Leading edge

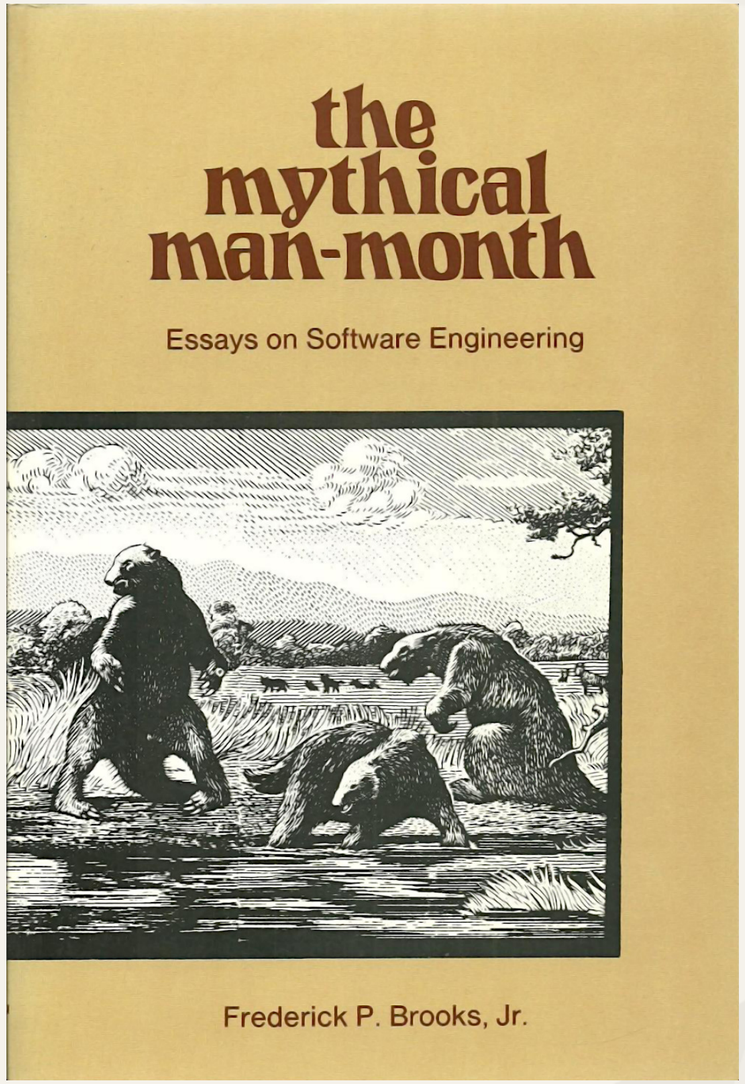
A roundup of C++20 and C++23

Kris van Rens

```
Kris van(Rens<>);
```

(source: Wikipedia, CC-BY-SA-3.0)

```
Kris van(Rens<>);
```



(source: Wikipedia, fair use)

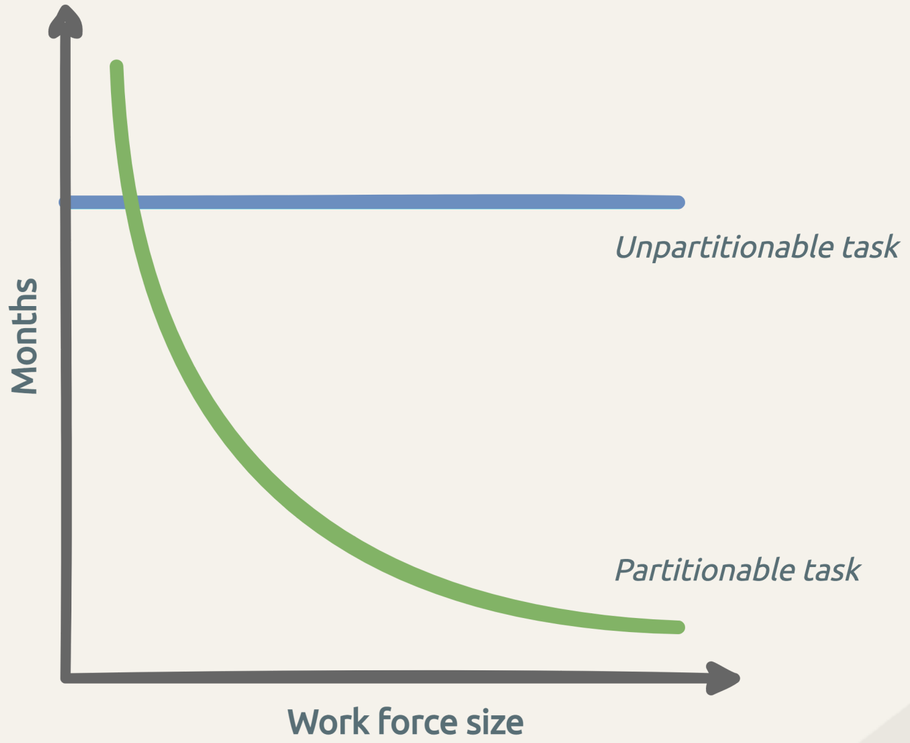
Frederick P. Brooks Jr.

April 19, 1931 – November 17, 2022

- Led the development of **IBM System/360**
- Teaching assistant for **Ken Iverson (of APL)**
- Coined term **“Computer Architecture”**
- Invented **“Brooks’s Law”**
- Moved **byte size from 6 to 8 bits**
- **IEEE John von Neumann Medal** (1993)
- **ACM Turing Award** (1999)

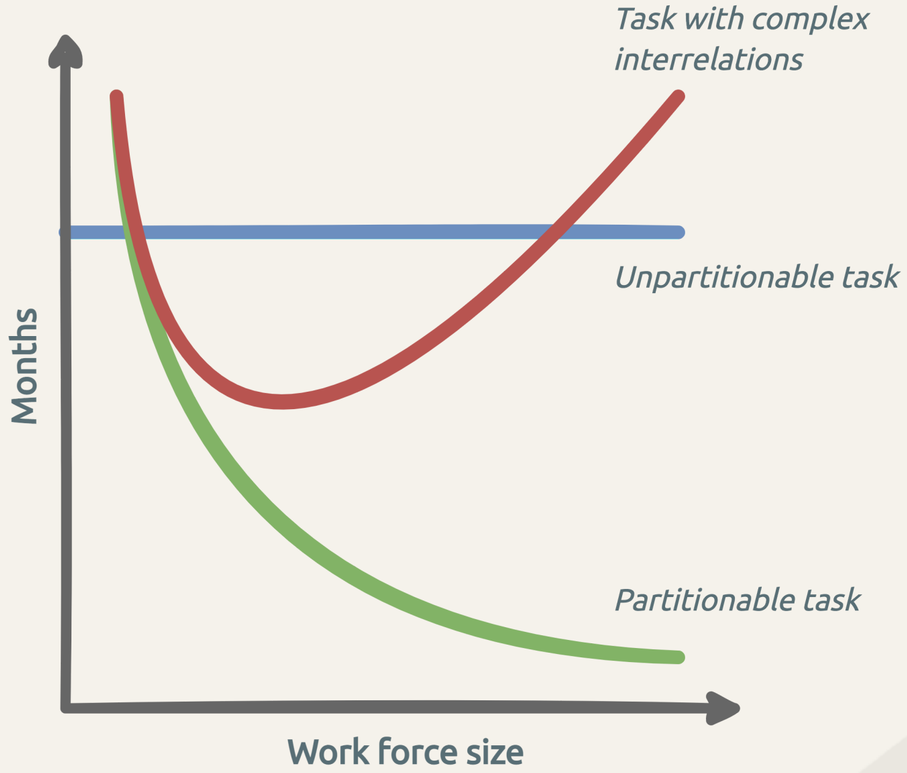
Brooks's Law:

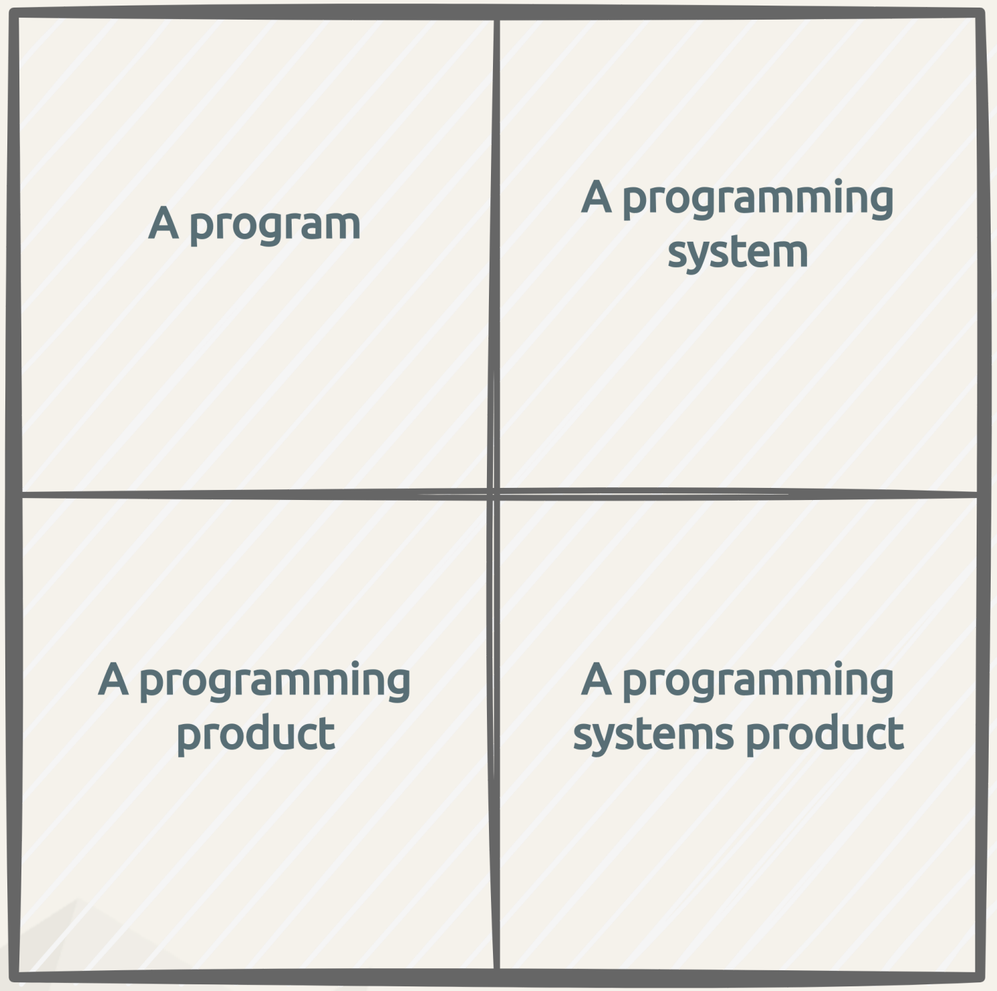
> *"Adding manpower to a late project makes it later."*

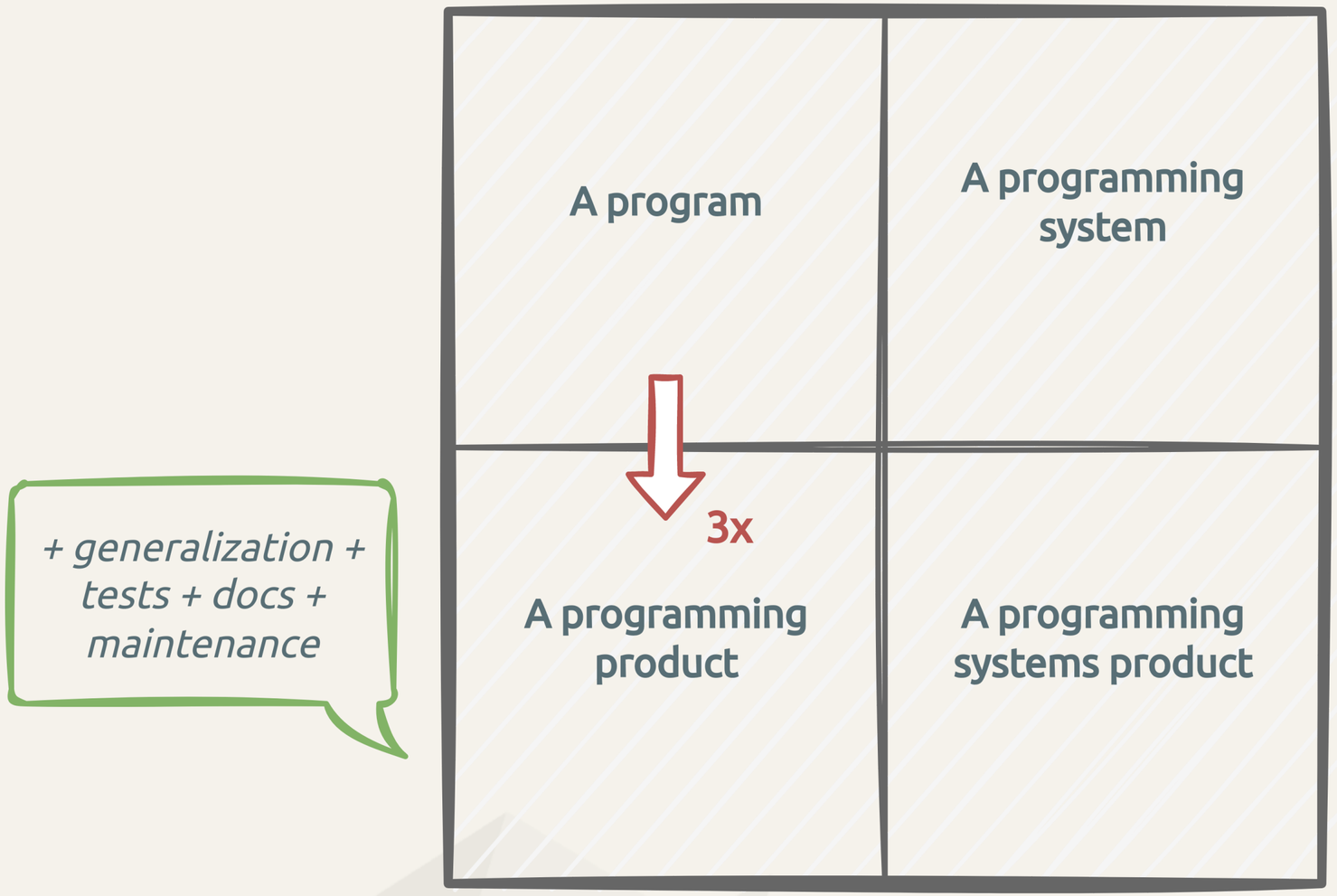


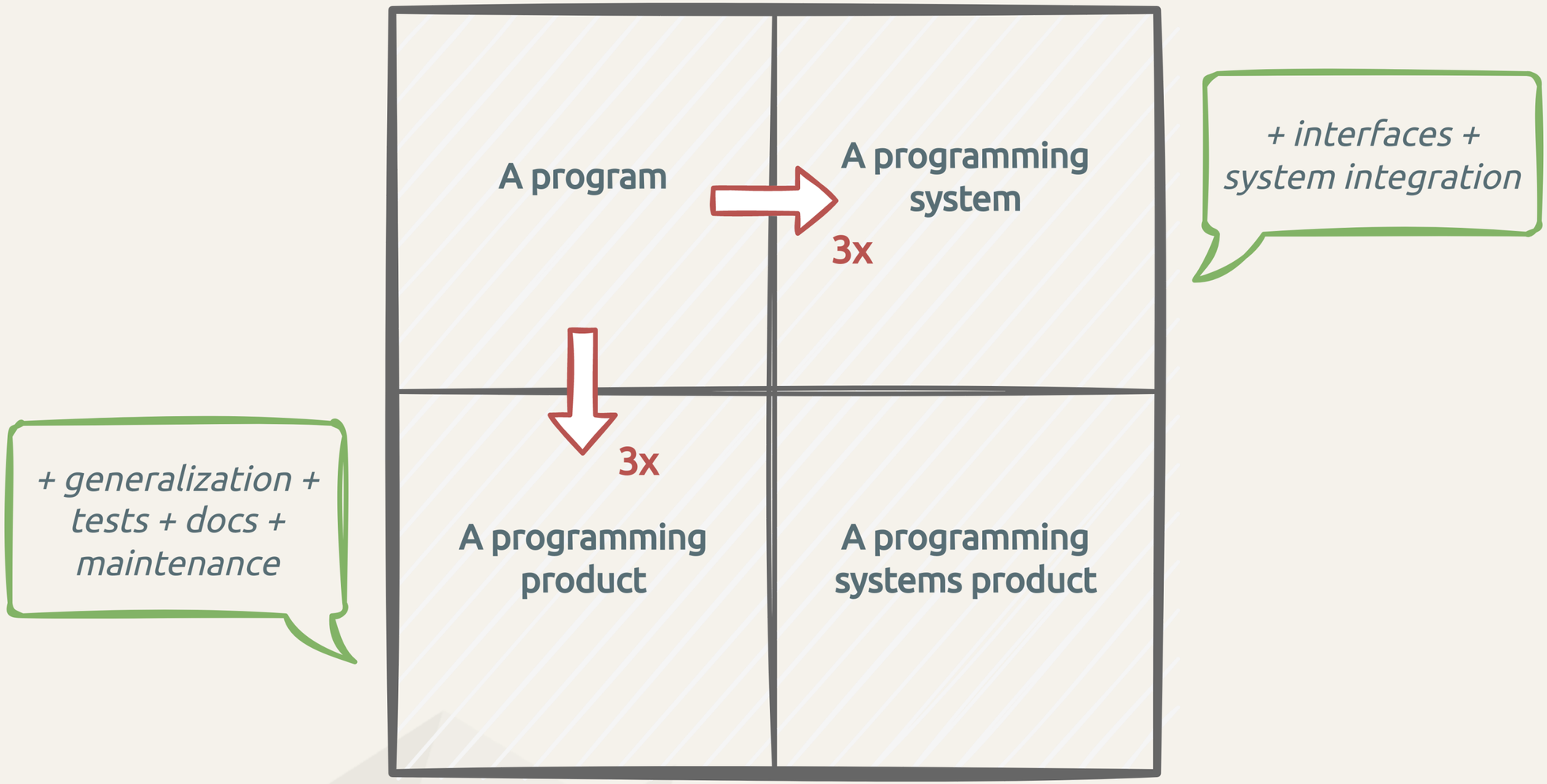
Brooks's Law:

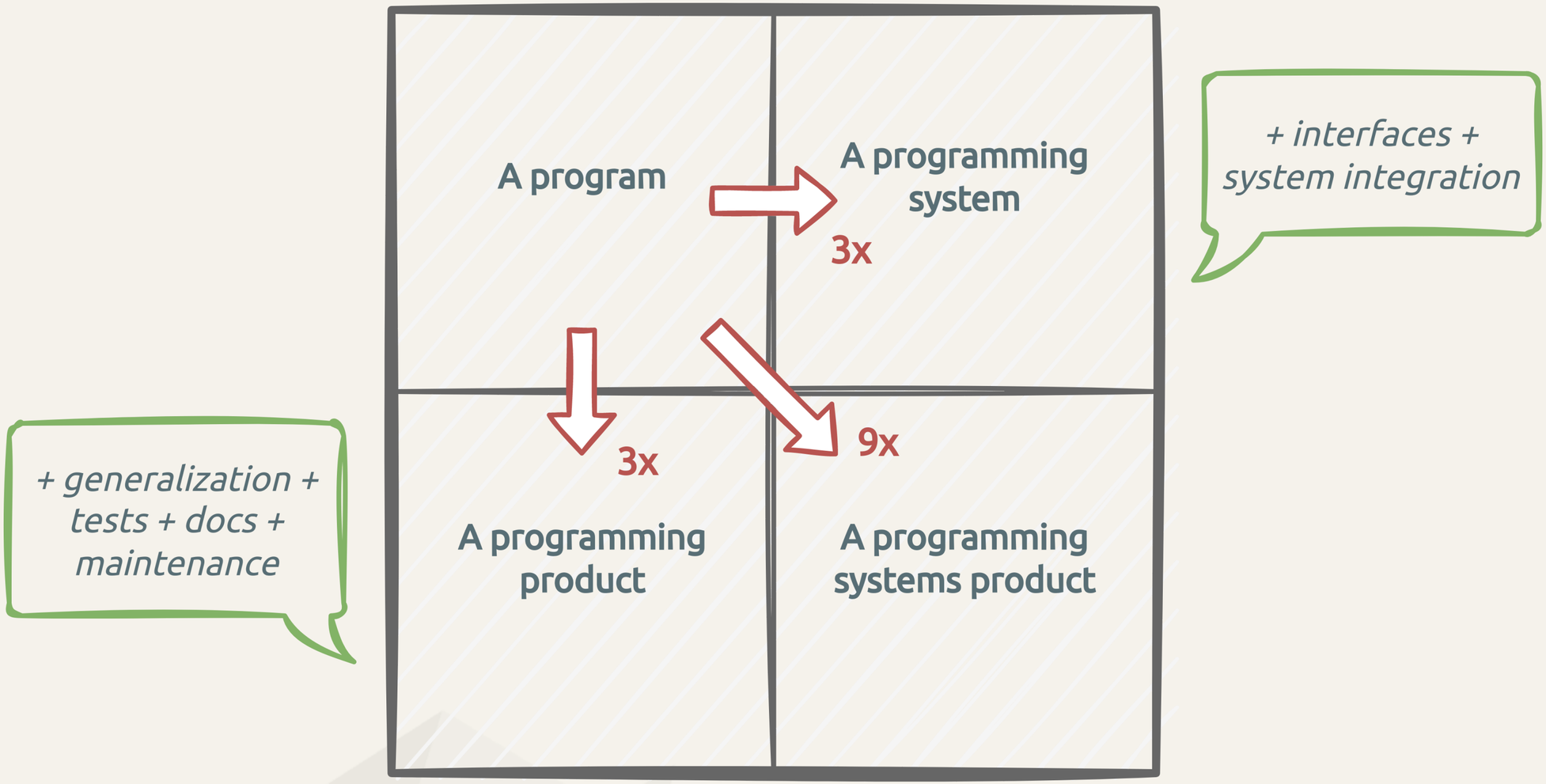
> *"Adding manpower to a late project makes it later."*




A program
A programming system
A programming product
A programming systems product


A program
A programming system
3x
A programming product
A programming systems product
+ generalization + tests + docs + maintenance


A program
A programming system
3x
3x
A programming product
A programming systems product
+ interfaces + system integration
+ generalization + tests + docs + maintenance


A program
A programming system
3x
3x
9x
A programming product
A programming systems product
+ interfaces + system integration
+ generalization + tests + docs + maintenance

What’s ahead?

- Leading Edge C++
- C++20 by example
- An outlook to C++23

A little bit about me

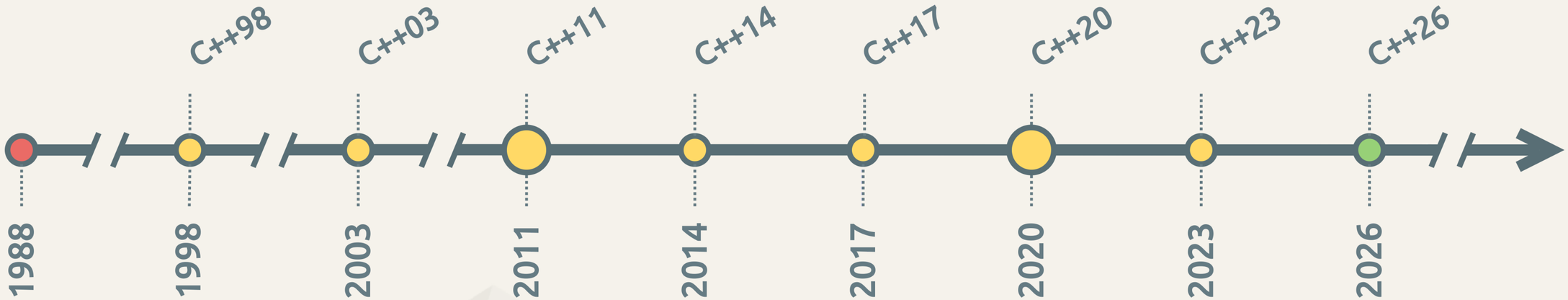
kris@vanrens.org

Leading Edge C++

Kris van(Rens<>);

How C++ is cooked up

- C++ is an **ISO-standardized language** (language + library),
- ISO WG21 comprises of **23 study groups** and a total over 150 members.



C++20 – not just any update

- **Concepts**
- **Modules**
- **Coroutines**
- **Ranges library**

- Three-way comparison ‘`operator <=>`’
- Array views with `std::span`
- Advanced text formatting library `<format>`
- New thread support primitives and `jthread`
- Calendars and time zones in `<chrono>`
- …

Compiler support for C++20/23

There are **only a few compilers** supporting C++20 in full

C++20

	Decent	Full
GCC	10	13
Clang	10	16
MSVC	19.25	19.30

(MS VS2019 ships MSVC 19.29 in v16.11.2)

C++23

	Decent	Full
GCC	13	-
Clang	16	-
MSVC	19.35⋆	-

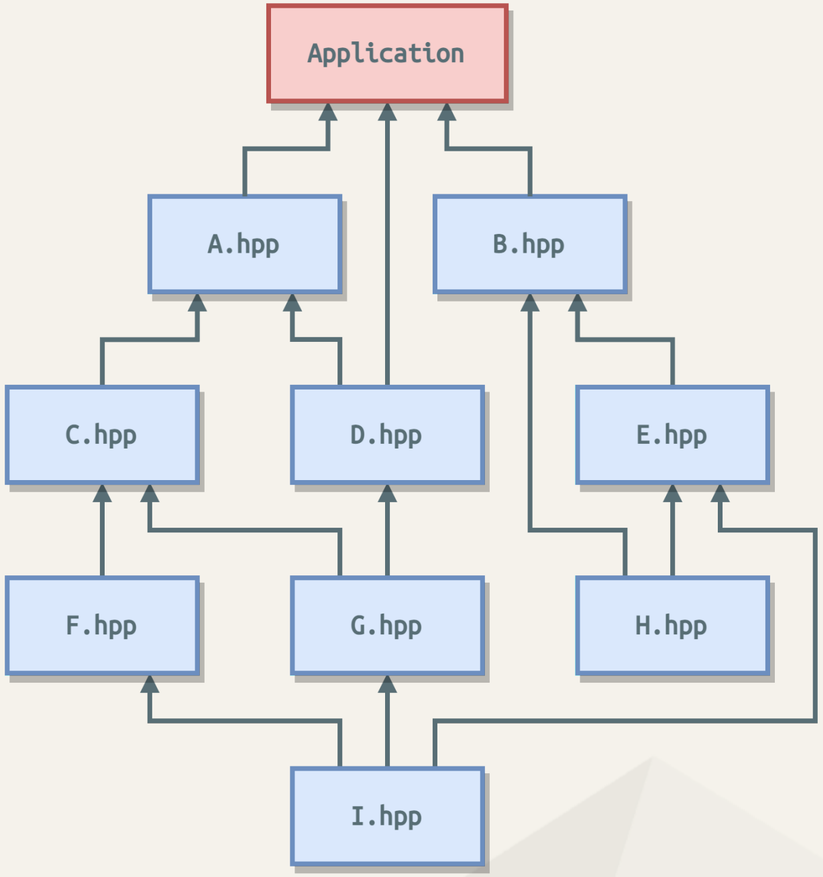
(⋆: mostly just the standard library)

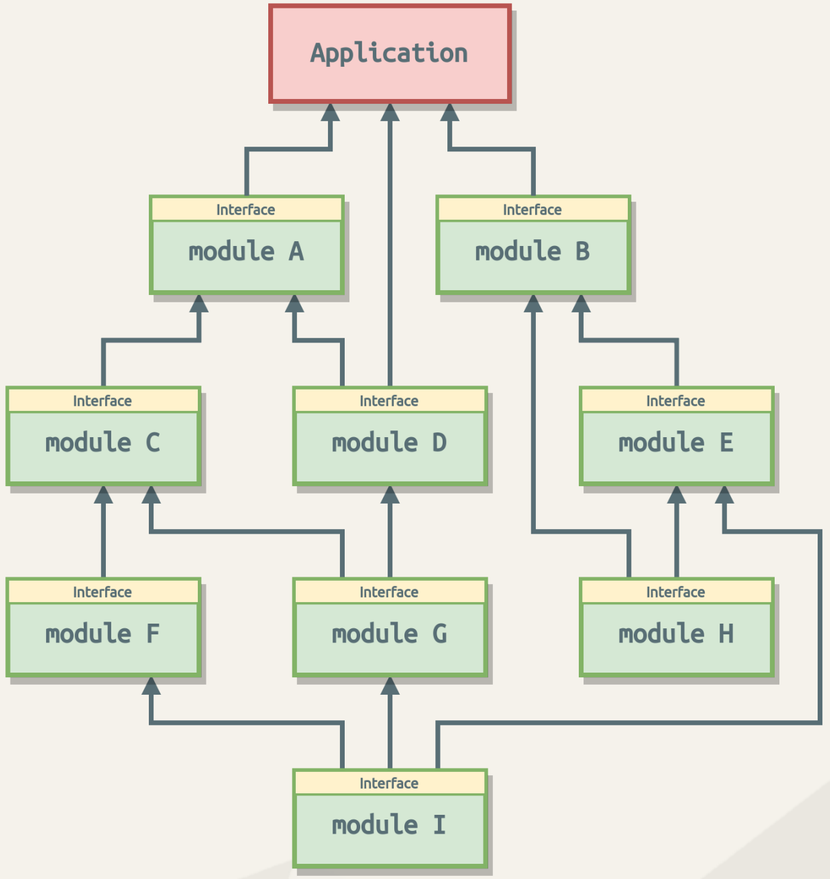
(MS VS2022 ships MSVC 19.33 in v17.4.3)

C++20 by example

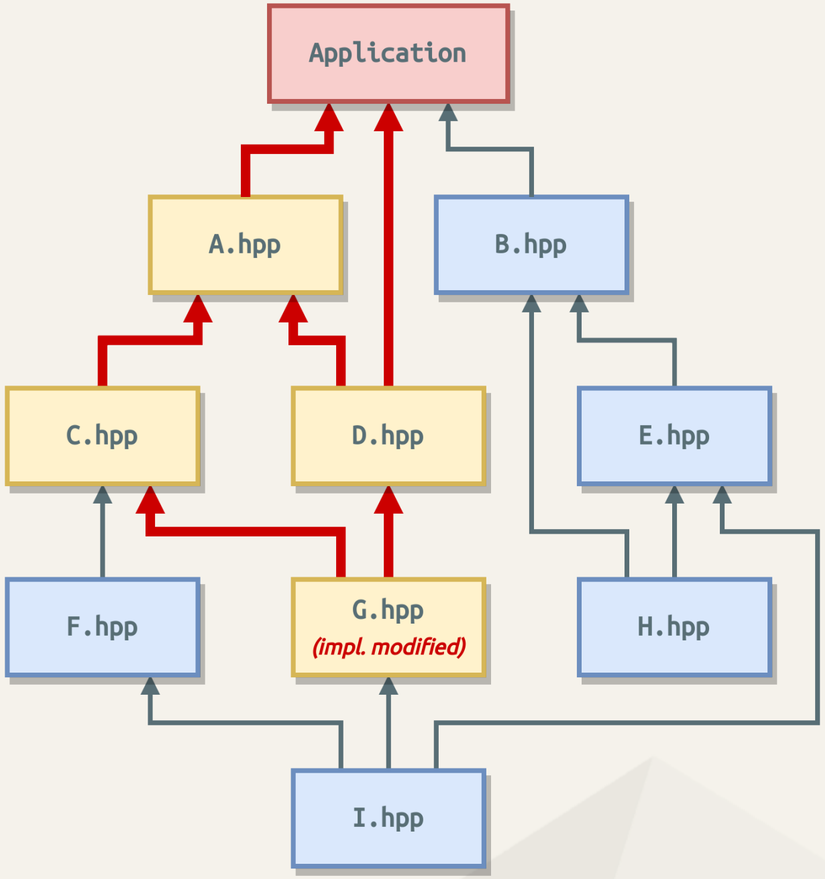
Modules

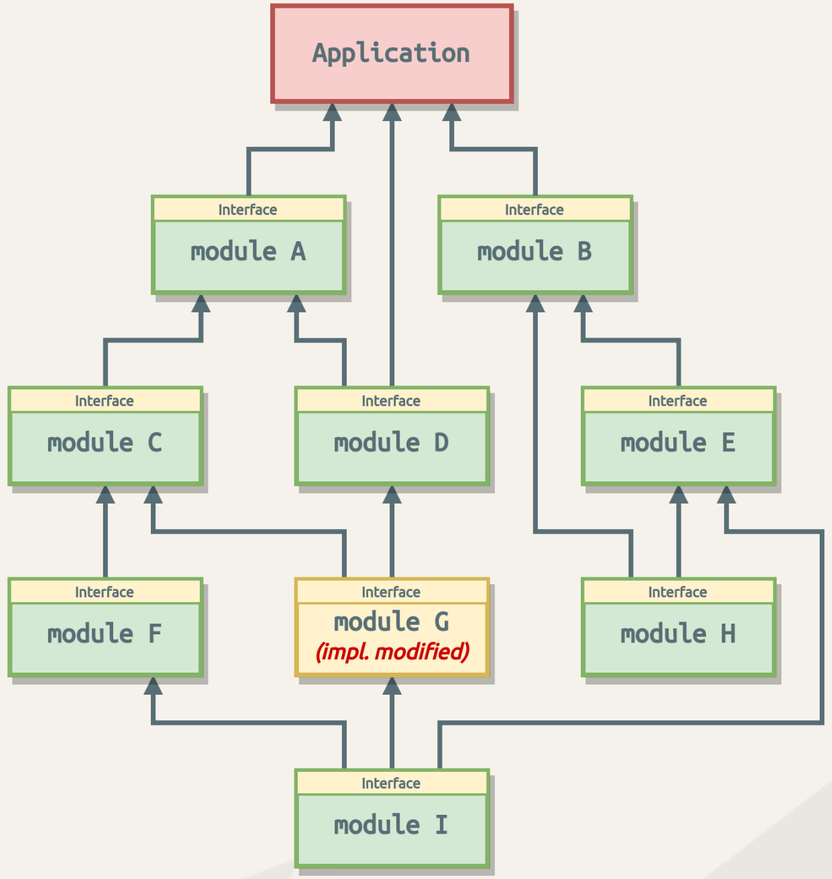
The problem modules solve





The problem modules solve





The simplest ever module

`fraction.cppm` (or `.ixx`):

```cpp
export module fraction;

template<typename Type>
concept arithmetic = std::integral<Type> || std::floating_point<Type>;

template<arithmetic Type>
export class Fraction final {
public:
  // ...
};

template<typename Type>
export std::ostream& operator<<(std::ostream& os, const Fraction<Type>& f) {
  os << "[" << f.numerator() << "/" << f.denominator() << "]";
  return os;
}
```

(there is no consensus on the extension `.ixx`/`.cppm` AFAIK)

`main.cpp`:

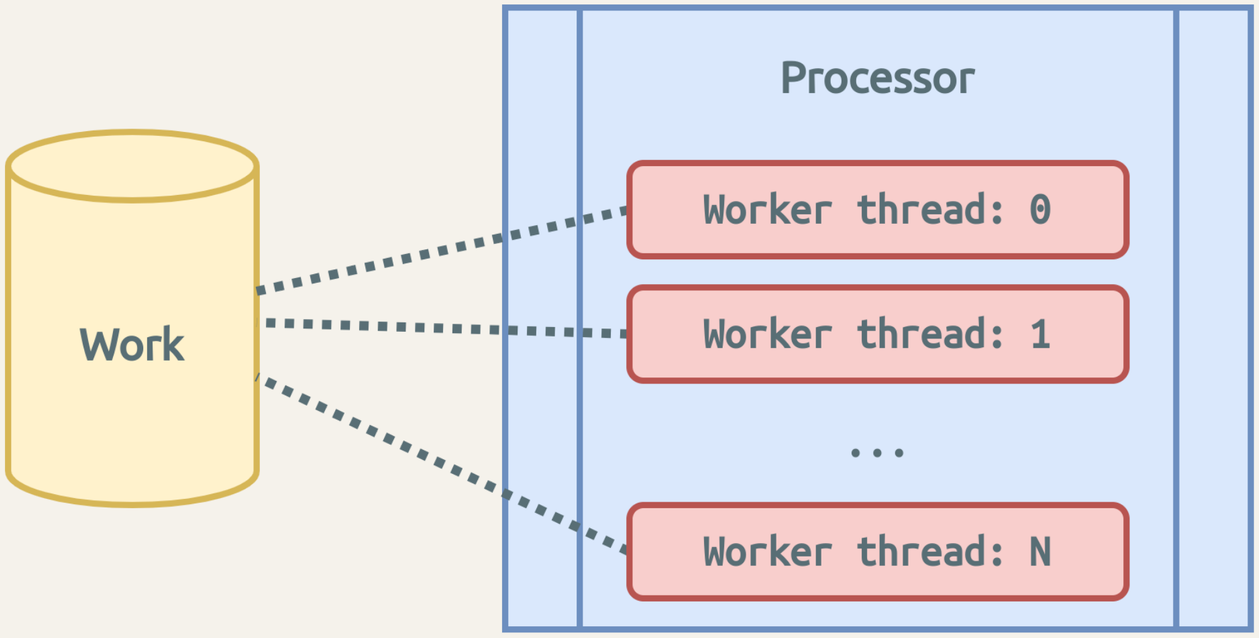
```cpp
import fraction;
import std; // Proposed in C++23.

int main() {
  Fraction f{22, 7};
  std::cout << "Fraction:" << f << '\n';
}
```

Not all build systems integrate support for **module**s still.. 😐

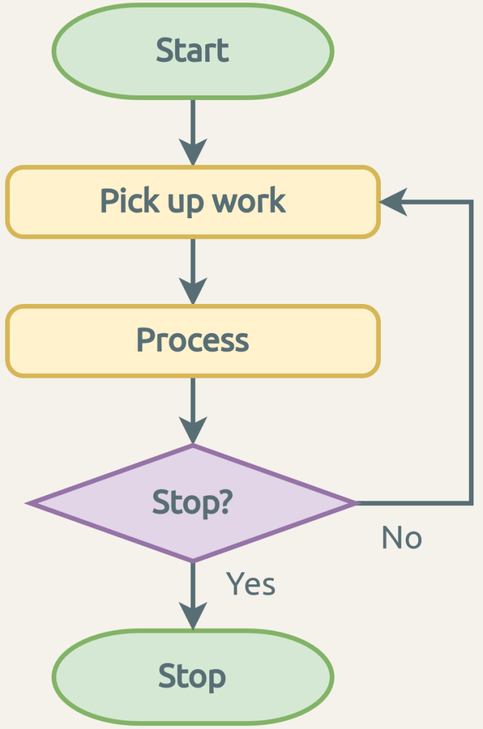
Concurrency

(with threads)

Example overview



Worker ow



Version 1: Setup + polling for work

```cpp
template<std::size_t NumWorkers>
class Processor {
  std::vector<std::jthread> workers_;

  void worker(std::stop_token stop_token);

public:
  Processor() {
    for (std::size_t i = 0; i < NumWorkers; i++) {
      workers_.emplace_back(
        std::bind_front(&Processor::worker, this));
    }
  }

  ~Processor() {
    for (auto& worker : workers_) {
      worker.request_stop();
    }
  }
};
```

```cpp
template<std::size_t NumWorkers>
void Processor<NumWorkers>::worker(std::stop_token stop_token) {
  while (!stop_token.stop_requested()) {
    //
    // ..poll for work and process it..
    //
  }
}
```

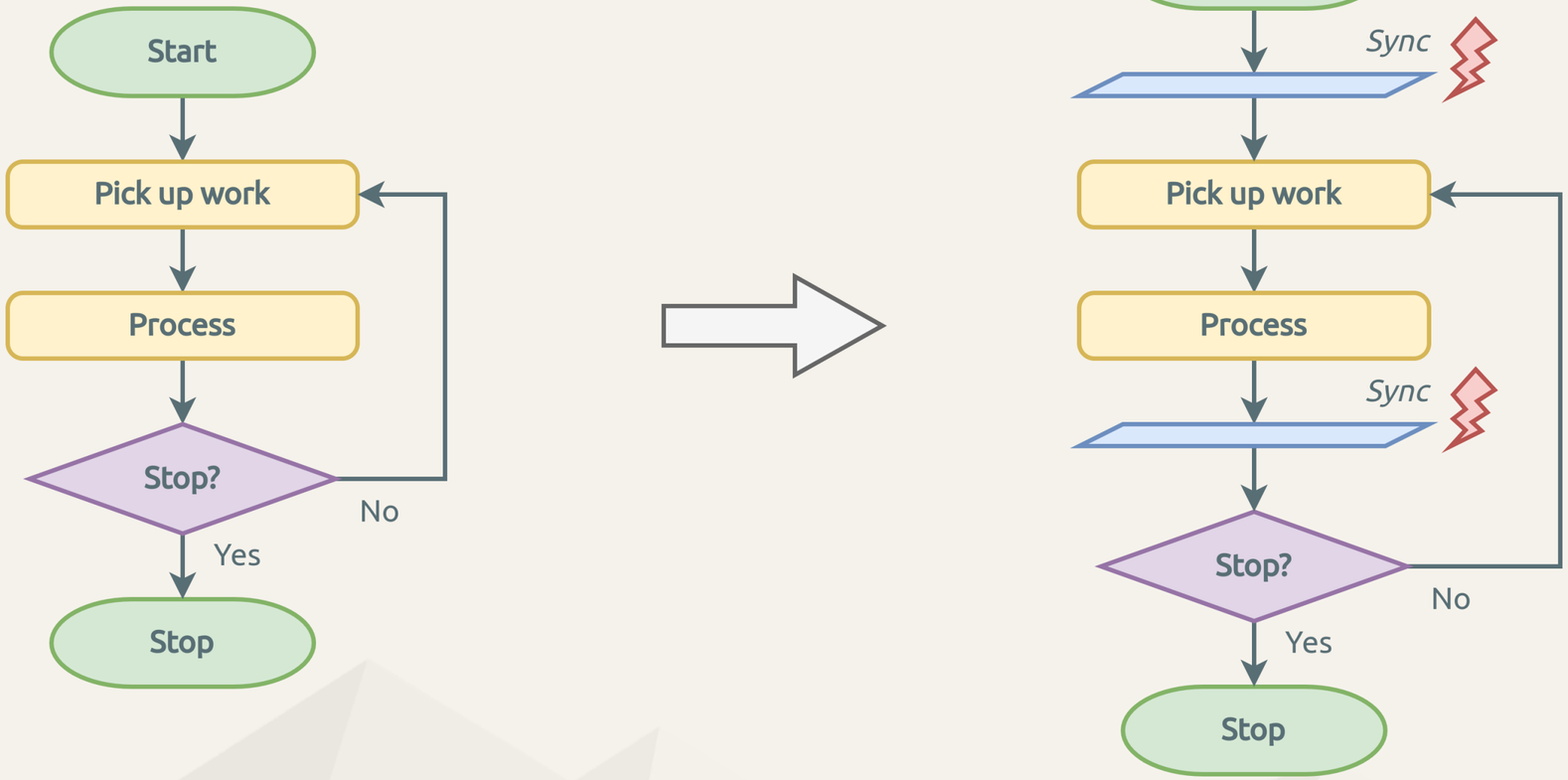
```cpp
int main() {
  Processor<4> proc;

  // ..exit when work is done..
}
```

Version 2: Properly waiting for work

```cpp
template<std::size_t NumWorkers>
class Processor {
  std::vector<std::jthread>   workers_;
  std::mutex                  work_mutex_;
  std::condition_variable_any work_cv_;

  void worker(std::stop_token stop_token);

public:
  Processor();

  ~Processor() {
    for (auto& worker : workers_) {
      worker.request_stop();
    }
  }
};
```

```cpp
template<std::size_t NumWorkers>
void Processor<NumWorkers>::worker(std::stop_token stop_token) {
  while (!stop_token.stop_requested()) {
    if (auto task{work_queue.get()}; task) {
      //
      // ..process work..
      //
    } else {
      std::unique_lock lock{work_mutex_};
      work_cv_.wait(lock, stop_token, [&]{ return !work_queue.empty(); });
    }
  }
}
```

Add worker synchronization points



Version 3: Add thread start synchronization point

```cpp
template<std::size_t NumWorkers>
class Processor {
  std::vector<std::jthread>   workers_;
  std::latch                  workers_start_{NumWorkers};
  std::mutex                  work_mutex_;
  std::condition_variable_any work_cv_;

  void worker(std::stop_token stop_token);

public:
  Processor();
  ~Processor();
};
```

A `std::latch` is a single-use synchronization point

```cpp
template<std::size_t NumWorkers>
void Processor<NumWorkers>::worker(std::stop_token stop_token) {

  // All threads will synchronize here:
  workers_start_.arrive_and_wait();

  while (!stop_token.stop_requested()) {
    if (auto task{work_queue.get()}; task) {
      //
      // ..process work..
      //
    } else {
      std::unique_lock lock{work_mutex_};
      work_cv_.wait(lock, stop_token, [&]{ return !work_queue.empty(); });
    }
  }
}
```

Version 4: Add post-work synchronization point

```cpp
template<std::size_t NumWorkers>
class Processor {
  std::vector<std::jthread>   workers_;
  std::latch                  workers_start_{NumWorkers};
  std::barrier<>              workers_sync_{NumWorkers};
  std::mutex                  work_mutex_;
  std::condition_variable_any work_cv_;

  void worker(std::stop_token stop_token);

public:
  Processor();
  ~Processor();
};
```

A `std::barrier` is a multi-use synchronization point

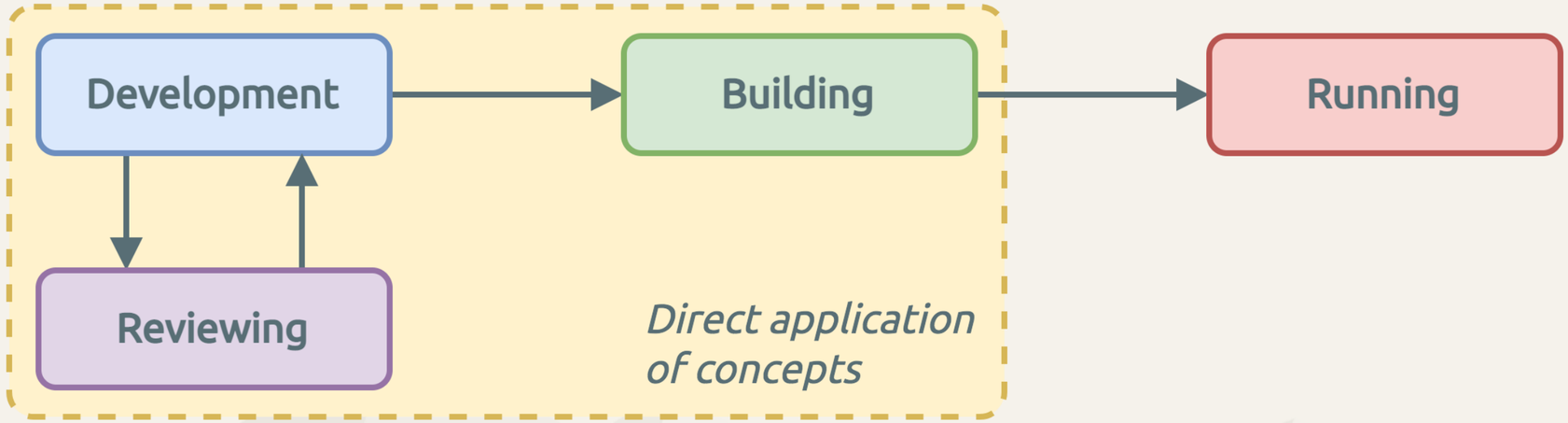
```cpp
template<std::size_t NumWorkers>
void Processor<NumWorkers>::worker(std::stop_token stop_token) {
  workers_start_.arrive_and_wait();

  while (!stop_token.stop_requested()) {
    if (auto task{work_queue.get()}; task) {
      //
      // ..process work..
      //

      workers_sync_.arrive_and_wait();

      //
      // ..post-process work..
      //
    } else {
      std::unique_lock lock{work_mutex_};
      work_cv_.wait(lock, stop_token, [&]{ return !work_queue.empty(); });
    }
  }
}
```

Compile-time validation with concepts

```
Kris van(Rens<>);
```

What concepts are about..

- Concepts are **named constraints**, predicates evaluated at compile-time,
- Concepts are **part of template interfaces** to improve *equational reasoning*,
- Concepts can be used to impose **syntactic and semantic constraints**.



Imposing type constraints

Use case: constrain class template type parameter.

```
SensorReading<float> r1; // OK.
SensorReading<int>   r2; // Compiler error.
```

Pre-C++20

```
#include <type_traits>

template<typename T>
class SensorReading {
  static_assert(std::is_floating_point_v<T>,
                "T must be a floating point type");
public:
  // ...
};
```

C++20

```
#include <concepts>

template<std::floating_point T>
class SensorReading {
public:
  // ...
};
```

Constraining (member-)functions

Use case: conditionally enable member function.

```
SensorReading<double> r1; // Has 'guess_snr'.
SensorReading<float>  r2; // Has no 'guess_snr'.

double snr1 = r1.guess_snr(); // OK.
double snr2 = r2.guess_snr(); // Compiler error!
```

Pre-C++20

```
#include <type_traits>

template<typename T>
class SensorReading {
public:
  template<typename U = T,
           typename = std::enable_if_t<std::is_same_v<double, U>>>
  double guess_snr() const;
};
```

C++20

```
#include <concepts>

template<std::floating_point T>
class SensorReading {
public:
  double guess_snr() const requires std::same_as<double, T>;
};
```

De ning custom concepts

Example: concept to test for type equivalence.

```cpp
class Sensor {
  // ...
};

class SpecificSensor : public Sensor {
  // ...
};

class MeasurementSystem {
public:
  operator Sensor() const { // Type conversion function.
    // ...
  }
};
```

All types can be used as a ‘Sensor’

```cpp
#include <concepts>

template<typename T>
concept sensor_like = std::same_as<Sensor, T>
                    || std::derived_from<T, Sensor>
                    || std::convertible_to<T, Sensor>;

template<sensor_like S>
void process(const S& s);
```

```cpp
Sensor            s1; // std::same_as        --> true.
SpecificSensor    s2; // std::derived_from   --> true.
MeasurementSystem s3; // std::convertible_to --> true.

process(s1);
process(s2);
process(s3);
```

`requires` expressions

Requires expressions can be used to **easily define powerful constraints**.

Testing for `get()`

```cpp
template<typename T>
concept has_get = requires (T t) { t.get(); };
//                ~~~~~~~~~~~~~~~~~~~~~~~~~ Requires expression.

template<typename Input> requires has_get<Input>
void process(const Input& i) {
  const auto value{i.get()};
  // ...
}
```

```cpp
class Sensor {
public:
  int get() const;
};

class SomeType {
public:
  // No 'get' function here..
};
```

```cpp
Sensor   x1;
SomeType x2;

process(x1); // OK.
process(x2); // Compiler error.
```

`requires` expressions

Example: arithmetic key/value mapping type definition.

```cpp
template<typename T>
concept key_value_store = requires (T x, int i) {
  { x.get()      };
  { x.get()      } noexcept;
  { x.get()      } -> std::same_as<std::pair<int, std::string>>;
  { x.get_val(i) } -> std::same_as<std::string>;
  { x + x        } -> std::same_as<T>;
} && std::equality_comparable<T>;

template<key_value_store Store>
void process(const Store& s) {
  const auto& [key, value] = s.get();

  // ...
}
```

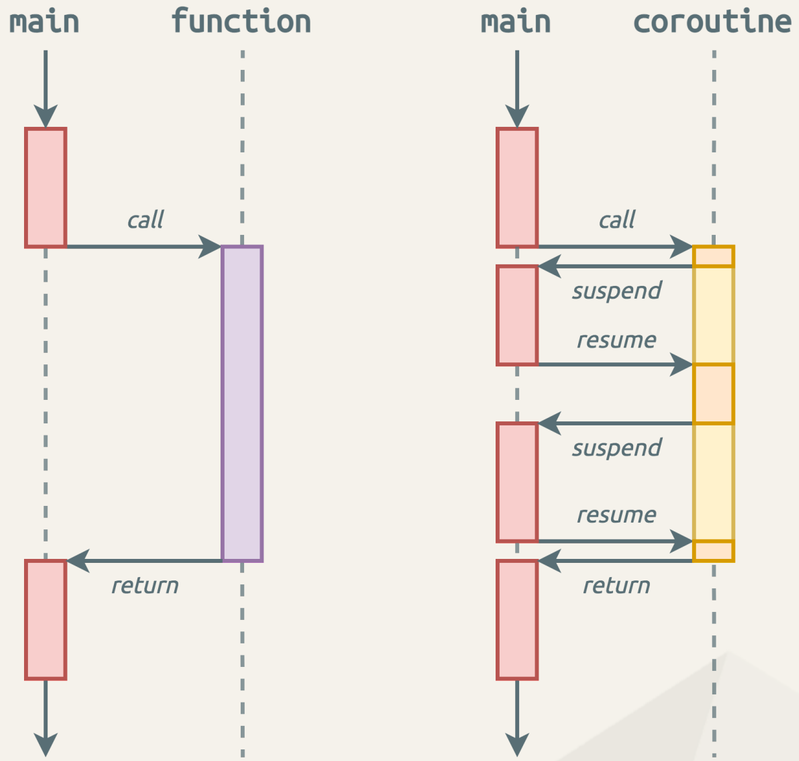
```cpp
class Mapping {
public:
  using KeyValue = std::pair<int, std::string>;

  [[nodiscard]] KeyValue get() const noexcept;
  [[nodiscard]] std::string get_val(int index) const;

  Mapping operator+(const Mapping&);

  bool operator<=>(const Mapping&) const = default;
};
```

```cpp
Mapping m1, m2;

process(m1 + m2);
```

Coroutines

What are coroutines?

Coroutines are **functions that can suspend execution to be resumed later**



Simple example

```
async_worker<void> read() {
  float value = co_await async_read();
  // ...
}
```

```
std::future<float> async_read() {
  // ..read 'value' from some source..
  co_return value;
}
```

async_read requires implementation of std::coroutine_traits:

```
template<typename T, typename... Args>
struct std::coroutine_traits<std::future<T>, Args...> {
  // ...
};
```

Basic implementation of a coroutine return type based on `std::future`:

```cpp
template<typename T, typename... Args>
struct std::coroutine_traits<std::future<T>, Args...> {
  struct promise_type : std::promise<T> {
    promise_type() = default;

    std::future<T> get_return_object() noexcept {
      return this->get_future();
    }

    void unhandled_exception() noexcept {
      this->set_exception(std::current_exception());
    }

    static std::suspend_never initial_suspend() noexcept {
      return {};
    }

    static std::suspend_never final_suspend() noexcept {
      return {};
    }

    template<std::convertible_to<T> U>
    void return_value(U&& value) noexcept(std::is_nothrow_constructible_v<T, decltype(std::forward<U>(value))>) {
      this->set_value(std::forward<U>(value));
    }
  };
};
```

Associated type `promise_type`

Each coroutine has a `promise_type`:

```cpp
struct promise_type {

  /****************** Mandatory interface ******************/
  promise_type() = default; // Option (A) for c'tor.
  promise_type(args...);    // Option (B) for c'tor.

  auto get_return_object();
  void unhandled_exception();
  auto initial_suspend();
  auto final_suspend() noexcept;

  /******************* Optional interface ******************/
  void return_value(auto expr); // Option (A) for result.
  void return_void();           // Option (B) for result.

  auto yield_value(auto expr);
  auto get_return_object_on_allocation_failure();
  auto await_transform(auto expr);
  void* operator new(std::size_t size, args...);
  void operator delete(void* ptr, std::size_t size);
};
```

Customizable interfaces:

- Implement asynchronous tasks,
- Implement *awaitable types*,
- Implement *generators*,
- …

No standard library support yet 😐

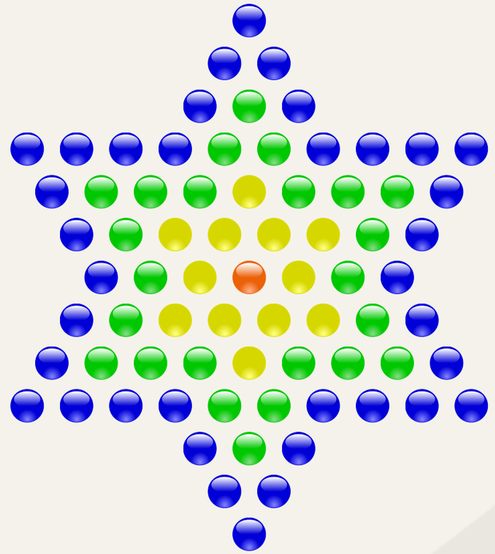
Range-enabled coroutine return type `generator`:

(Code @ Compiler Explorer)

```
struct coro_deleter {
  template<typename Promise>
  void operator()(Promise* promise) const noexcept {
    if (auto handle = std::coroutine_handle<Promise>::from_promise(*promise); handle) {
      handle.destroy();
    }
  }
};

template<typename T>
using promise_ptr = std::unique_ptr<T, coro_deleter>;

template<typename T>
class [[nodiscard]] generator {
public:
  using value_type = std::remove_reference_t<T>;
  using reference  = std::conditional_t<std::is_reference_v<T>, T, const value_type&>;
  using pointer    = const value_type*;

  struct promise_type {
    pointer value;

    static std::suspend_always initial_suspend() noexcept {
      return {};
    }

    static std::suspend_always final_suspend() noexcept {
```

Star numbers:

1 13 37 73 121 181 253 337 433 ..



(source: Wikipedia, CC BY-SA 4.0)

Improving algorithms using ranges

Ranges

A “range” is an **abstraction over iterators**, enabling a *functional style*.

Classic algorithms

```
1 std::vector vec{60, 4, 84, 14, 79, 51, 93, 25, 59};
2
3 std::sort(vec.begin(), vec.end());
```

We supply a pair of iterators.

C++20 ranges

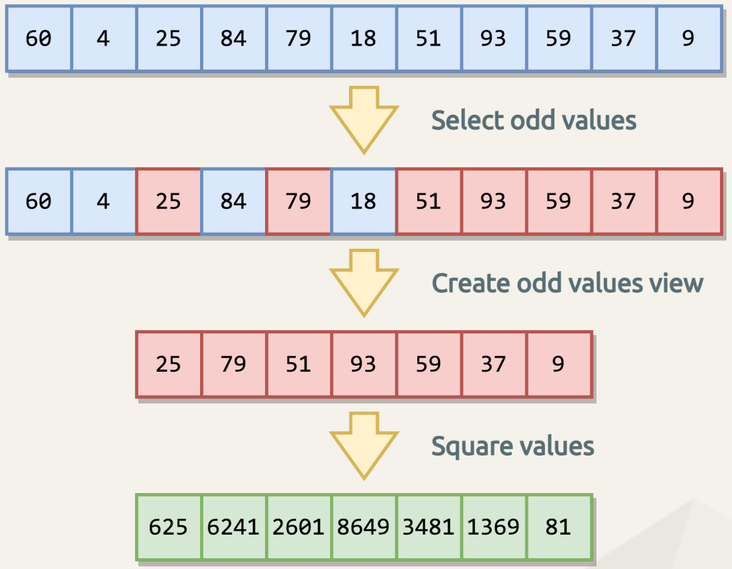
```
1 std::vector vec{60, 4, 84, 14, 79, 51, 93, 25, 59};
2
3 std::ranges::sort(vec);
```

We supply a single range.

One of the main motivations for ranges is **correctness at construction**.

Range pipelines and views

A range view is a **lightweight (sub-)range representation** for ad-hoc processing.




```cpp
const std::vector vec{60, 4, 84, 14, 79, 51, 93, 25, 59, 37, 9};

const auto is_odd = [](int value) -> bool { return value % 2;     };
const auto square = [](int value) -> int  { return value * value; };
const auto print  = [](int value) { std::cout << value << '\n'; };

// Shorthand names to remove noise (advice: don't apply 'using namespace').
namespace rg = std::ranges;
namespace rv = std::views;
```

Business logic:

```cpp
rg::for_each(vec | rv::filter(is_odd) | rv::transform(square), print);
```

Ranges, views and projections

Range views have *reference semantics* and use **lazy evaluation**.

```cpp
enum class EventType { Error, Info };

struct Event {
  unsigned long id{};
  EventType     type{EventType::Info};
  std::string   message;
};

const std::vector<Event> system_log = {
  {0, EventType::Info,  "motor1: start"},
  {1, EventType::Info,  "motor2: start"},
  {2, EventType::Info,  "motor1: OK"   },
  {3, EventType::Error, "motor2: FAIL" },
  {4, EventType::Info,  "motor1: stop" },
  // ...
};
```

```cpp
namespace rg = std::ranges;
namespace rv = std::views;

void process_log(rg::input_range auto log); // C++20 abbreviated function syntax.

const auto is_info  = [](const Event& e) { return (e.type == EventType::Info);  };
const auto is_error = [](const Event& e) { return (e.type == EventType::Error); };
```

```cpp
for (auto& e : system_log | rv::filter(is_error)) {
  std::cerr << std::format("Error event: ({}: '{}')\n", e.id, e.message);
}

process_log(system_log);                              // Feed the whole range.
process_log(system_log | rv::take_while(is_info));    // Feed up to first non-info.
process_log(system_log | rv::take(3));                // Feed first three items.
process_log(system_log | rv::drop(2) | rv::take(3)); // Feed third to fifth item.
```

Ranges, views and projections

Projections can be used in algorithms to **transform their behavior**.

```cpp
enum class EventType { Error, Info };

struct Event {
  unsigned long id{};
  EventType     type{EventType::Info};
  std::string   message;
};

std::vector<Event> system_log = {
  {0, EventType::Info,  "motor1: start"},
  {1, EventType::Info,  "motor2: start"},
  {2, EventType::Info,  "motor1: OK"   },
  {3, EventType::Error, "motor2: FAIL" },
  {4, EventType::Info,  "motor1: stop" },
  // ...
};
```

Projections for `sort` transformation:

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

rg::sort(system_log);                       // Default: take default comparison op.
rg::sort(system_log, {}, &Event::id);       // Sort using field 'id'.
rg::sort(system_log, {}, &Event::message); // Sort using field 'message'.
```

Many other algorithms take projections.

The Range-v3 library

The ranges library is **fully implemented in Range-v3** and can be used today.

```cpp
enum class EventType { Error, Info };

struct Event {
  unsigned long id{};
  EventType     type{EventType::Info};
  std::string   message;
};

const std::vector<Event> system_log = {
  {0, EventType::Info,  "motor1: start"},
  {1, EventType::Info,  "motor2: start"},
  {2, EventType::Info,  "motor1: OK"   },
  {3, EventType::Error, "motor2: FAIL" },
  {4, EventType::Info,  "motor1: stop" },
  // ...
};
```

```cpp
namespace rg = ranges;
namespace rv = ranges::views;   // Lazy views of ranges.
namespace ra = ranges::actions; // Eager in-place range modifications.

void process_log(rg::input_range auto log);

const auto is_error = [](const Event& e) { return (e.type == EventType::Error); };
```

Use an intermediate for modification:

```cpp
process_log(system_log | rv::take(50)
                       | rv::filter(is_error)
                       | rg::to<std::vector>() // <-- Create a mutable intermediate.
                       | ra::sort(std::less<>{}, &Event::id));
```

More notable stuff

Array view type `std::span`

A **reference *value type*** to any contiguous sequence of values

Can be used on `std::vector`, `std::array` and C-style arrays (including mutation)

```cpp
void process_region(std::span<int> region) {
  for (int &value : region) {
    // Round up to the nearest decade.
    value = static_cast<int>(10.0f * std::ceil(value / 10.0f));
  }
}
```

```cpp
std::array<int, 8> data{/* ... */};

// Process values with index 3..6 (i.e. items 4..7):
process_region({std::begin(data) + 3, 4});

// Process the whole array:
process_region({data});
```

`std::span` can also be fixed-size by using `std::span<Type, Size>`

String formatter `std::format`

```
float       real1{3.141592654f};
float       real2{2.718281828f};
int         natural{82649};
std::string name{"Gideon Mantell"};
```

Desired print result:

```
Values: [3.141593, 003, 2.7183, 82649, 0x142d9, Gideon Mantell]
```

I/O streams

```
1 // Print and format using I/O streams:
2 std::cout << "Values: ["
3           << std::setprecision(7) << real1 << ", "
4           << std::setw(3) << std::setfill('0') << std::setprecision(0) << real1 << ", "
5           << std::setprecision(5) << real2 << ", "
6           << natural << ", "
7           << std::hex << "0x" << natural << ", "
8           << name << "]\n";
```

`std::format`

```
1 // Print using I/O streams, formatting with std::format:
2 std::cout << std::format("Values: [{:.7}, {:03.0}, {:.5}, {}, 0x{:x}, {}]\n",
3                          real1, real1, real2, natural, natural, name);
```

Default comparisons and operator<=>

```cpp
struct DbEntry {
  std::uint64_t key;
  Uuid          uuid;

  struct Data {
    std::uint32_t id;
    std::string   value;
  } data;
};
```

```cpp
std::map<std::string, DbEntry> entries;
entries["alpha"] = DbEntry{
  .key  = 123UL,
  .uuid = "3d5d476e-f80c-4dec-a308-27cd190399e3",
  .data = {.id    = 0U,
           .value = "Io"}};
entries["beta"] = DbEntry{
  .key  = 456UL,
  .uuid = "0f090b6f-2d63-4a25-8438-5b211baf7d8e",
  .data = {.id    = 1U,
           .value = "Ganymede"}};
```

```cpp
const bool result = (entries["alpha"] < entries["beta"]);
```

```
error: no match for 'operator<'
  (operand types are 'std::map<std::__cxx11::basic_string<char>,
                               DbEntry>::mapped_type' {aka 'DbEntry'} and
                      'std::map<std::__cxx11::basic_string<char>,
                               DbEntry>::mapped_type' {aka 'DbEntry'})
  |   const bool result1 = (entries["alpha"] < entries["beta"]);
```

Implement custom **operator<**

or

use C++20 **default comparison ops**

Default comparisons and operator<=>

```cpp
struct DbEntry {
  std::uint64_t key;
  Uuid          uuid;

  struct Data {
    std::uint32_t id;
    std::string   value;

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

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

```cpp
std::map<std::string, DbEntry> entries;
entries["alpha"] = DbEntry{
  .key  = 123UL,
  .uuid = "3d5d476e-f80c-4dec-a308-27cd190399e3",
  .data = {.id    = 0U,
           .value = "Io"}};
entries["beta"] = DbEntry{
  .key  = 456UL,
  .uuid = "0f090b6f-2d63-4a25-8438-5b211baf7d8e",
  .data = {.id    = 1U,
           .value = "Ganymede"}};
```

```cpp
const bool result = (entries["alpha"] < entries["beta"]);
```

```
OK
```

Three-way comparison

```cpp
#include <compare>

const auto result = (entries["beta"] <=> entries["gamma"]);

if        (std::is_eq(result)) {
  // ...
} else if (std::is_gt(result)) {
  // ...
} else if (std::is_lt(result)) {
  // ...
}
```

Kris van(Rens<>);

Designated initializers

```
struct SystemId {
  Uuid        uuid;
  std::string name;
  std::string alias;
  bool        verified{};
};

void verify(SystemId&& id);
```

Pre-C++20

```
verify({"465ff086-92c8-43a9-854e-f41f13bac804",
        "Quantum vector shape interceptor",
        "QVSI",
        true});
```

C++20

```
verify({.uuid     = "465ff086-92c8-43a9-854e-f41f13bac804",
        .name     = "Quantum vector shape interceptor",
        .alias    = "QVSI",
        .verified = true});
```

Omitted fields are zero-initialized.

Initialization order must be correct.

Initialization in range-based for

Pre-C++20

```cpp
auto data = receive(); // Outside the scope of the loop..
for (auto& chunk : data.chunks) {
  if (!error_correct(chunk)) {
    log_error("Failed to correct chunk #{}", chunk.id);
  }
}
```

```cpp
std::size_t index = 0; // Outside the scope of the loop..
for (const auto& e : collection) {
  register_element(e, index);
  ++index;
}
```

C++20

```cpp
for (auto data = receive(); auto& chunk : data.chunks) {
  if (!error_correct(chunk)) {
    log_error("Failed to correct chunk #{}", chunk.id);
  }
}
```

```cpp
for (std::size_t index = 0; const auto& e : collection) {
  register_element(e, index);
  ++index;
}
```

Method `contains` for STL containers

Pre-C++20

```
1  PartId lookup(std::string_view name) {
2    static CacheMap cache;
3
4    auto result = cache.find(name);
5    if (result == cache.end()) {
6      auto id = lookup_expensive(name);
7      cache[name] = id;
8      return id;
9    }
10
11   return result->second;
12 }
```

C++20

```
1  PartId lookup(std::string_view name) {
2    static CacheMap cache;
3
4    if (!cache.contains(name)) {
5      auto id = lookup_expensive(name);
6      cache[name] = id;
7      return id;
8    }
9
10   return cache[name];
11 }
```

Immediate functions with consteval

```cpp
#include <array>

using Lut = std::array<int, 10>;

consteval Lut init_lut() { // Always evaluated at compile-time when called.
  Lut result;

  int a{0}, b{1};
  for (auto &value : result) {
    value = a;
    a = b;
    b += value;
  }

  return result;
}

// constinit will ensure compile-time initialization of the variable:
constinit Lut lut{init_lut()}; // 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

int main() {
  return lut[8];
}
```

Binary output

```asm
main:
        mov     eax, DWORD PTR lut[rip+32]
        ret
lut:
        .long   0
        .long   1
        .long   1
        .long   2
        .long   3
        .long   5
        .long   8
        .long   13
        .long   21
        .long   34
```

An outlook to C++23

C++23 in one slide

C++23: not nearly as huge as C++20, but still has a lot to offer!

- Standard library modules (`import std`),
- Result type `std::expected`,
- Multidimensional array view `std::mdspan`,
- Proper printing support with `std::print`,
- Associative containers `std::flat_map` and `std::flat_set`,
- Coroutine generator `std::generator`,
- Stack trace evaluation with `std::stacktrace`,
- Lots of range library extensions.

Multidimensional array view `mdspan`

```cpp
namespace rg = std::ranges;
namespace rv = std::ranges::views;

const auto data = rv::iota('a') | rv::take(12) | rg::to<std::vector>; // 'a'..'l'

auto view1 = std::mdspan{data.data(), 3, 4}; // Single plane, 3x4 elements.

for (std::size_t x{0}; x != view1.extent(0); x++) {
  for (std::size_t y{0}; y != view1.extent(1); y++) {
    std::print(" {}", view1[x, y]);
  }
  std::print("\n");
}

auto view2 = std::mdspan{data.data(), 3, 2, 2}; // Three planes, 2x2 elements.

for (std::size_t x{0}; x != view2.extent(0); x++) {
  std::print("Plane {}:\n", x);
  for (std::size_t y{0}; y != view2.extent(1); y++) {
    for (std::size_t z{0}; z != view2.extent(2); z++) {
      std::print(" {}", view2[x, y, z]);
    }
    std::print("\n");
  }
}
```

view1:

```
 a b c d
 e f g h
 i j k l
```

view2:

```
Plane 0:
 a b
 c d
Plane 1:
 e f
 g h
Plane 2:
 i j
 k l
```

Result type `std::expected`

Exceptions

```cpp
float read_temperature() {
  // ..may throw an exception!..
}

void display_temperature() {
  const float t = read_temperature();
  std::print("Temperature: {} °C", t);
}

display_temperature();
```

Exception handling?

C++23 `std::expected`

```cpp
using ReadResult = std::expected<float, std::string>;

ReadResult read_temperature() {
  // ...
}

void display_temperature() {
  if (const auto t = read_temperature(); t) {
    std::print("Temperature: {} °C", t.value());
  } else {
    std::print("Error ({})", t.error());
  }
}

display_temperature();
```

Coroutine generator `std::generator`

Standard library support for **synchronous coroutine generators**

```cpp
#include <cstdint>
#include <generator>
#include <print>
#include <ranges>

std::generator<std::uint64_t> star(std::uint64_t n = 0L) {
  while (true) {
    n++;
    co_yield 6 * n * (n - 1) + 1;
  }
}

int main() {
  for (auto i : star() | std::views::take(20)) {
    std::print("{}", i);
  }
  std::print("\n");
}
```

```
1 13 37 73 121 181 253 337 433 541 661 793 937 1093 1261 1441 1633 1837 2053 2281
```

Range extensions

Many useful range views have been added, including **range conversion**

```cpp
namespace rg = std::ranges;
namespace rv = std::ranges::views;

using namespace std::literals;
std::vector brands = {"Gibson"sv, "Fender"sv, "Ibanez"sv, "Martin"sv};
std::vector types  = {"Les Paul"sv, "Telecaster"sv, "PGM"sv, "D-45"sv};

const auto result = rv::zip(brands, types) | rg::to<std::vector>();

for (std::size_t index{0}; auto [a, b] : result) {
  std::print("{}: {} {}\n", index++, a, b);
}
```

```
0: Gibson Les Paul
1: Fender Telecaster
2: Ibanez PGM
3: Martin D-45
```

std::flat_set and std::flat_map

"Flat storage" associated containers (**faster iteration + efficient memory use**)

```cpp
enum class Brand { Gibson, Fender, Ibanez, Martin };

using KeyValue = std::pair<Brand, std::string_view>;
const std::initializer_list<KeyValue> list{{Brand::Gibson, "Les Paul"sv  },
                                           {Brand::Fender, "Telecaster"sv},
                                           {Brand::Ibanez, "PGM"sv       },
                                           {Brand::Martin, "D-45"sv      }};

std::map<Brand, std::string_view>      map1;
std::flat_map<Brand, std::string_view> map2;

for (const auto& kv : list) {
  map1.insert(kv); // Insertion is cheaper for std::map.
  map2.insert(kv); // Insertion in std::flat_map requires copyability + movability.
}

// Verify order-equality:
for (auto kv1 = map1.cbegin(); const auto& [key2, value2] : map2) {
  const auto& [key1, value1] = *kv1++;
  assert((key1 == key2) && (value1 == value2));
}
```

End

Thank you 😀

🔗 github.com/krisvanrens