies(m, o.lhs) or implies(m, o.rhs);
4: }
```

And the code is straightforward.

Now we can express

```
1: template <typename Op, typename Field, auto X, auto Y>
2: constexpr auto implies(relational_matcher_t<Op, Field, X>,
3:                        relational_matcher_t<Op, Field, Y>) {
4:   return X == Y or Op{}(X, Y);
5: }
```

Remember the generic `relational_matcher_t`? This is saying that e.g.:

`(X < A) ⇒ (X < B) ≡ (A < B)`

(And also that `X ⇒ X` in this case.)

We can warrant more

```
1: template <typename Field, auto X, auto Y>
2: constexpr auto implies(less_equal<Field, X> const &,
3:                        less_than<Field, Y> const &) -> bool {
4:   return X < Y;
5: }
```

This is saying that:

$$(X <= A) \Rightarrow (X < B) \equiv (A < B)$$

And we can write similar warrants for implications between the other relational matchers.

Before: `simplify` for `and`

```
 1: template <matcher L, matcher R>
 2: constexpr auto simplify(and_t<L, R> const &m) {
 3:   auto l = simplify(m.lhs);
 4:   auto r = simplify(m.rhs);
 5:
 6:   if constexpr (/* l is a never_t or r is a never_t */) {
 7:     return never;
 8:   } else if constexpr (/* r is an always_t */) {
 9:     return l;
10:   } else if constexpr (/* l is an always_t */) {
11:     return r;
12:   } else {
13:     return and_t{l, r};
14:   }
15: }
```

This was what `simplify` used to look like for `and_t` terms.

Simplify, revisited (and)

```
 1: template <matcher L, matcher R>
 2: constexpr auto simplify(and_t<L, R> const &m) {
 3:   auto l = simplify(m.lhs);
 4:   auto r = simplify(m.rhs);
 5:
 6:   if constexpr (implies(l, r)) {
 7:     return l;
 8:   } else if constexpr (implies(r, l)) {
 9:     return r;
10:   } else if constexpr (implies(l, negate(r)) or implies(r, negate(l))) {
11:     return never;
12:   } else {
13:     return and_t{l, r};
14:   }
15: }
```

This is what it looks like now.

Simplify, revisited (`or`)

```
 1: template <matcher L, matcher R>
 2: constexpr auto simplify(or_t<L, R> const &m) {
 3:   auto l = simplify(m.lhs);
 4:   auto r = simplify(m.rhs);
 5:
 6:   if constexpr (implies(l, r)) {
 7:     return r;
 8:   } else if constexpr (implies(r, l)) {
 9:     return l;
10:   } else if constexpr (implies(negate(l), r) or implies(negate(r), l)) {
11:     return always;
12:   } else {
13:     return or_t{l, r};
14:   }
15: }
```

The overload for `or_t` is the dual of that for `and_t`.

Machinery sorted

This is all that's required.

- customize `negate`
- customize `implies`
- a few basic negation/implication rules
- short simplification functions for `and_t`, `or_t`, `not_t`

And all the simplifications fall out:
identities, annihilators, idempotence, absorptions, etc.

Results

We now have:

- the expressive power of a matcher DSL
- compile-time simplification of expressions
- minimal work to do at runtime
- a library (3 libraries) that compose(s)

The libraries fit together to make the job easy and fast.
(More about the indexing library is another talk.)

An important point

Although the truth table for `A ⇒ B` is equivalent to `¬A ∨ B`,
notice that *it wasn't useful to formulate implication that way*.

The whole reason for our using implication was to allow the programmer to warrant it as `true`, in order to simplify expressions.

`¬A ∨ B` is not in a form we can use this way. `A ⇒ B` *is* useful.
(Compare: code has `and`, `or`, `xor` and `not`, not just `nand`.)
See: `https://wg21.link/P2971` *Implication for C++*

Another observation

Did you notice that `A ⇒ B` is a higher-order function
when we're dealing with matchers?

What's more, it's a "real" higher-order function.
It *works directly on the functions*. It *never evaluates them*.

Most higher-order functions end up evaluating the functions they are passed, either directly, or indirectly by returning them somehow wrapped for later evaluation.

Epilogue

What else?

There are two laws we haven't dealt with yet,
because they are mostly* unused for simplification:

- de Morgan's laws
- distributive law

They are used for other transformations...
see also Luke Valenty's talk from C++Now 2024.

Remember the Github issue?

https://github.com/intel/compile-time-init-build

Issue #374

Author: lukevalenty (Luke Valenty)

Assigned to: elbeno (Ben Deane)

> Factor arbitrary matcher expressions into "sum-of-products" expressions

Disjunctive Normal Form

(a.k.a. sum of products, `or` of `and`s)

A Boolean expression is in *disjunctive normal form* when it is a disjunction (`or`) of conjunctions (`and`s) or single terms.

- there are no `or` terms inside `and` terms
- any `not`s only apply to single terms

Disjunctive Normal Form

For example:

Expression	DNF?
$(A \wedge \neg B \wedge \neg C) \vee (\neg D \wedge E \wedge F)$	✅
$(A \wedge B) \vee C$	✅
$\neg(A \vee B)$	❌
$A \wedge (B \vee (C \wedge D))$	❌

The transformation

To transform an expression into this form,
we will recursively apply two transformations where necessary:

Distributive law:

```
A ∧ (B ∨ C) -> (A ∧ B) ∨ (A ∧ C)
```

De Morgan's laws:

```
¬(A ∧ B) -> ¬A ∨ ¬B
¬(A ∨ B) -> ¬A ∧ ¬B
```

Sum of Products

```
// default transformation: the identity
template <matcher M>
[[nodiscard]] constexpr auto sum_of_products(M const &m) -> M {
  return m;
}
```

OK, so we just need to implement the transformation overloads for `and_t`, `or_t` and `not_t`.

Sum of Products: `not`

```
template <matcher M>
[[nodiscard]] constexpr auto sum_of_products(not_t<M> const &n) {
  if constexpr (/* M is an and_t */) {
    return or_t{sum_of_products(negate(n.m.lhs)),
                sum_of_products(negate(n.m.rhs))};
  } else if constexpr (/* M is an or_t */) {
    return sum_of_products(and_t{sum_of_products(negate(n.m.lhs)),
                                 sum_of_products(negate(n.m.rhs))});
  } else {
    return n;
  }
}
```

Looks like De Morgan's laws to me.

Notice the extra call to `sum_of_products` in the case where we make an `and_t`: there might still be an `or` inside it.

Sum of Products: and

```
template <matcher L, matcher R>
[[nodiscard]] constexpr auto sum_of_products(and_t<L, R> const &m) {
  auto l = sum_of_products(m.lhs);
  auto r = sum_of_products(m.rhs);

  if constexpr (/* l is an or_t */) {
    auto lr = sum_of_products(and_t{l.lhs, r});
    auto rr = sum_of_products(and_t{l.rhs, r});
    return or_t{lr, rr};
  } else if constexpr (/* r is an or_t */) {
    auto ll = sum_of_products(and_t{l, r.lhs});
    auto lr = sum_of_products(and_t{l, r.rhs});
    return or_t{ll, lr};
  } else {
    return and_t{l, r};
  }
}
```

Looks like distributive law to me.

Sum of Products: `or`

```
template <matcher L, matcher R>
[[nodiscard]] constexpr auto sum_of_products(or_t<L, R> const &m) {
  auto l = sum_of_products(m.lhs);
  auto r = sum_of_products(m.rhs);
  return or_t{l, r};
}
```

Almost nothing to do here; just recursively apply.

What does this buy us?

Disjunctive Normal Form is used to build indices.

To index on a field `F`:

- convert a callback's matcher to DNF
- walk the expression(s)
- if an `F` equality term occurs:
  - add the callback to the index
  - turn the term into `always`

DNF is the key to building indices on fields
(and removing them from the remaining matcher).

Epi-epilogue

Matcher ordering, and a new intuition for implication

Matcher ordering

What does it mean for matchers to be ordered?

`A ⇒ B` means that `A <= B`.

Intuitively, if `A < B` then

`A` is a more constrained matcher than `B`. (It matches fewer things.)

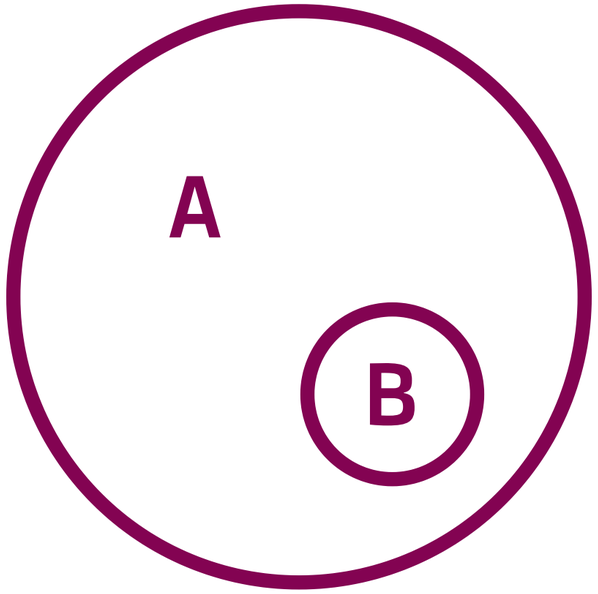
And of course, if `A ⇒ B` and `B ⇒ A` then `A` and `B` are equivalent.

Matcher ordering

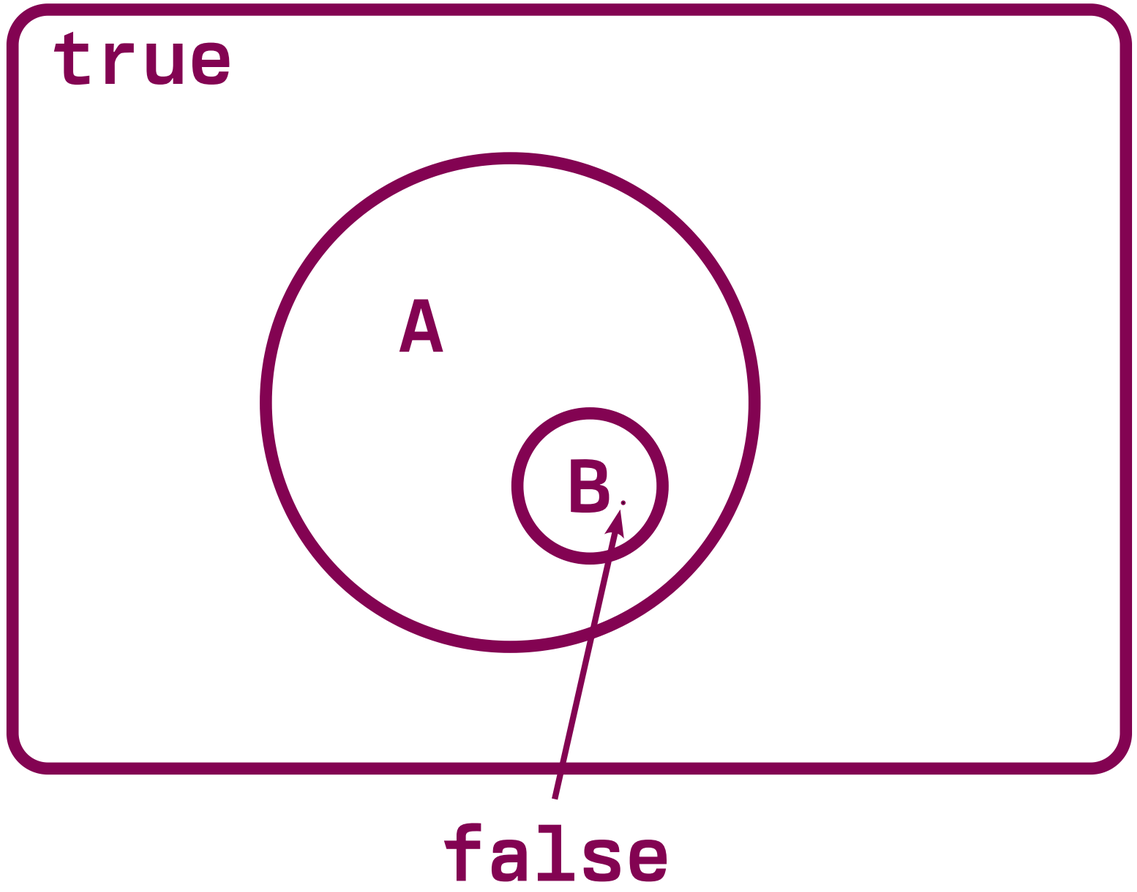
`A > B` means that:

{matched by `A`} ⊃ {matched by `B`}

alternatively, `B ⇒ A`



Ordering and implication



An intuition for why `false ⇒ X`

Matcher ordering

We can also now define a partial order
and an equivalence between matchers.

```
 1: template <matcher L, matcher R>
 2: constexpr auto operator<=>(L const &lhs, R const &rhs)
 3:     -> std::partial_ordering {
 4:   auto const l = simplify(lhs);
 5:   auto const r = simplify(rhs);
 6:   auto const x = implies(l, r);
 7:   auto const y = implies(r, l);
 8:   if (not x and not y) {
 9:     return std::partial_ordering::unordered;
10:   }
11:   return y <=> x;
12: }
```

The End

Decoupling types from storage is often
a good idea in the design of a message library.

Boolean algebra underlies everything we do.
The correspondence with type algebra is fundamental.

Implication is important, with surprising applications.
`A ⇒ B` is qualitatively different from `¬A ∨ B`.

Code: `https://github.com/intel/compile-time-init-build`