

EASY TO USE, HARD TO MISUSE

DECLARATIVE STYLE IN C++

BEN DEANE / bdeane@blizzard.com / [@ben_deane](https://twitter.com/ben_deane)

C++NOW / THURSDAY MAY 10TH, 2018

DECLARATIVE STYLE

"I keep hearing this term and every time I hear it the definition seems very hand-wavy."

– A C++Now submission reviewer

IN THIS TALK

1. Definitions & motivation
2. Where we came from
3. Where we are
4. Where we could be headed

WHAT DO WE MEAN?

WIKIPEDIA

"A programming paradigm ... that expresses the logic of a computation without describing its control flow."

WIKI.C2.COM

"Often it involves the separation of 'facts' from operations on the facts."

"... generalizes the pure functional model."

- **commutativity**
- **idempotency**

LANGUAGE CLASSIFICATIONS?

- imperative/procedural (FORTRAN, C)
- object-oriented (Smalltalk, Java)
- functional (ML, Haskell)
- etc.

COMPLEXITY AND LINE COUNT

Fewer lines of code => fewer bugs?

<insert hand-wave here?>

DECLARATIVE STYLE MOTIVATION

"Declarative programming is everyone's dream, because it looks like it's easier to prove correct."

– Another C++Now submission reviewer

DECLARATIVE STYLE INDICATORS

- referential transparency
- say WHAT in preference to HOW
- minimize imperative style
- declaring things
- expressions over statements

EXPRESSIONS VS STATEMENTS

EXPRESSIONS

"An expression is a sequence of operators and operands that specifies a computation.
An expression can result in a value and can cause side effects." [expr.pre] § 1

Properties of expressions:

- value category
- type

EXPRESSIONS COMPOSE ON MULTIPLE AXES

```
auto expr = a @ b @ c;
```

Consider this snippet.

STATEMENTS

"Except as indicated, statements are executed in sequence." [stmt.stmt] § 1

Properties of statements:

- er...

STATEMENTS "COMPOSE" ONLY BY SEQUENCING

```
x;  
y;  
z;
```

- no type checking
- value checking is manual, intrusive
- implicit constraints
- temporal reasoning is poor

IMPERATIVE SAFETY GEAR

Many of our guidelines, best practices, idioms, and much of our tooling, analysis, and brainpower work in service of checking the implicit constraints around statement "composition".

DECLARATIVE STYLE: AVOID STATEMENTS!

- expression statement
- selection statement (`if`, `switch`)
- iteration statement (`for`, `while`, `do`)
- jump statement (`break`, `continue`, `return`, `goto`)
- declaration statement

LET'S EXAMINE HISTORY...

Let's look at where we've come from, and see how it informs moving to declarative style.

WORLD'S LAST BUG

```
while (true)
{
    status = GetRadarInfo();
    if (status = 1)
        LaunchMissiles();
}
```

Ancient history you say?

ODD THING #1: ASSIGNMENTS ARE EXPRESSIONS

Assignment as an expression is a historical choice.

It's doing us no favours today.

Assignment should be a statement.

ODD THING #1: ASSIGNMENTS ARE EXPRESSIONS

"Expression is not a statement."

– Modula-3 compiler, 1993

And quite right, too.

ODD THING #1: ASSIGNMENTS ARE EXPRESSIONS

```
/* The following function will print a non-negative number, n, to
   the base b, where 2<=b<=10. This routine uses the fact that
   in the ASCII character set, the digits 0 to 9 have sequential
   code values. */
printn(n, b) {
    extrn putchar;
    auto a;

    if (a = n / b)
        printn(a, b); /* recursive */
    putchar(n % b + '0');
}
```

ODD THING #1: ASSIGNMENTS ARE EXPRESSIONS

We've learned to deal with this. But we don't really like it.

- yoda conditions
- compiler warnings
- P0963: discouraged

ODD THING #2: = MEANS ASSIGNMENT

```
/* The following function will print a non-negative number, n, to
   the base b, where 2<=b<=10. This routine uses the fact that
   in the ASCII character set, the digits 0 to 9 have sequential
   code values. */
printn(n, b) {
    extrn putchar;
    auto a;

    if (a = n / b)    /* assignment, not test for equality */
        printn(a, b); /* recursive */
    putchar(n % b + '0');
}
```


ODD THING #2: = MEANS ASSIGNMENT

"A notorious example for a bad idea was the choice of the equal sign to denote assignment."

– Niklaus Wirth

ODD THING #2: = MEANS ASSIGNMENT

- Superplan (1951) introduced = for assignment
- FORTRAN (1957) used = (because .GT. .LT. .EQ. etc)
- ALGOL-58 introduced := (assignment) distinct from = (equality)
 - Subsequently many languages went this way
- BCPL (1967) used :=
- B (1969) simplified a lot of BCPL syntax, went with =
 - Followed by C (1972) and many other languages

ODD THING #2: = MEANS ASSIGNMENT

"Since assignment is about twice as frequent as equality testing in typical programs, it's appropriate that the operator be half as long."

– Ken Thompson

DECLARATION VS (RE-)ASSIGNMENT

In moving from BCPL to B, the distinction between declaration and reassignment was blurred.

```
int a = 42; // declaration/initialization  
  
a = 1729; // assignment
```






*"It cannot be overemphasized that **assignment and initialization are different operations.**"*

– Bjarne Stroustrup, The C++ Programming Language

<END OF HISTORICAL DIVERSION>

- Declaring things is – has always been – fine.
- Declaration and assignment are different things that look the same.
- Assignment as an expression statement is best avoided.
 - Chained assignments are a syntactic laziness.

DECLARATIVE STYLE: AVOIDING STATEMENTS

Statement	Status
assignment	
selection	
iteration	
jump	
declaration	

A QUICK DECLARATIVE STUDY

EXAMPLE

Given:

```
weak_ptr<Foo> wp;
```

How to write:

```
Bar b;  
{  
    auto sp = wp.lock();  
    if (sp) b = sp->bar();  
}
```

In a (more) declarative way.

C++17 IF-INITIALIZER?

```
Bar b;  
if (auto sp = wp.lock(); sp)  
    b = sp->bar();
```

This still has the declaration/initialization split. Still has mutable state.

CONDITIONAL OPERATOR?

```
Bar b = wp.lock() ? wp.lock()->bar() : Bar{};
```

Hm...

C++?? CONDITIONAL-OPERATOR-INITIALIZER?

```
// this isn't real syntax...  
Bar b = [auto sp = wp.lock(); sp] ? sp->bar() : Bar{};
```

Might be nice... but not today.

GCC EXTENSION?

```
Bar b =  
(  
    auto sp = wp.lock();  
    sp ? sp->bar() : Bar{};  
));
```

Not ISO C++.

I+LE?

```
Bar b = [&] () {  
    if (auto sp = wp.lock(); sp) return sp->bar();  
    return Bar{};  
}();
```

Immediately-invoked, inline, initializing, ...

OPTIONAL-LIKE?

```
Bar b = get_bar_or(wp.lock(), Bar{});
```

Not really generic enough.

FUNCTORIAL/MONADIC INTERFACE?

```
shared_ptr<Bar> b = fmap(wp.lock(),  
                        [] (auto foo) { return foo.bar(); });
```

```
template <typename T, typename F>  
[[nodiscard]] auto fmap(const shared_ptr<T>& p, F f)  
    -> shared_ptr<invoke_result_t<F, T>>  
{  
    ...  
}
```

STUDY CONCLUSIONS

"Total" declarative style is not always achievable in C++.

A more declarative style is a reasonable goal.

Some features of C++ help us get there.

Different domains lean towards different approaches.

EXISTING DECLARATIVE PRACTICE

We are surrounded by guidelines, goals and idioms.
Looking through a declarative lens, we can tie it together.

CORE GUIDELINES

Con. 1 By default, make objects immutable.






Con. 4 Use `const` to define objects with values that do not change after construction.

ES. 21 Don't introduce a variable (or constant) before you need to use it.

ES. 22 Don't declare a variable until you have a value to initialize it with.

ES. 28 Use lambdas for complex initialization.

DECLARATIVE STYLE: AVOIDING STATEMENTS

Statement	Status	Killed by
assignment		guidelines
selection		
iteration		
jump		
declaration		

FUNCTIONS IN GENERAL

Which is better?

```
// do A  
...  
// do B  
...  
// do C  
...
```

or

```
do_A();  
do_B();  
do_C();
```

?

THE "NORMAL" REASONS

- shorter is more expressive, understandable
- encapsulation of variable scopes, lifetimes, concerns
- functions give things names

ANOTHER REASON

Functions turn statements into expressions.

- `return` is the socially acceptable `goto`
- way better than `break`
- and if that wasn't enough, RVO

<ALGORITHM>

"No Raw Loops"

What does that mean?

- encapsulate iteration statements
- encapsulate remaining assignments
- encapsulate `break` and `continue`

```
#include "my_algorithms.h"
```

- min_unused
- is_prefix_of
- join
- transform_if
- set_differences (aka before and after)
- push_back_unique

DECLARATIVE STYLE: AVOIDING STATEMENTS

Statement	Status	Killed by
assignment	💀	guidelines
selection	❤️	
iteration	💀	"no raw loops"
jump	💀	"no raw loops"
declaration	❤️	

DECLARATIVE DOMAINS AND PATTERNS

TESTING

```
TEST_CASE( "Factorials are computed", "[factorial]" ) {  
    REQUIRE( Factorial(1) == 1 );  
    REQUIRE( Factorial(2) == 2 );  
    REQUIRE( Factorial(3) == 6 );  
}
```

Conditions are encapsulated; nothing is dependent.

- idempotent
- minimal temporal dependency between statements
- leverage constructors/RAII
- popularity of sections over fixture management

LOGGING : IMPERATIVE TURNED DECLARATIVE

```
fprintf(g_debugLogFile, "R Tape loading error, %d:%d", line, stmt);
```

VS

```
LOG("R Tape loading error, " << line << ':' << stmt);
```

WHERE DID THE GLOBAL GO?

```
// g_debugLogFile is a global variable  
fprintf(g_debugLogFile, "R Tape loading error, %d:%d", line, stmt);
```

```
// somewhere, a "global" variable lurks? where does the log go to?  
LOG("R Tape loading error, " << line << ':' << stmt);
```

C-STYLE LOG SINK

```
fprintf(g_debugLogFilep, "R Tape loading error, %d:%d", line, stmt);
```

What would we do if we wanted to change where the log went?

LOG SINKS: OO TURNED DECLARATIVE

A study in compositional design.

```
class Sink
{
    ...
    virtual bool Push(const Entry& e);
    ...
};
```

SINK VARIATIONS

```
class FileSink : Sink
{
    ...
    FileSink(string_view pathname);
    ...
};

class DebugSink : Sink { ... };
```


SINK VARIATIONS

```
class FilterSink : Sink
{
    ...
    template <typename Pred>
    FilterSink(Pred p);
    ...
    using Predicate = std::function<bool(const Entry&)>;
    Predicate pred;
};
```

SINK VARIATIONS

```
// Exercise for the reader: ExecutionPolicy Concept  
template <typename ExecutionPolicy>  
class ExecSink : Sink { ... };
```

SINK VARIATIONS

```
class MultiSink : Sink
{
    ...
    vector<unique_ptr<Sink>> sinks;
};
```

SINK VARIATIONS

```
class NullSink : Sink
{
    ...
    virtual bool Push(const Entry&) override { return true; }
    ...
};
```

DECLARATIVE SINK CONSTRUCTION

```
auto fileSink = [&] () -> std::unique_ptr<Sink> {  
    if (logToFile) {  
        return std::make_unique<FileSink>(generate_filename());  
    } else {  
        return std::make_unique<NullSink>();  
    }  
}();
```

Conditions are encapsulated at the point of construction.

The point of use is condition-free and declarative.

DECLARATIVE STYLE: AVOIDING STATEMENTS

Statement	Status	Killed by
assignment	💀	guidelines
selection	💀	paradigm shift
iteration	💀	"no raw loops"
jump	💀	"no raw loops"
declaration	❤️	

DESIGN PATTERNS

OO PATTERNS

Several patterns lean towards declarative style.

Many patterns are about replacing conditions with polymorphism.

- Null object
- Command
- Composite

THE "BUILDER PATTERN"

AKA "Fluent Style" (not the original GoF pattern)

```
FluentGlutApp(argc, argv)
    .withDoubleBuffer().withRGBA().withAlpha().withDepth()
    .at(200, 200).across(500, 500)
    .named("My OpenGL/GLUT App")
    .create();
```

"In which the author turns what should be 5 lines of glut calls at the start of `main` into 100 lines of buggy OOP."

– Nicolas Guillemot (via Twitter)

BUILDER PATTERN: A BETTER EXAMPLE

```
// Schedule& Schedule::then(interval_t);

auto s = Schedule(interval::fixed{1s})
    .then(repeat::n_times{5, interval::random_exponential{2s, 2.0}})
    .then(repeat::forever{interval::fixed{30s}});

// template <typename Timer, typename Task>
// void Schedule::run(Timer, Task);
s.run(timer, task);
```

BUILDER PATTERN: HELP FROM C++17

P0145: Refining Expression Evaluation Order for Idiomatic C++

```
void f()
{
    std::string s = "but I have heard it works even if you don't believe in it";
    s.replace(0, 4, "")
      .replace(s.find("even"), 4, "only")
      .replace(s.find(" don't"), 6, "");
    assert(s == "I have heard it works only if you believe in it");
}
```

PUTTING TYPES TO WORK

This "builder pattern" is an ideal place to put strong types to work.

```
// Build a request object
request_t req = make_request()
    .set_req_field_1(...)
    .set_req_field_2(...)
    .set_opt_field(...)
    .set_opt_field(...)
    .set_opt_field(...);

// Use it
send_request(req);
```

PUTTING TYPES TO WORK

The "normal" construct for this behaviour.

```
struct request_t {  
    request_t& set_req_field_1(field_t f) {  
        f1 = f;  
        return *this;  
    }  
    request_t& set_req_field_2(field_t f);  
    request_t& set_opt_field(field_t f);  
  
    field_t f1;  
    // etc ...  
};  
  
request_t make_request() { ... }
```

BEHAVIOUR IN THE TYPE

One way: use a bitfield.

```
constexpr static uint8_t OPT_FIELDS = 1 << 0;  
constexpr static uint8_t REQ_FIELD1 = 1 << 1;  
constexpr static uint8_t REQ_FIELD2 = 1 << 2;  
constexpr static uint8_t ALL_FIELDS = OPT_FIELDS | REQ_FIELD1 | REQ_FIELD2;
```

BEHAVIOUR IN THE TYPE

```
template <uint8_t N>
struct request_t;

template <>
struct request_t<0>
{
    field_t f1;
    // etc ...
};

template <uint8_t N>
struct request_t : request_t<N-1>
{
    request_t<N & ~REQ_FIELD1>& set_req_field1(field_t f) {
        this->f1 = f;
        return *this;
    }
    request_t<N & ~REQ_FIELD2>& set_req_field2(field_t f);
    request_t& set_opt_field(field_t f);
};
```

BEHAVIOUR IN THE TYPE

Use `=delete` to enable the `send_request` function only for a correctly-filled-in request.

```
request_t<ALL_FIELDS> make_request();  
  
template <uint8_t N>  
void send_request(const request_t<N>& req) = delete;  
  
void send_request(const request_t<OPT_FIELDS>& req);
```


BUILDER PATTERN GUIDELINES

Fluent style is more suitable when:

- you have a single verb (`then`, `set_field`)
- you'll be building objects a lot
- you can make types work for you
- rvalues aren't too verbose

RANGES

Let's talk about ranges a little.

RANGES: EXAMPLE 0

```
dates_in_year(2015)    // 0. Make a range of dates.
| by_month()           // 1. Group the dates by month.
| layout_months()      // 2. Format the month into a range of
                        // strings.
| chunk(3)             // 3. Group the months that belong
                        // side-by-side.
| transpose_months()   // 4. Transpose the rows and columns
                        // of the side-by-side months.
| view::join           // 5. Ungroup the side-by-side months.
| join_months()        // 6. Join the string of the transposed
                        // months.
```

RANGES: EXAMPLE N

```
std::mt19937 gen(std::random_device{}());
auto rsvps = rsvp_json // json is a valid range
| view::remove_if([](auto&& elem) {
    return "yes" != elem.at("response"); }) // filter out non-"yes" RSVP responses
| view::transform([](auto&& elem) {
    return elem["member"]["name"].dump(); }) // keep name as string
| ranges::to_vector // convert lazy range to vector
| action::shuffle(gen); // random shuffle vector elements
```

From <https://github.com/CoreCppIL/raffle>

RANGES: READABILITY IS FAMILIARITY

What does this do?

```
+\\l10
```

RANGES: READABILITY IS FAMILIARITY

```
int arr[] = {1,2,3,4,5,6,7,8,9,10};  
int sum = 0;  
for (int i = 0; i < 10; ++i)  
{  
    sum += arr[i];  
    arr[i] = sum;  
}
```

RANGES: READABILITY IS FAMILIARITY

```
std::array<int, 10> input;  
std::iota(input.begin(), input.end(), 1);  
std::partial_sum(input.begin(), input.end(), input.begin());
```

RANGES: READABILITY IS FAMILIARITY

```
+\\l10
```

```
auto r = view::iota(1)  
      | view::take(10)  
      | view::partial_sum(std::plus<>{});
```


READABLE & ROBUST

Code that says WHAT is just as readable as code that says HOW.

We are used to seeing code that says HOW. It's more familiar.

Code that says WHAT is more likely to remain robust.

"WHOLEMEAL PROGRAMMING"

Declarative style is about processing data pipelines.

When you have composable pieces, rearranging and exploring data is quick and easy.

Compare: unix command-line.

- generators (find, `iota`)
- selections (grep, `unique`)
- transformations (cut, tr, `transform`)
- permutations (sort, `shuffle`)
- reductions/unfolds (wc, xargs, `accumulate`)

OPERATOR OVERLOADING

Good or bad?

Answer: good. When principled.

EXPRESSIVE USER-DEFINED TYPES

Regular types are great!

Operators give us compositional style with concision.

```
// which would you rather see?  
  
// option 1  
a = operator+(x, operator*(y, z));  
  
// option 2  
a = x + y * z;
```

COMMAND-LINE PARSING: CLARA

Phil Nash: *A Composable Command Line Parser*
(CppCon 2017 Lightning Talks)

```
auto cli
= ExeName( config.processName )
| Help( config.showHelp )
| Opt( config.listTests )
  [ "-l" ][ "--list-tests" ]
  ( "list all/matching test cases" )
| Opt( config.listTags )
  [ "-t" ][ "--list-tags" ]
  ( "list all/matching tags" )
...
```

<https://www.youtube.com/watch?v=Od4bjLfwI-A>

OPERATOR OVERLOADING ADVICE

"When in doubt, do as the `ints` do."

– Scott Meyers, More Effective C++

"It is probably wise to use operator overloading primarily to mimic conventional use of operators."

– Bjarne Stroustrup, The C++ Programming Language

OPERATOR OVERLOADING

When in doubt, do what `operator+` does?

operator+ PROPERTIES

Property	Math(s)	C++
Closed	✓	✗ (overflow)
Associative	✓	✗ (floating point)
Commutative	✓	✗ (strings)
Has Identity	✓	✓ ✓ (+0.0, -0.0!)

C++ OPERATOR CONVENTIONS

Operator(s)	Convention
<code>== !=</code>	Math(s)-like
<code>< > <= >= <=></code>	Math(s)-like
<code>+ - * /</code>	<i>Mostly</i> math(s)-like
<code> </code>	Pipelining, monoidal
<code>->*</code>	Expression templates
<code>&& ,</code>	Just don't
other	Open for abuse?

OPERATORS IN COMPILER HISTORY

(from https://jeffreykegler.github.io/personal/timeline_v3)

1956: The IT Compiler

"...the first really useful compiler."

– Donald E Knuth

But it didn't have operator precedence as we know it today.

"The lack of operator priority ... in the IT language was the most frequent single cause of errors by the users of that compiler."

– Donald E Knuth

OPERATOR OVERLOADING LEARNINGS

- operators communicate properties
- operators make sense for binary functions
- operators should be conventional
- identify your monoids!

WHERE CAN WE GO FROM HERE?

Where is C++ giving declarative code good support?

Where can it be improved?

WHERE C++ IS STRONG

RAII, INITIALIZATION

RAII is the bread-and-butter of C++ programming. It's a natural fit for a declarative style.

Initialization is complex, but getting easier.

- aggregate initialization
- rule of zero
- UDLs for extra expressiveness
- class template deduction
- C++20 designators

FUNCTIONS & LAMBDAS

Functions:

- turn statements into expressions
- give expressions names
- encapsulate conditions
- are the optimizer's bread and butter

Structured bindings work around single-return-value limitation.

OVERLOADS & TEMPLATES

Parametric polymorphism: enable use of functions without conditionals.

Let the compiler do the right thing.

```
template <typename A, typename B = A,  
         typename C = std::common_type_t<A, B>,  
         typename D = std::uniform_int_distribution<C>>  
inline auto make_uniform_distribution(const A& a,  
                                     const B& b = std::numeric_limits<B>::max())  
    -> std::enable_if_t<std::is_integral_v<C>, D>  
{  
    return D(a, b);  
}
```

Andy Bond: *AAAARGH!?* (CppCon 2016)

<https://www.youtube.com/watch?v=ZCGyvPDM0YY>

OVERLOADS & TEMPLATES

```
template <typename A, typename B = A,  
          typename C = std::common_type_t<A, B>,  
          typename D = std::uniform_real_distribution<C>>  
inline auto make_uniform_distribution(const A& a,  
                                     const B& b = B{1})  
    -> std::enable_if_t<std::is_floating_point_v<C>, D>;  
  
class uniform_duration_distribution;  
  
template <typename A, typename B = A,  
          typename C = std::common_type_t<A, B>,  
          typename D = uniform_duration_distribution<C>>  
inline auto make_uniform_distribution(const A& a,  
                                     const B& b = B::max()) -> D;
```

OTHER FEATURES

- Guaranteed copy elision P0135
- Evaluation order guarantees P0145
- Fold expressions

WHERE C++ IS WEAKER

AKA: write a paper!

C++Now is a good place to start...

INCONSISTENCIES

In C++17, we gained `if`- and `switch`-initializers.

```
if (auto [it, inserted] = m.emplace("Jenny", 8675309); inserted)
{
    ...
}
```

But no love for the expression equivalent of `if`.

```
auto result =
    (auto [it, inserted] = m.emplace("Jenny", 8675309); inserted)
    ? // some expression ...
    : // some other expression ...
```

HERITAGE: ASSIGNMENT

Assignment is an expression.

- implementation burden: lvalues
- `operator=` must be a member function
- but `operator@=` can be free?
- chained assignments? convenient but a smell

Assignment is blurred with construction.

- historic: rule of N
- conflicting sink parameter advice
- now it's worse: move vs copy, reference qualifiers

TYPE SYSTEM: "FUNCTIONS"?

```
int steps = 0;
auto f = [&](int x) { ++steps; return x / 2; };
auto g = [&](int x) { ++steps; return 3 * x + 1; };

// why doesn't this work?
auto h = (x % 2 == 0) ? f : g;
```

HERITAGE: OPERATORS

C++ inherits pretty much all of its operators from C.

We also inherit some fixed semantics (despite operator overloading).

Operators can be amazing for expressivity of code and declarative constructs.

HERITAGE: OPERATORS

Operators are hard to deal with in C++.

- fixed syntactic set
- fixed precedence
- fixed associativity
- fixed arity
- fixed fixity
- fixed evaluation semantics (which may change on overload)
- ADL

OPERATOR OVERLOADING AND FUTURES



```
// imaginary-ish code  
my_future<A> f(X);  
my_future<B> g1(A);  
my_future<C> g2(A);  
my_future<D> h(B, C);
```

OPERATOR OVERLOADING AND FUTURES

```
auto fut = f();  
auto split1 = fut.then(g1);  
auto split2 = fut.then(g2);  
auto fut2 = when_all(split1, split2).then(h);
```

```
auto fut = f() > (g1 & g2) > h;
```

Operator overloading can clarify the computational structure when combining futures/promises.

A CALL TO ACTION

What convention are we going to adopt for monadic operators?

The future for `future` operators is uncertain.

Please, let's not abuse more operators like we did with `>>` and `<<` for streams.

WHERE C++ IS GETTING BETTER

"IMPERATIVE SAFETY GEAR"

- better warnings
- static analysis
- `[[nodiscard]]` attribute (another default?)
- `[[fallthrough]]` attribute
- `if`-initializer

HERITAGE: DECLARATION SYNTAX

Something we're too close to to appreciate how painful it is?

```
int (*daytab)[13];  
int *daytab[13];  
  
char ((*x[3])())[5];
```

K&R: *5.12 Complicated Declarations*

"C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions."

"...because declarations cannot be read left-to-right, and because parentheses are over-used."

DECLARATION SYNTAX HELP

Prefer `using` over `typedef`.

Prefer trailing return syntax in aliases.

Think of `auto(*)` as a token that means "pointer-to-function".

```
typedef int (*FP)(float, string);  
  
using FP = auto(*) (float, string) -> int;
```

RICHNESS OF LIBRARY HELP

Seemingly-unimportant helper functions (or metafunctions) can be very important in avoiding conditionals.

- `std::exchange`
- `std::as_const`
- `std::apply`
- expanding `type_traits`
- monadic interface to `std::optional`

RICHNESS OF LIBRARY HELP

```
template <typename T>
decltype(auto) identity(T&& t) {
    return std::forward<T>(t);
}
```

```
template <typename T>
auto always(T&& t) {
    return [x = std::forward<T>(t)](auto...) { return x; };
};
```

GUIDELINES FOR DECLARATIVE CODE

Meta-guideline *reductio*: avoid writing statements.
(Principally control-flow and assignment.)

REPLACING CONDITIONALS

Style	Signature Element	Elimination Strategy
Imperative	Statement	multi-computation
Object-Oriented	Object construction	polymorphism
Functional	Function call	higher order function
Generic	Type instantiation	traits class

The Conditional-Replacement Meta-Pattern.

REPLACING CONDITIONALS

- Push conditionals down the callstack
 - intrinsic to data structures
 - optional/monadic interface
 - handle at leaf, don't leak
- Push conditionals up the callstack
 - dependency injection
 - higher-order functions
 - power to the caller
 - lifted to root, abstracted
- Goal: total functions

REPLACING CONDITIONALS => FEWER STATEMENTS

When you replace/encapsulate conditionals:

- less call-site logic (obviously)
- simpler, total functions
- simpler loops (no break/continue without conditions)
- more reason-ability

REPLACING LOOPS => FEWER STATEMENTS

No Raw Loops: encapsulate and replace iteration and jumps

- less call-site logic
- simpler, total functions
- more reason-ability
- vocabulary grows

REPLACING ASSIGNMENTS

- Declare-at-use
 - use l+LEs
 - leverage `const`
 - use AAA-style if you like
- Overload operators for declaration power

LET THE LANGUAGE HELP

Where you can't avoid statements, use "imperative safety gear"

- `nodiscard` attribute
- `if`-initializer
- static analysis

DECLARATIVE INTERFACES

- dependency injection
- higher-order functions
- builder pattern / fluent style
- identify monoids
- start with composition

DECLARATIVE GOALS

Expressions over statements.

Declarations over assignments.

Unconditional code.