C++ONLINE

ANKUR SATLE

TALK:

DECLARATIVE C++

2025

“

The essence of a program is
to create a specification
of the system

”

Ankur Satle

Desirable Qualities of Code?

Desirable Software Qualities

Functional

Correct

Complete

As per specifications

Consistent

Safe

Verifiable

Tested

Lawful

Portable

Interoperable

Non-functional

Efficient

High-performance

Scalable

Safe

Debuggable

Monitorable

Predictable

Responsive

Reliable

Available

Self-healing

Usable

Long-term

Maintainable

Extendible

Reusable

Flexible

Simple

Intention-revealing

Obvious

Modular

Non-repetitive

“PIT OF SUCCESS”


Declarative
C++

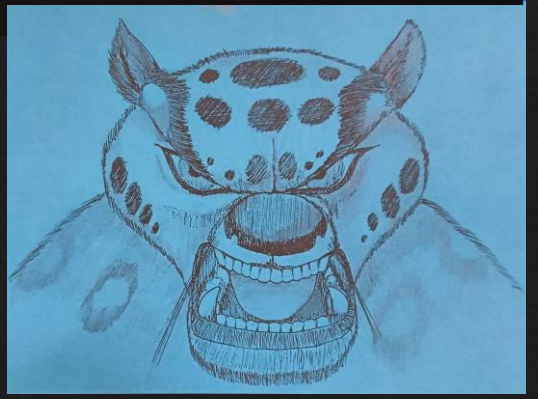

Declarative
Correct
Reusable
Composable
Simple
Complete
Extendible
Safe
Obvious
Readable
Intention-revealing
Resource Managed
Error Handled
Maintainable
Non-repetitive
Reliable

Me!

- Indian National Body at WG21
- Architect: High-perf Cloud-Native Products & Libraries
- Company Events, Conferences & CppIndia
- Interests: CI, DevOps, Lean, Automation
- Product & Team Transformations
  - Faster Teams with time

- Learning (natural) languages & cultures
- Art with children
- Treks

- https://ankursatle.wordpress.com/professional
- https://www.linkedin.com/in/ankursatle



A style of building the structure and elements of programs
that express the logic of computation
without describing its control flow

Wikipedia

“

Declarative is when you declare it
but don’t use it

”

Imperative vs Declarative

Imperative

- How
- Commands/statements
- Control Structures
- Procedural, OO
- Reference Semantics
- Recipes, Algorithms, State, Conditions, Side-effects, Sharing,
- Tests prove correctness

Declarative

- What
- Expressions
- Pipelines
- Constraints, Functional
- Value Semantics
- Composition, Pure Functions, Lambda closures, Higher-order functions, Lazy evaluations, Monads
- Correctness by design, trivial to test

P.3 Express Intent

P.3: Express intent

Reason

Unless the intent of some code is stated (e.g., in names or comments), it is impossible to tell whether the code does what it is supposed to do.

Note

Alternative formulation: Say what should be done, rather than just how it should be done.

Note

Some language constructs express intent better than others.

- Intention-revealing implementations
- Express ideas directly in code

SQL

```sql
SELECT country, sum(customer_count) total_customers, sum(spend) ad_spend
FROM (
    SELECT customers.ip_country, count(email) customer_count
    FROM customers
    WHERE customers.createdate > '2020-01-01'
    GROUP BY country) new_customers
JOIN facebook_ads ON facebook_ads.country = new_customers.ip_country
WHERE ad_spend.date > '2020-01-01'
GROUP BY country
ORDER BY ad_spend desc;
```


Source: https://trevorfox.com/2020/01/getting-started-with-sql-for-marketing-with-facebook-ads-example/

Network IPv4 Header

IPv4 header format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total length															
4	32	Identification																Flags			Fragment offset											
8	64	Time to Live								Protocol								Header checksum															
12	96	Source address																															
16	128	Destination address																															
20	160	Options (if IHL > 5)																															
⋮	⋮																																
56	448																																

Source: https://en.wikipedia.org/wiki/IPv4

ABNF for SIP Messages

see ABNF in RFC 3261

SIP-message =

Request / Response

Request =

Request-Line *(message-header) CRLF [message-body]

; see an example Here

Response =

Status-Line *(message-header) CRLF [message-body]

; see an example Here

Request-Line =

Method SP Request-URI SP SIP-Version CRLF

Status-Line =

SIP-Version SP Status-Code SP Reason-Phrase CRLF

Method =

INVITEm / ACKm / OPTIONSm / BYEm / CANCELm / REGISTERm / INFOm / PRACKm /

SUBSCRIBEm / NOTIFYm / UPDATEm / MESSAGEm / REFERm / PUBLISHm / extension-method

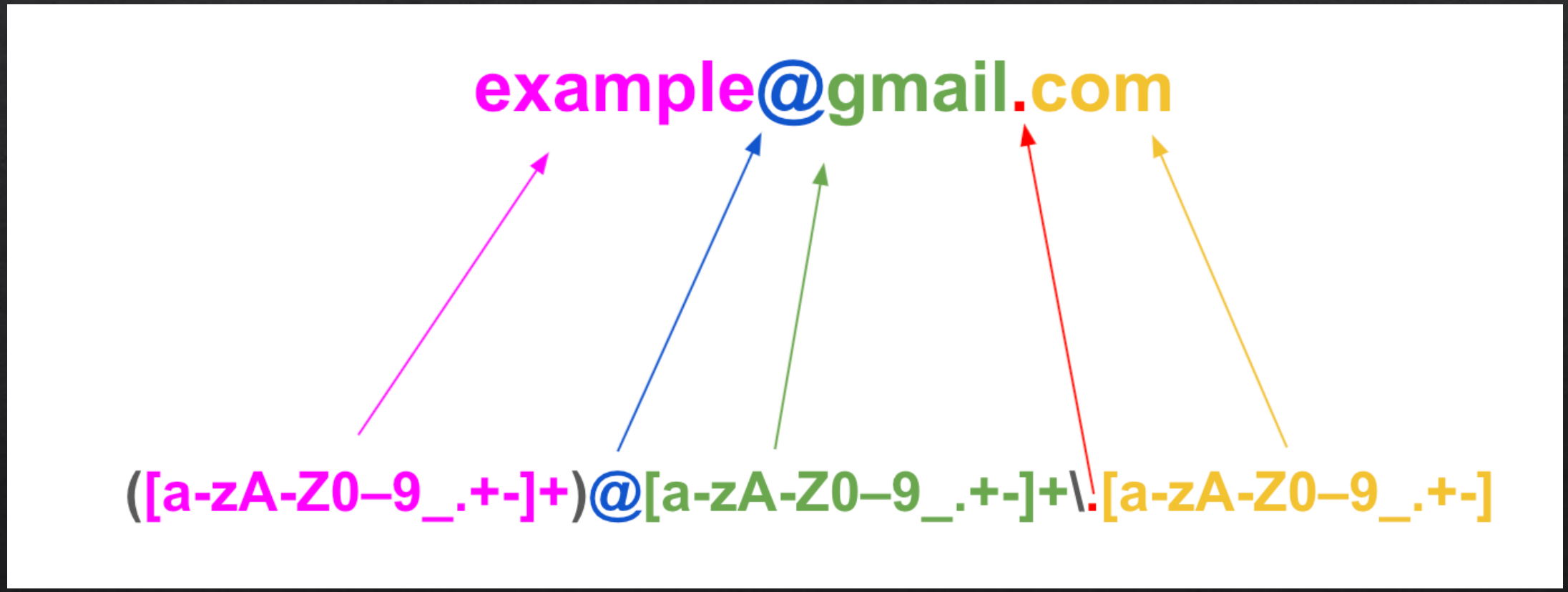
---

ABNF Message Definition

- SIP – Message format
- Succinct & very clear

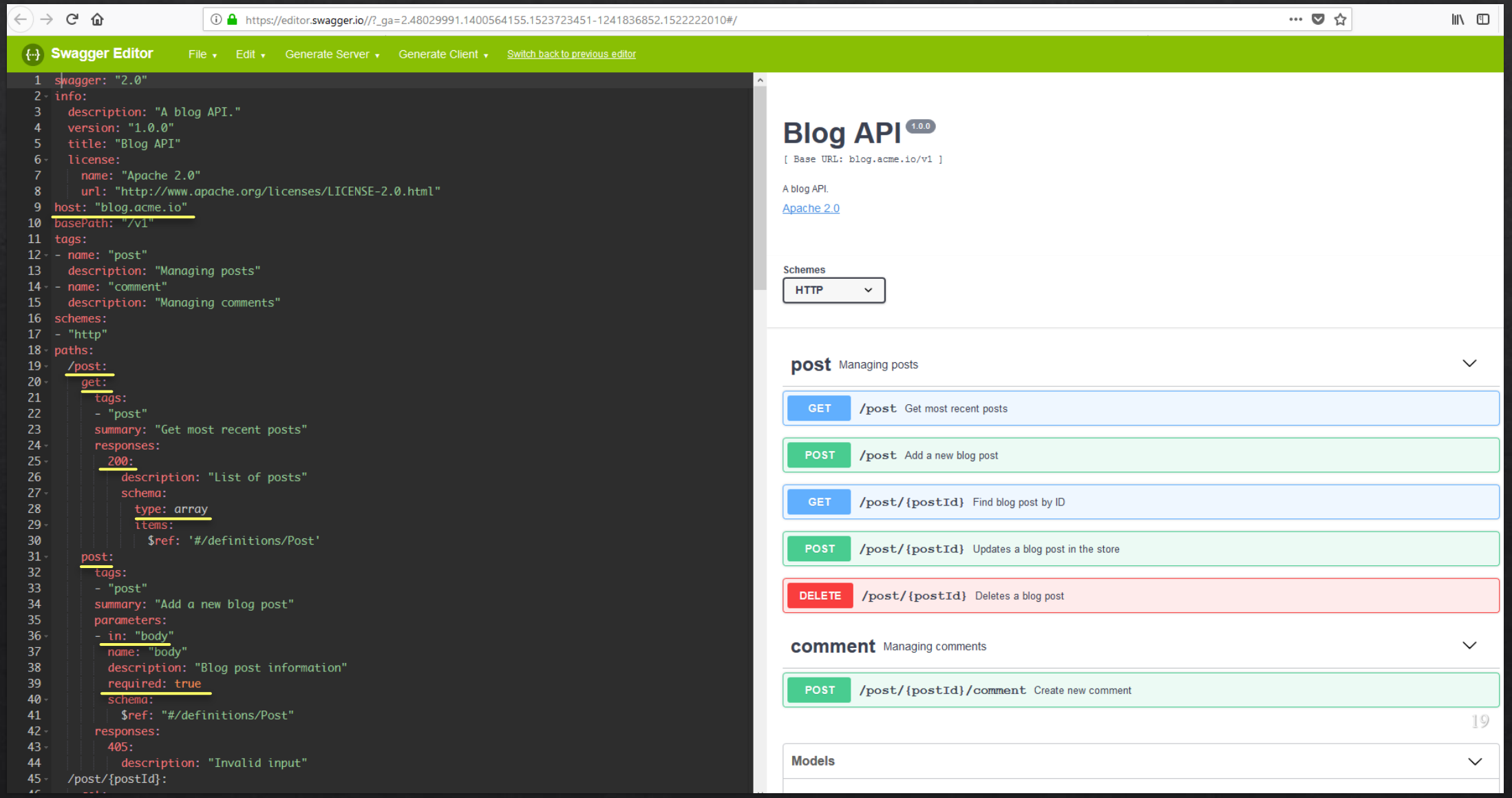
Source: https://www.tech-invite.com/fo-abnf/tinv-fo-abnf-sip.html

Regex



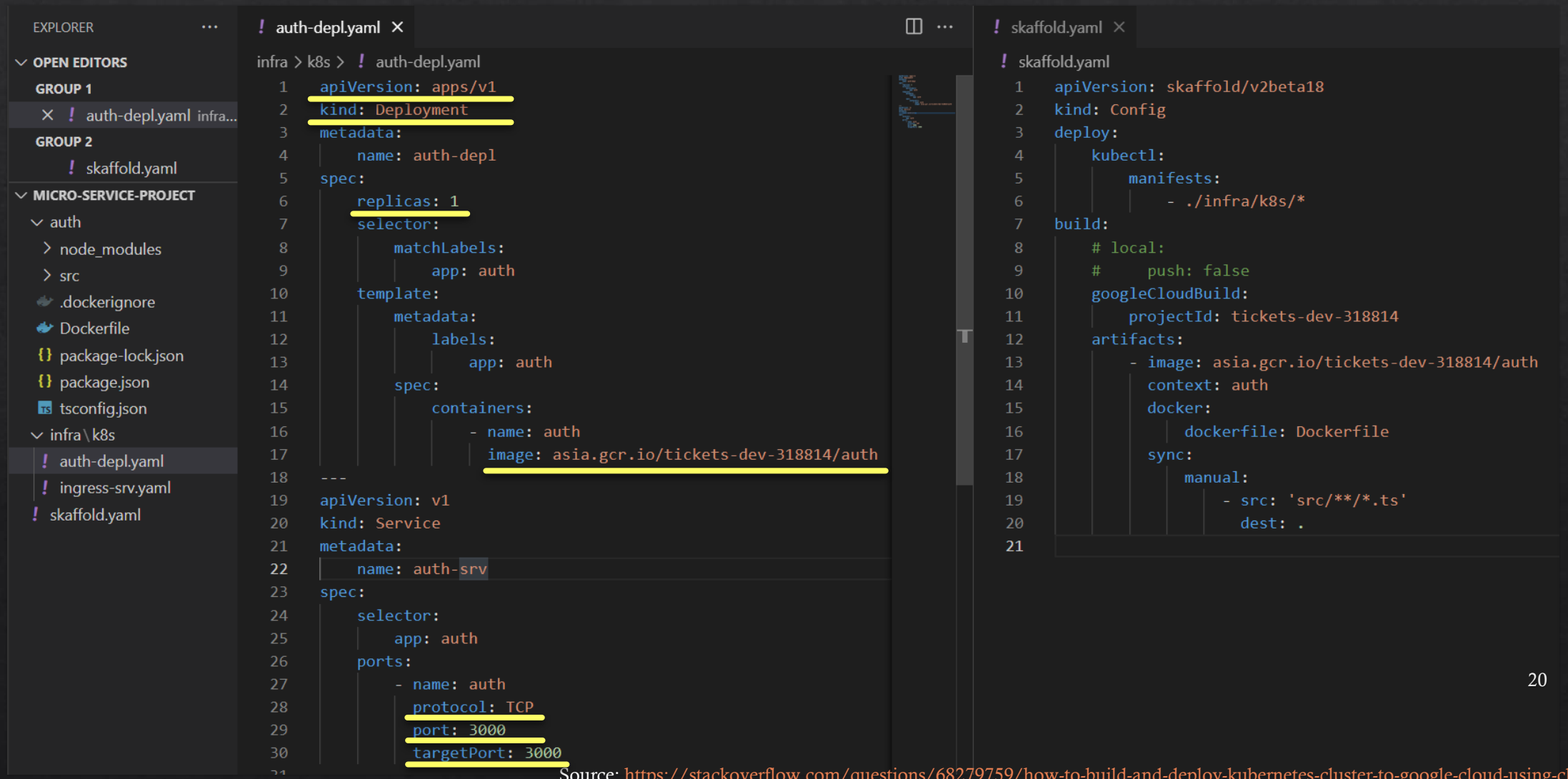
Source: https://towardsdatascience.com/easiest-way-to-remember-regular-expressions-regex-178ba518bebd

REST API Definition



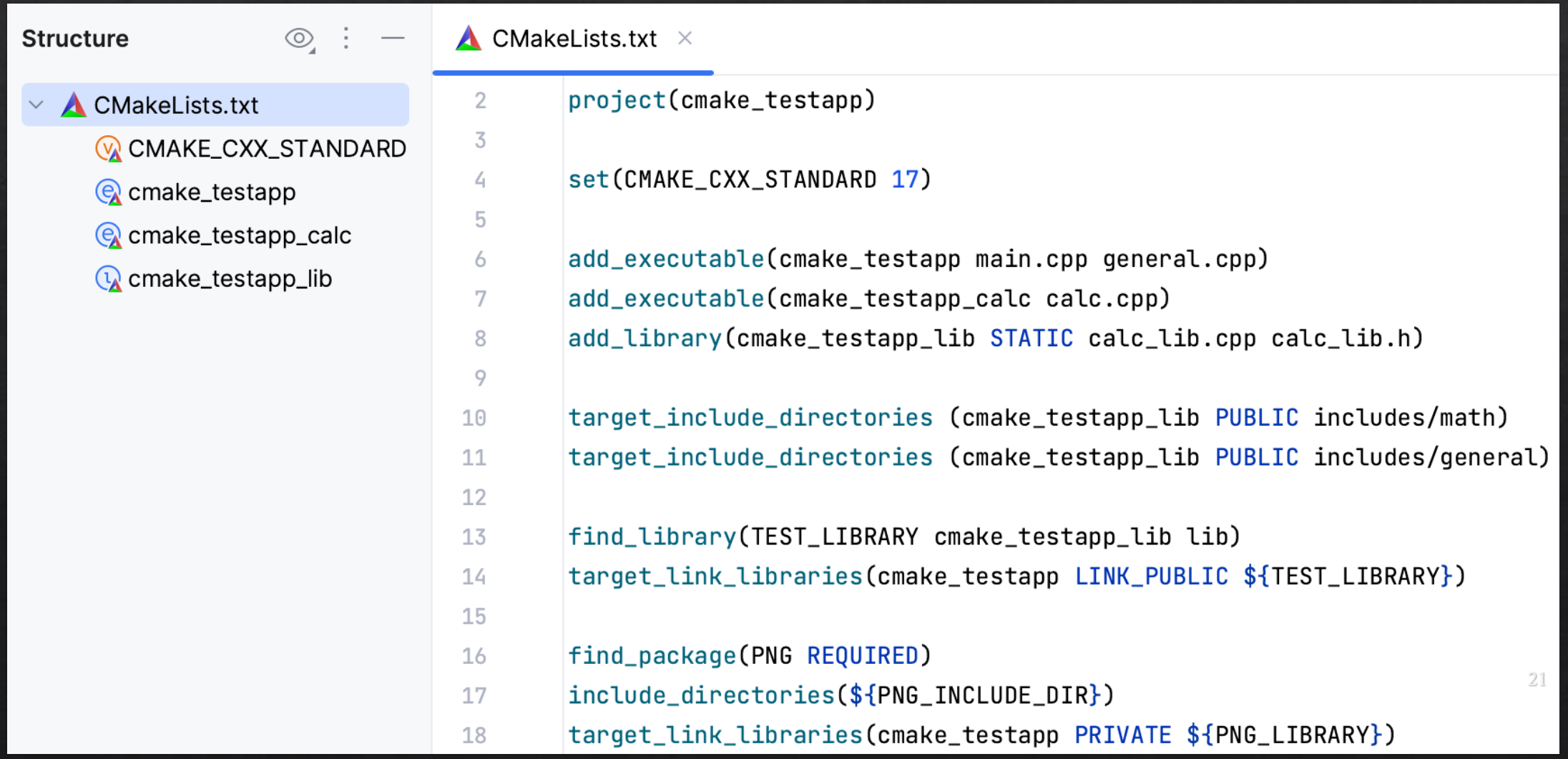
Source: https://ngeor.com/assets/2018/04/15/09_15_41.png

Cloud Deployment



Source: https://stackoverflow.com/questions/68279759/how-to-build-and-deploy-kubernetes-cluster-to-google-cloud-using-cloud-build-and

Make/CMake



Source: https://www.jetbrains.com/help/clion/quick-cmake-tutorial.html#code-insight-cmake

This Talk

- Easy-to-use-correctly, difficult-to-use-incorrectly
- Declarative C++ language
- Declarative support in the C++ library
- Declarative Design & Practices beyond the language and library

Let's make some Declarative C++ choices

Source: https://www.world-of-lucid-dreaming.com/interstellar-lucid-dreams-and-the-fifth-dimension.html

Looping – very imperative

Use the CX register to count the loops

```
mov cx, 3
startloop:
   cmp cx, 0
   jz endofloop
   push cx
loopy:
   Call ClrScr
   pop cx
   dec cx
   jmp startloop
endofloop:
   ; Loop ended
   ; Do what ever you have to do here
```

This simply loops around 3 times calling `ClrScr`, pushing the CX register onto the stack, comparing to 0, jumping if ZeroFlag is set then jump to `endofloop`. Notice how the contents of CX is pushed/popped on/off the stack to maintain the flow of the loop.

Share Edit Follow Flag

edited Oct 26, 2015 at 0:31

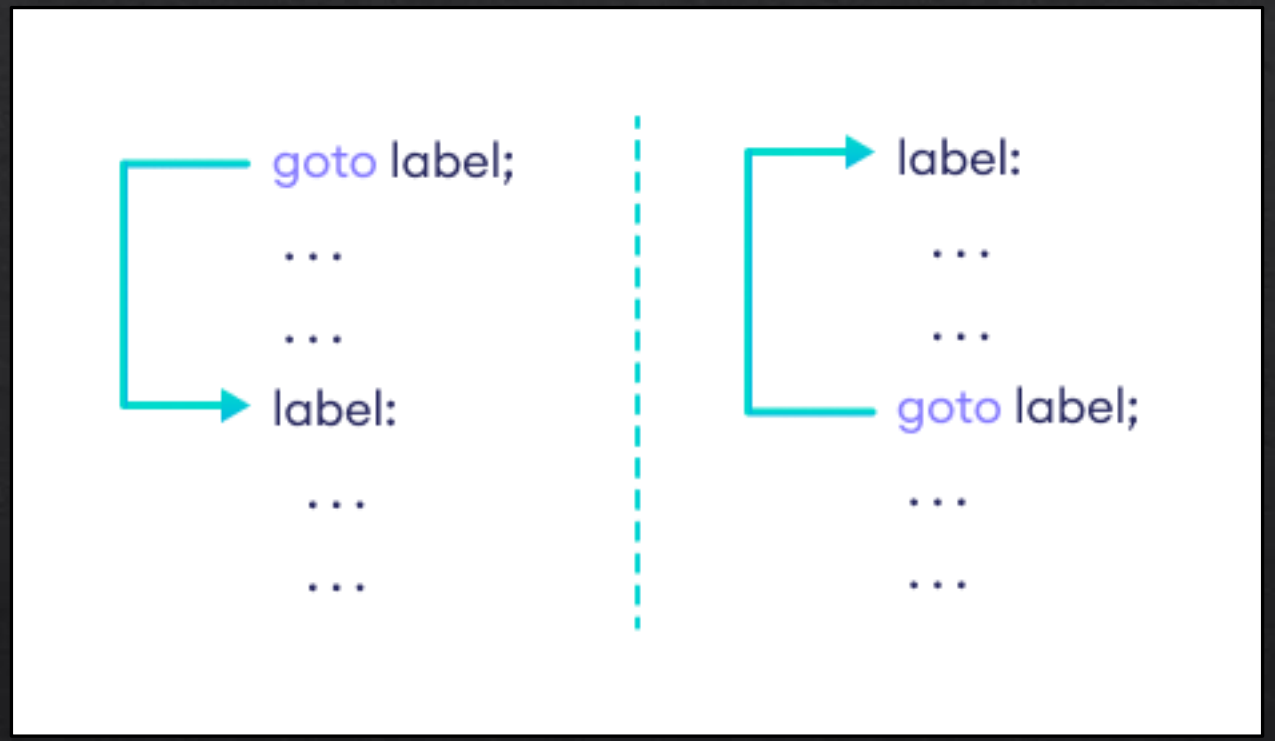
answered Feb 5, 2010 at 18:38

t0mm13b

34.6k • 8 • 84 • 112

```
.L282:
        movzx   edx, BYTE PTR [rax]
        mov     r13, rax
        add     rax, 1
        cmp     dl, 32
        jne     .L234
        cmp     rbx, rax
        je      .L263
        cmp     BYTE PTR [rax], 0
        jne     .L282
        lea     rax, [r13+2]
```

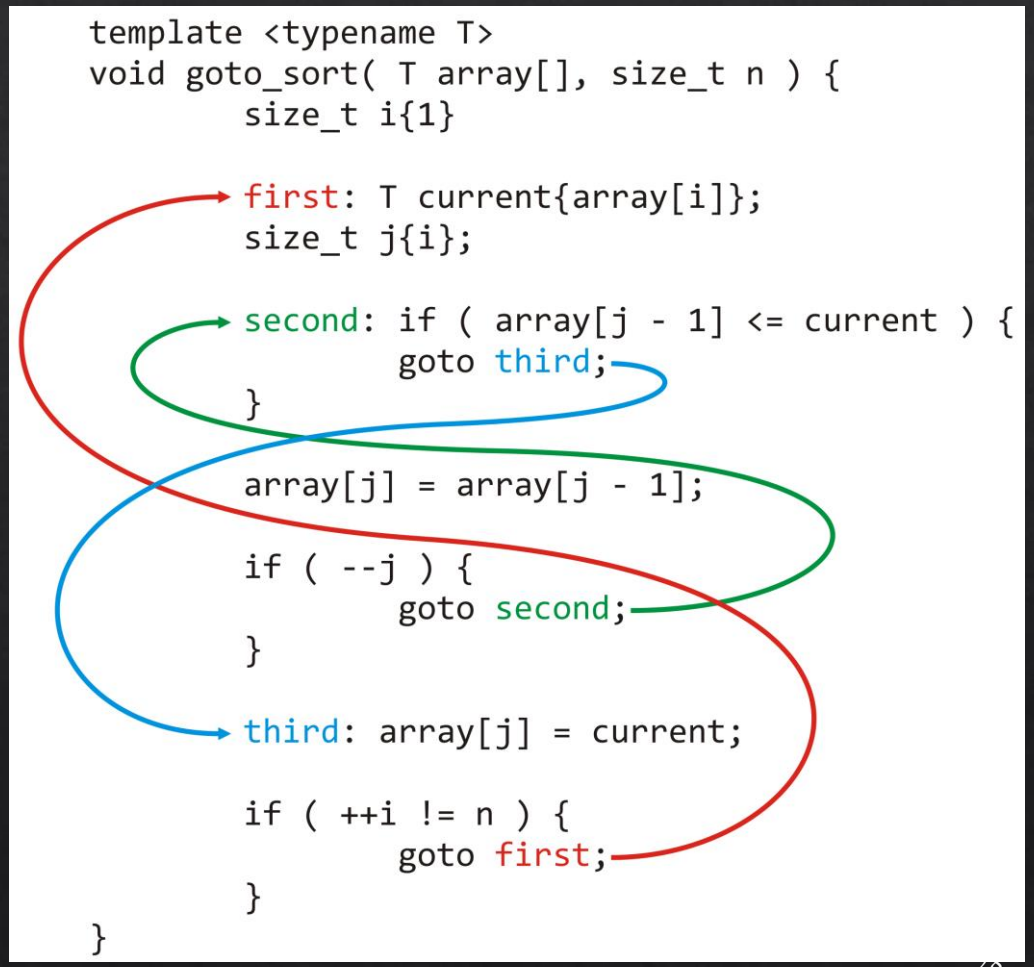
Goto – inspired by the assembler



Source: https://www.programiz.com/cpp-programming/goto

goto hell

- It quickly became goto hell
- And, who would've known that this is sorting!



```
template <typename T>
void goto_sort( T array[], size_t n ) {
        size_t i{1}

first:  T current{array[i]};
        size_t j{i};

second: if ( array[j - 1] <= current ) {
                goto third;
        }

        array[j] = array[j - 1];

        if ( --j ) {
                goto second;
        }

third:  array[j] = current;

        if ( ++i != n ) {
                goto first;
        }
}
```


Source: https://ece.uwaterloo.ca/~ece150/Other_features/12/

`while` – flush streams more than 90% full

```
void flush_streams(std::vector<stream>& streams) {
    int i = 0;
    while (i < streams.size()) {
        auto s = streams[++i];
        if (s.occupancy() >= 90) {
            s.flush();
            ++stats.no_flushes;
        }
    }
}
```

- Much clearer!
- Any issues?

for indexed

```
void flush_streams(std::vector<stream>& streams) {
    for (int i = 0; i < streams.size(); ++i) {
        auto s = streams[i++];
        if (s.occupancy() >= 90) {
            s.flush();
            ++stats.no_flushes;
        }
    }
}
```

- Better – for is more declarative

for with iterator

```
void flush_streams(std::vector<stream>& streams) {
    for (std::vector<stream>::iterator it = begin(streams);
         it != end(extremes); ++it) {
        auto s = *it;
        if (s.occupancy() >= 90) {
            s.flush();
            ++stats.no_flushes;
        }
    }
}
```

- No index issues. The intent of iterating over a vector is clear!
- All well now?

range-for

```
void flush_streams(std::vector<stream>& streams) {
    for (auto& s : streams) {
        if (s.occupancy() >= 90) {
            s.flush();
            ++stats.no_flushes;
        }
    }
}
```

- Good enough?

<algorithm>

```
void flush_streams(std::vector<stream>& streams) {
    auto it = std::find_if(begin(streams), end(streams),
        [](auto& s) { return s.occupancy() >= 90; });

    if (it != end(streams)) {
        it->flush();
        ++stats.no_flushes;
    }
}
```

- Separate business logic and control flow
- Algorithms name & express intent – find vs plain looping
- Okay, this is flushing only the first stream, but you get the idea!

100+ algorithms!

- count, copy, transform, fill, remove, reverse, sort, next_permutation, nth_element, etc.
- xxx_if, xxx_if_not, xxx_n, xxx_of
- Enable easy parallelism!

Algorithm library

Constrained algorithms and algorithms on ranges (C++20)

Constrained algorithms, e.g. ranges::copy, ranges::sort, ...

Execution policies (C++17)

is_execution_policy (C++17)

execution::seq (C++17)
execution::par (C++17)
execution::par_unseq (C++17)
execution::unseq (C++20)

execution::sequenced_policy (C++17)
execution::parallel_policy (C++17)
execution::parallel_unsequenced_policy (C++17)
execution::parallel_unsequenced (C++20)

Non-modifying sequence operations

Batch operations

for_each
for_each_n (C++17)

Search operations

all_of (C++11)
any_of (C++11)
none_of (C++11)
count
count_if
mismatch
equal

find
find_if
find_if_not (C++11)
find_end
find_first_of
adjacent_find
search
search_n

Modifying sequence operations

Copy operations

copy
copy_if (C++11)
copy_backward

copy_n (C++11)
move (C++11)
move_backward (C++11)

Swap operations

swap
iter_swap

swap_ranges

Transformation operations

replace
replace_if
transform

replace_copy
replace_copy_if

Generation operations

fill
fill_n

generate
generate_n

Removing operations

remove
remove_if
unique

remove_copy
remove_copy_if
unique_copy

Order-changing operations

reverse
reverse_copy
rotate
rotate_copy

random_shuffle (until C++17)
shuffle (C++11)
shift_left (C++20)
shift_right (C++20)

Sampling operations

sample (C++17)

Numeric operations

iota (C++11)
inner_product
adjacent_difference

accumulate
reduce (C++17)
transform_reduce (C++17)

Operations on uninitialized memory

uninitialized_copy
uninitialized_move (C++17)
uninitialized_fill

uninitialized_copy_n (C++11)
uninitialized_move_n (C++17)
uninitialized_fill_n

Sorting and related operations

Partitioning operations

partition
partition_copy (C++11)
stable_partition

is_partitioned (C++11)
partition_point (C++11)

Sorting operations

sort
stable_sort
partial_sort
partial_sort_copy

is_sorted (C++11)
is_sorted_until (C++11)
nth_element

Binary search operations (on partitioned ranges)

lower_bound
upper_bound

equal_range
binary_search

Set operations (on sorted ranges)

includes
set_union
set_intersection

set_difference
set_symmetric_difference

Merge operations (on sorted ranges)

merge

inplace_merge

Heap operations

push_heap
pop_heap
make_heap

sort_heap
is_heap (C++11)
is_heap_until (C++11)

Minimum/maximum operations

max
min
minmax (C++11)
clamp (C++17)

max_element
min_element
minmax_element (C++11)

Lexicographical comparison operations

lexicographical_compare
lexicographical_compare_three_way (C++20)

Permutation operations

next_permutation
prev_permutation

is_permutation (C++11)

C library

qsort

bsearch

partial_sum
inclusive_scan (C++17)
exclusive_scan (C++17)

transform_inclusive_scan (C++17)
transform_exclusive_scan (C++17)

destroy (C++17)
destroy_n (C++17)
destroy_at (C++17)
construct_at (C++20)

uninitialized_default_construct (C++17)
uninitialized_value_construct (C++17)
uninitialized_default_construct_n (C++17)
uninitialized_value_construct_n (C++17)

<algorithm> <execution>

```
void flush_all_streams(std::vector<stream>& streams) {
    std::for_each(std::execution::par
        begin(streams), end(streams),
        [&stats](auto& s) { s.flush(); ++stats.no_flushes; });
}
```

- Easy parallelism
- Just ask for it and the compiler will oblige
- Execution policies: `seq`, `par`, `par_unseq`, `unseq`
- Custom policies in the future like `cuda`, `opencl`, etc.
- New possibilities become a reality when functionality is segregated into small blocks
- Enabling for a longevity & a better future?!

Concurrency & Parallelism

Concurrency!?

- If the send function below had to work in a concurrent context

```
void send(const std::vector<message>& msgs);
void send(std::vector<message>&& msgs);
```

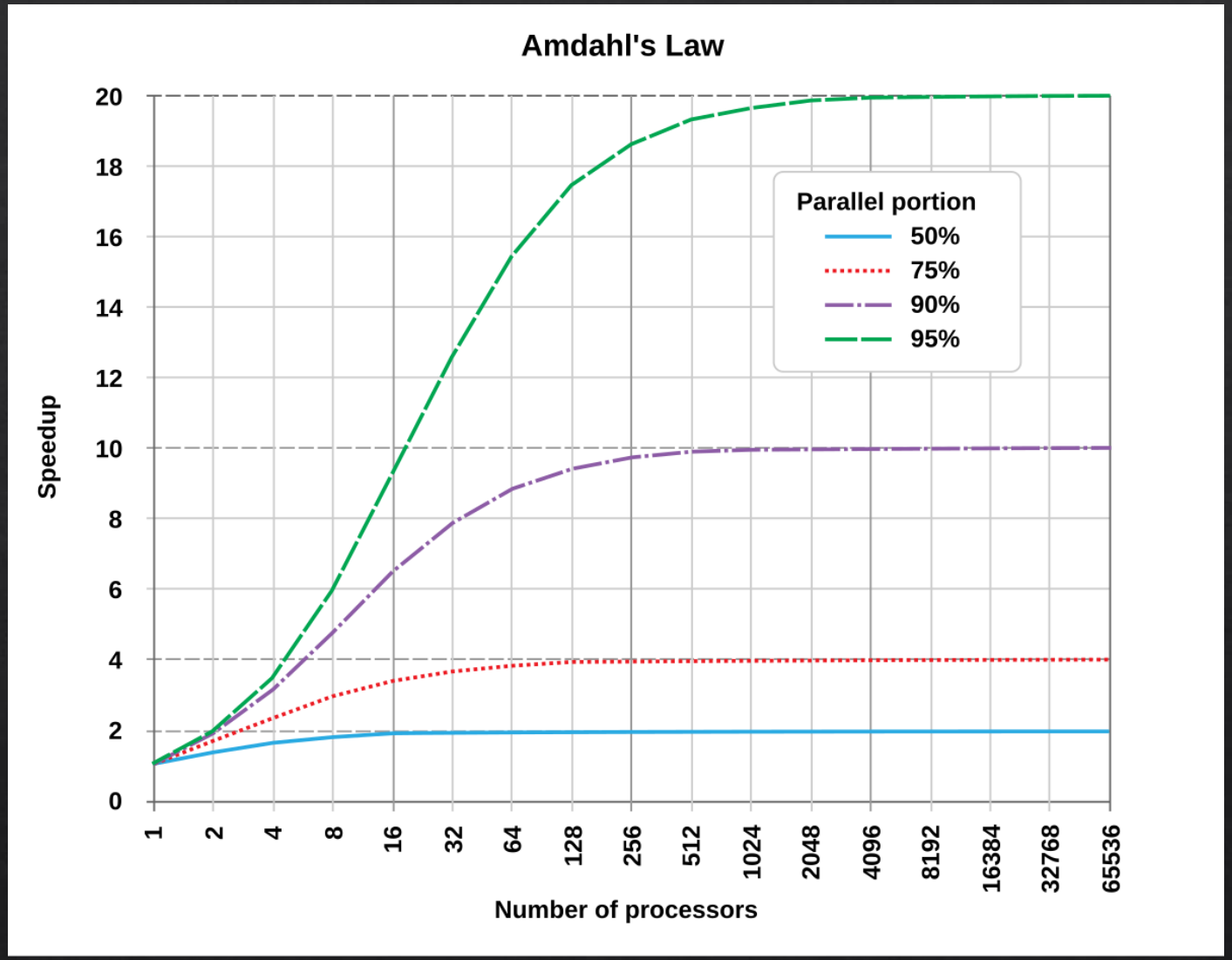
- rvalue&& forces the caller to relinquish the object

- Better still, work with value types
- It promotes unidirectional flow

```
void send(std::vector<message> msgs);
```

Shared Objects

- Pass values & ownership
- Just avoid sharing
- Remember Amdahl's law
- The slowdown is exponential
- Dataraces kill (literally)



Source: https://en.wikipedia.org/wiki/Amdahl%27s_law

Source: https://steemit.com/child-development/@phunke/how-to-teach-siblings-how-to-share-20171225t94730203z

Parallel Algorithms – just ask for it

```
int a[] = {0, 1};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int i)
{
    v.push_back(i * 2 + 1); // Error: data race
});
```

- Beware of concurrent accesses
- Attachment… er sharing… beware

Parallel Algorithms – Sharing

```
int a[] = {1, 2};
std::mutex m;
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int i)
{
    std::lock_guard<std::mutex> guard(m);
    v.push_back(i * 2 + 1); //No data race
});
```

- Mutual exclusion reinstates sanity & correctness
- ~~Attachment… er sharing… is a big problem~~

Guarded Shared Objects for concurrent access

```
template <typename T, typename M = std::mutex>
class plain_guarded
{
      template <typename... Us>
      plain_guarded(Us&&... params_to_construct_T);

   private:
      T m_obj;
      mutable M m_mutex;
};
```

- Source: https://github.com/copperspice/cs_libguarded

Guarded Shared Objects for concurrent access


```
template <typename T, typename M = std::mutex>
class plain_guarded
{
   private:
      class deleter;
      class const_deleter;
   public:
      using handle       = std::unique_ptr<T, deleter>;
      using const_handle = std::unique_ptr<const T, const_deleter>;

      [[nodiscard]] handle lock();
      [[nodiscard]] const_handle lock() const;
};
```

Guarded Shared Object

- The Guarded Shared Object cannot be used without locking

```
plain_guarded<std::vector<int>> guarded_vi{1, 2, 4};
CHECK(std::cref(guarded_vi).lock()->size() == 3);
```

- Non-const operations cannot be performed on a const object – Safe!

```
auto const_vec = std::cref(guarded_vi).lock();
const_vec->insert(begin(const_vec), 10);      //Doesn't compile
```

- Non-const on non-const is okay

```
CHECK(guarded_vi.lock()->push_back(10));    //Works!
```

Guarded Shared Object – Beware: Hidden locks

- If the `operator->` locked, the same code could be terser

```
plain_guarded<std::vector<int>> guarded_vi{1, 2, 4};
CHECK(std::cref(guarded_vi)->size() == 3);
```

- Non-const operations cannot be performed on a const object – Safe!

```
auto const_vec = std::cref(guarded_vi).lock();
const_vec->insert(begin(const_vec), 10);      //Doesn't compile
```

- Non-const on non-const is okay

```
CHECK(guarded_vi->push_back(10));    //Works!
```

Deducing this

- The plain_guarded can enable seamless usage by defining `operator->`

```
template<typename Self>
auto operator->(this Self&& self) {
    return self.lock();
}
```

- The return is const based on whether this was deduced to be const
- More bang for the buck!

synchronized_value

```
template<typename Type>
class synchronized_value
{
public:
    synchronized_value(const synchronized_value&) = delete;
    synchronized_value &operator=(const synchronized_value&) = delete;

    template<typename... Args>
    synchronized_value(Args&&... args)
        : val (std::forward<Args> (args)...)
    {}

    template<typename Fn, typename Up, typename... Types>
    friend std::invoke_result_t<Fn, Up&, Types&...> apply (Fn&&, synchronized_value<Up>&,
                                                           synchronized_value<Types>&...);

private:
    std::mutex mutex;
    Type val;
};
```

Sharing

- Look out for data sharing when using parallel algorithms

- Make invalid operations (data races) impossible
- RAII magic takes care of locks
- Distinguish const access vs. non-const access

- With deducing this, write once & generate many
- Exploit the type system

Operators

3-way comparison

- Just ask for the comparisons to be generated!

```
struct destination
{
    ip_t ip;
    port_t port;
    protocol proto;

    auto operator<=>(const destination&) const = default;
};


destination d1{"192.168.0.1"_ip, 80, udp};

destination d2 = d1;

CHECK(d1 == d2);
```

User-defined literals

- Add to your vocabulary

```
boost::asio::ip::address operator ""_ip(const char* val, std::size_t) {
    return to_ip(val);
}

CHECK("192.168.0.1"_ip == boost::asio::ip::address::from_string("192.168.0.1"));
CHECK("ff06::c3"_ip == boost::asio::ip::address::from_string("ff06::c3"));
CHECK("ff06::c3:8080"_hostport == hostport{"ff06::c3", 8080});
```

```
using RE = RegExp<Begin,
Select<Char<'a','b','c'>,String<'x','y','z'>>,
                Plus<Anything>,End>
```

```
auto re = "^(?:[abc]|xyz).+$"_pre;
static_assert(is_same(RE,decltype(re)));
```

I write LL1 or parser, which transformed this string

Regular Expressions Redefined in C++

Operators

- Overloading operators makes generic implementations possible
- Functionality can be hidden behind operators

- Operators +, -, *, / on any custom numeric types, matrices, physical quantities,
- Dereferencing operators for pointer-like types
- Comparison operators

- Operators can enable expressiveness

Parameter Packs

Fold Expressions

- Go functional

```
template<typename... Args>
void log(Args&&... args) {
    (std::cout << ... << args) << '\n';
}
log(time_point1, service_name, “No response from remote: ", destination1);
```

More Fold

```
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
```

Easy…[1] Peasy…[2]! (P2662)

- Working with parameter packs is just like working with arrays!

```cpp
template <typename... T>
constexpr auto first(T... values) -> T...[0] {
  return values...[0];
}

template <typename... T>
constexpr auto last(T... values) -> T...[sizeof...(values)-1] {
  return values...[sizeof...(values)-1];
}

int main() {
  // first(); // ill-formed: invalid index 0 for pack 'T' of size 0
  static_assert(first(1, 2, 3, 4, 5) == 1);
  static_assert(last(1, 2, 3, 4, 5) == 5);
  static_assert(first(1, 2, 3, 4, 5) + last(1, 2, 3, 4, 5) == 6);
}
```


Source: https://www.sandordargo.com/blog/2025/01/22/cpp26-pack-indexing

Feature Packed!

- Working with Parameter packs got so much easier
- Folds enable declarative expressivity and terseness
- Compiler does the magic

- Working with packs = easy!
- ~~indexed_sequence_for~~ or ~~tuples~~ – phew!

Generic Functions

Custom algorithm

```
void flush_streams(std::vector<stream>& streams) {
    for_each_if(begin(streams), end(streams),
        [&stats](auto& s) { s.flush(); ++stats.no_flushes; }); //action
        [](const auto& s) { return s.occupancy() >= 90; });    //condition
}
```

- Functions and Lambdas enable implementation in terms of expressions
- Separate the decisions from the actions
- The code clearly tells us what it is doing
- Or does it???

Name it!

```
void flush_streams(std::vector<stream>& streams) {
    auto is_full = [](const auto& s) { return s.occupancy() >= 90; };
    auto do_flush = [&stats](auto& s) { s.flush(); ++stats.no_flushes; };

    for_each_if(begin(streams), end(streams),
        do_flush,
        is_full);
}
```

- Naming things makes the intent clear – can't be stressed enough

No hard-coding!

```
void flush_streams(std::vector<stream>& streams) {
    auto is_full = [](const auto& s) { return s.occupancy() >= g_threshold; };
    auto do_flush = [&stats](auto& s) { s.flush(); ++stats.no_flushes; };

    for_each_if(begin(streams), end(streams),
        do_flush,
        is_full);
}
```

- Better not to hard-code
- But dependency on globals changes the behaviour of this function! Huh!

Pure functions!

```
void flush_streams(std::vector<stream>& streams, int threshold) {
    auto is_full = [threshold](const auto& s) {
        return s.occupancy() >= threshold; };
    auto do_flush = [&stats](auto& s) { s.flush(); ++stats.no_flushes; };

    for_each_if(begin(streams), end(streams),
        do_flush,
        is_full);
}
```

- Pure function – depends only on passed parameters – no globals too!
- Eliminate hard-coding – the function is not tied to a value

Pure vs. Impure

Impure

- Global dependency = painful to understand
- Hard to test
- Taxing to debug
- Not easy to parallelise
- Not composable

Pure

- Local reasoning applies = easy to understand
- Easily testable
- Quick to debug
- Easy to parallelise
- Enables composition

Pure functions!

```
void flush_streams(std::vector<stream>& streams, int threshold) {
    auto is_full = [threshold](const auto& s) {
        return s.occupancy() >= threshold; };
    auto do_flush = [&stats](auto& s) { s.flush(); ++stats.no_flushes; };

    for_each_if(begin(streams), end(streams),
        do_flush,
        is_full);
}
```

- Pure function – depends only on passed parameters – no globals too!
- Eliminate hard-coding – the function is not tied to a value

Different, yet similar!

```
void dump_streams(std::vector<stream>& streams) {
  auto is_errored = [](const auto& s) { return s.status() == status::errored;};
  auto do_dump = [&stats](auto& s) { s.dump(); ++stats.no_dumps; };

  for_each_if(begin(streams), end(streams),
        do_dump,
        is_errored);
}
```

- Easy support for new functionality!

Algorithm implementation

```
template<typename InputIt, typename Action, typename Predicate>
void for_each_if(InputIt b, InputIt e, Action action, Predicate pred) {
    for (; b != e; ++b) {
        if (std::invoke(pred, *b)) {       //condition
            std::invoke(action, *b);       //action
        }
    }
}
```

- A generic functionality enables new uses
- The control flow is hidden in this function and enables declarative code elsewhere
- But a template parameter is like a void pointer!?

Algorithm implementation

- What are our expectations from the template parameters?

```
template<typename InputIt, typename Action, typename Predicate>
requires std::input_iterator<InputIt> &&
         std::indirectly_unary_invocable<Action, InputIt> &&
         std::indirect_unary_predicate<Predicate, InputIt>
void for_each_if(InputIt b, InputIt e, Action action, Predicate pred) {
    for (; b != e; ++b) {
        if (std::invoke(pred, *b)) {       //condition
            std::invoke(action, *b);       //action
        }
    }
}
```


- Having declared our expectations, the compiler will help catch type mismatches at the call site

Parental advice!

- It is great to be able to identify an algorithm and add it to the vocabulary of operations!
- Very declarative!

Source: https://tenor.com/en-GB/view/rotate-that%27s-a-rotate-sean-parent-gif-13970213258665078607

Overloading with concepts

```
#include <bit>
constexpr bool is_pow_of_2(std::unsigned_integral auto n) {
    return std::has_single_bit(n);
}
constexpr bool is_pow_of_2(auto n) {
    return false;
}
```

- The right overload is invoked depending on whether an unsigned is passed or not
- Concepts help overload – keeping user code the same

Generic Functions

- Purity enables for predictable, correct and testable behaviour
- Functions & lambdas help name operations

- Generic implementations build a vocabulary of constructs
- Enables novel, yet unknown, uses

- Concepts and constraints help specify type expectations
- Enables overloads
- Helpful compiler errors at the place of incorrect use

Higher-level decisions & actions

```
void flush_streams(std::vector<stream>& streams, int threshold) {
    for_each_if(streams,
        [threshold](const auto& s) { return s.occupancy() == threshold; },
        [&stats](const auto& s) { s.flush(); ++stats.no_flushes; });
}
void dump_streams(std::vector<stream>& streams) {
    for_each_if(streams,
        [threshold](const auto& s) { return s.status() == status::errored; },
        [&stats](const auto& s) { s.dump(); ++stats.no_dumps; });
}
```

- The functions only take domain level decisions and do NOT do low-level actions

<ranges>


```
void flush_streams(std::vector<stream>& streams, int tshld) {
    auto is_full = [tshld](const auto& s) { return s.occupancy() >= tshld; };
    auto do_flush = [&stats](auto& s) { s.flush(); ++stats.no_flushes; };

    auto full_streams = streams | std::views::filter(is_full);

    std::ranges::for_each(full_streams, do_flush);
}
```

- Passing `begin` & `end` everywhere – there was a missing abstraction – the `range`!
- Several operations chained to compose the pipeline – Lazy & declarative
- Ranges take lambdas, functions, member function pointers, std::function – any callable
- I have shown, both, Ranges factory & adaptors style (pipe) & Ranges algorithms (for_each)

std::ranges::to<>

- Return all words in a string that are greater than 5

```
auto words_longer_than_5(const std::string& text) {
  namespace stdv = std::views;
  return stdv::split(text, " ")
         | stdv::filter([](const auto& r) { return ssize(r) > 5; })
         | stdv::transform([](const auto& r) {
             return std::string(begin(r), end(r)); })
         | std::ranges::to<std::vector>();
}
```

- `to` terminates the pipeline & collects the items
- Type of data in the vector is what it receives from the previous step
- No raw loops, remember?

Ranges Adaptors

- Traverse only the keys

```
std::map<std::string, int> map{{"one", 1}, {"two", 2}};

for (auto const& key : std::views::keys(map))
    std::cout << key << ' ';
// prints: one two
```

Range Adaptors

```
const std::vector<std::tuple<std::string, double, bool>> quark_mass_charge {
    // name, MeV/c², has positive electric-charge:
    {"up", 2.3, true}, {"down", 4.8, false},
    {"charm", 1275, true}, {"strange", 95, false},
    {"top", 173'210, true}, {"bottom", 4'180, false},
};
```

Range Adaptors

```
const std::vector<std::tuple<std::string, double, bool>> quark_mass_charge;

std::cout.imbue(std::locale("en_US.utf8"));
std::cout << "Quark name:   | ";
for (std::string const& name : std::views::keys(quark_mass_charge))
    std::cout << std::setw(9) << name << " | ";

std::cout << "\n" "Mass MeV/c²: | ";
for (const double mass : std::views::values(quark_mass_charge))
    std::cout << std::setw(9) << mass << " | ";

std::cout << "\n" "E-charge:     | ";
for (const bool pos : std::views::elements<2>(quark_mass_charge))
    std::cout << std::setw(9) << (pos ? "+2/3" : "-1/3") << " | ";
std::cout << '\n';
```

Range Algorithms

```cpp
namespace ranges = std::ranges;

int main()
{
    const auto v = {3, 9, 1, 4, 1, 2, 5, 9};
    const auto [min, max] = ranges::minmax_element(v);
    std::cout
        << "min = " << *min << ", at [" << ranges::distance(v.begin(), min) << "]\n"
        << "max = " << *max << ", at [" << ranges::distance(v.begin(), max) << "]\n";
}
```

A <ranges> of options!

- Enjoy the laziness!
- Express ideas clearly
- Compose, compose, compose!
- Only implement your custom condition or transformation
- Enjoy the machinery

- No raw loops!

std::optional<T>

Maybe

- Series of operations
- Clean way of indicating failure

- Too much packing & unpacking
- Still full of conditionals

```
1  std::optional<image> get_cute_cat (const image& img
2      auto cropped = crop_to_cat(img);
3      if (!cropped) {
4        return std::nullopt;
5      }
6
7      auto with_tie = add_bow_tie(*cropped);
8      if (!with_tie) {
9        return std::nullopt;
10     }
11
12     auto with_sparkles = make_eyes_sparkle(*with_ti
13     if (!with_sparkles) {
14       return std::nullopt;
15     }
16
17     return add_rainbow(make_smaller(*with_sparkles)
18 }
```

Monadic operations

- Easy chaining!
- No manual wrapping & unpacking/checks
- and_then → callables returning optional
- transform → callables returning value

```
2  crop_to_cat(img)
3     .and_then(add_bow_tie)
4     .and_then(make_eyes_sparkle)
5     .transform(make_smaller)
6     .transform(add_rainbow);
```

Check Jonathan Muller's talk tomorrow!

Coroutines & Generators

Coroutines & Generators

```
template<typename T>
struct Tree
{
    T value;
    Tree *left{}, *right{};
};
```

Coroutines & Generators


```
template<typename T>
struct Tree
{
    std::generator<const T&> traverse_inorder() const {
        if (left)
            for (const T& x : left->traverse_inorder())
                co_yield x;

        co_yield value;
        if (right)
            for (const T& x : right->traverse_inorder())
                co_yield x;
    }
};
```

Coroutines & Generators

```
Tree<char> tree[]
{
                                        {'D', tree + 1, tree + 2},
        //                                  |
        //                  ┌───────────────┴──────────────┐
        //                  |                              |
                    {'B', tree + 3, tree + 4},       {'F', tree + 5, tree + 6},
        //                  |                              |
        //          ┌───────┴──────┐               ┌───────┴──────┐
        //          |              |               |              |
          {'A'},                {'C'}, {'E'},                  {'G'}
};

for (char x : tree->traverse_inorder())
   std::cout << x << ' ';     //A B C D E F G
std::cout << '\n';
```

co_await greater_support()

- The compiler manages saving the state and context of the coroutines
- Coroutines have a basic low-level support for now
- Tedious

- Generators can represent an infinite range

[[attributes]]

[[nodiscard]]

- We may declare that the returned object from a function be handled

```
[[nodiscard]] auto words_longer_than_5(const std::string& text) { ... }
```

```
[[nodiscard(“DB results must be processed")]]
auto fetch(const std::string& key) { ... }
```

- When status or data from expensive actions is returned
- Compiler issues a warning in the user code

[[noreturn]]

- Indicates that the function does not return

```
[[noreturn]] void event_loop(const std::string& text) { ... }
```

- If this returns, the behaviour is runtime-defined.

[[deprecated]]

- Flag unsupported functionality – convey better options

```
[[deprecated]]
void TriassicPeriod() { std::clog << "[251.9 - 208.5] million years ago.\n"; }

[[deprecated("Use NeogenePeriod() instead.")]]
void JurassicPeriod() { std::clog << "[201.3 - 152.1] million years ago.\n"; }

[[deprecated("Use calcSomethingDifferently(int).")]]
int calcSomething(int x) { return x * 2; }

int main() {
    TriassicPeriod();
    JurassicPeriod();
}
```

delete("should have a reason") (P2573)

- Say what you mean:

```
template<typename T>
struct A {/* ... */};

template<typename T>
A<T> factory(const T&) {/* process lvalue */}

template<typename T>
A<T> factory(const T&&) = delete("Using rvalue to construct A may result
in dangling reference");
```

C++26 delete now has a reason!

- Mean what you say:

```
struct MoveOnly
{
    // ... (with move members defaulted or defined)
    MoveOnly(const MoveOnly&) = delete("Copy-construction is expensive;
please use move construction instead.");
    MoveOnly& operator=(const MoveOnly&) = delete("Copy-assignment is
expensive; please use move assignment instead.");
};
```

[[strong]] types

Convert runtime values to types!

- Compile time types – no runtime overhead!

```
int start_service(const deployment_type dt, const config& conf) {
    switch(dt) {
        case flavour1: return start_service<flavour1>(conf);
        case flavour2: return start_service<flavour2>(conf);
        case flavour3: return start_service<flavour3>(conf);
        default: break;
    }
}
```

- Now the service works without any decisions for deployment types

CTAD

- Class Template Argument Deduction

```
template<class T>
struct container
{
    container(T t) {}
};

container c(7); // OK: deduces T=int using an implicitly-generated guide
std::vector<double> v = {/* ... */};
auto d = container(v.begin(), v.end()); // A container of iterators!?
```

CTAD

- Add the deduction guideline

```
template<class T>
struct container
{
    container(T t) {}

    template<class Iter>
    container(Iter beg, Iter end);
};

// additional deduction guide
template<class Iter>
container(Iter b, Iter e)
    -> container<typename std::iterator_traits<Iter>::value_type>;
```

CTAD

```
template<class Iter>
container(Iter b, Iter e) -> container<typename
std::iterator_traits<Iter>::value_type>;

// uses
container c(7); // OK: deduces T=int using an implicitly-generated guide
std::vector<double> v = {/* ... */};
auto d = container(v.begin(), v.end()); // OK: deduces T=double
container e{5, 6}; // Error: there is no std::iterator_traits<int>::value_type
```

Arguments

- If a function call needs a particular step done, ensure it through type safety

```
void send(std::vector<message> msgs);
void send(std::vector<encrypted<message>> msgs);
```

- State the intent so that the compiler can enforce the correct use

- Here, encrypted is just a wrapper to stamp out a new type

Encryptor

- The encryptor may essentially give the same type back
- But the only way to get it would be via encryption!

```
encrypted<message> s.encrypt(message);
```

Arguments

- Between these:

```
void send(const std::vector<message>& msgs);
void send(std::vector<message>&& msgs);
```

- State the intent so that the compiler can enforce optimal use

- Moreover, we pass by reference not to share but to prevent copy!!!

{ Scope-based }

Must-happen action on leaving scope

- Submit local stats to global stats

```
void f(std::vector<stream>& streams) {
    metrics stats;
    ...
    ++stats.item1;
    ...
    ++stats.item2;

    global_stats += stats;
}
```

- Is it possible that stats are not updated to `global_stats`?

Must-happen action on leaving scope

- Submit local stats to global stats

```
void f(std::vector<stream>& streams) {
    metrics stats;
    scope_exit on_exit{[stats] { global_stats += stats; }};
    ...
    ++stats.item1;
    ...
    ++stats.item2;
}
```

- RAII just works!
- Inspired by `std::experimental::scope_exit`

On the other hand, several professional programmers familiar with these techniques have related to me an experience that is too common in my own programming: when they construct a program, the "hard" parts work the first time, while the bugs are in the "easy" parts.

Jon Bentley, "Writing Correct Programs", Communications of the ACM, 26(12), December 1983

Structurally speaking

- Composition, composition, composition → flexible, changeable implementations from small parts
- Use strong types → enforce operations, conditions become type choices
- Use Value Types, regular types, movable types → chain operations, eliminate state
- Let types model concepts → compiler helps when expectations explicit, overloading
- Variadic templates, fold expressions, sum types → code generation, strong types, correctness
- Small, separate & composable → small provable reusable functionality
- Generic implementations & specialisations → contain complexity, overloading replaces conditions
- Giving up options, restrict use → easy correct use, difficult incorrect use
- Exploit the strength of C++ - RAII, Zero-overhead → higher level, automate scope-based actions

- If the code compiles, it works!

The code is the spec!

“

The essence of a program is
to create a specification
of the system

”

Ankur Satle

Thank you

Ankur Satle