A. Spaceship operator and strong / weak / partial ordering
B. Range library
C. Concepts and Constraints
D. Coroutine and Generator *cooperative scheduling*
E. Cooperative cancellation with `std::stop_source` and `std::jthread` *cooperative cancellation*
F. Module

A. Spaceship operator and strong / weak / partial ordering

(please read https://brevzin.github.io/c++/2019/07/28/comparisons-cpp20)

Lets take a look at the defficiency of operators before C++20.

Three problems

1. When we declare a new struct `T`, all 6 operators (`==`, `!=`, `<`, `>`, `<=` and `>=`) are :

- non-reversed, if we define `bool T::operatorOP(int)`, it doesn't imply `bool operatorOP(int,T)`, it must be defined explicitly
- non-rewritten, if we define `bool T::operator==(int)`, it doesn't imply `bool T::operator!=(int)`, it must be defined explicitly
- ► henever define a new `struct T`, which can be constructed from `int`, we have to offer 18 operators in total :

```
// member functions                      // member functions                 // global functions for heterogenous comparison
bool T::operator==(const T& rhs)         bool T::operator==(int)             bool operator==(int, const T&)
bool T::operator!=(const T& rhs)         bool T::operator!=(int)             bool operator!=(int, const T&)
bool T::operator> (const T& rhs)         bool T::operator> (int)             bool operator> (int, const T&)
bool T::operator>=(const T& rhs)         bool T::operator>=(int)             bool operator>=(int, const T&)
bool T::operator< (const T& rhs)         bool T::operator< (int)             bool operator< (int, const T&)
bool T::operator<=(const T& rhs)         bool T::operator<=(int)             bool operator<=(int, const T&)
```

As these operators are all independent, there is no way to guarantee the following, resulting in contradicting operator definitions.

```
(t0==t1 && t0!=t1) == false
(t0==t1 || t0!=t1) == true
(t0==t1 || t0> t1) == (t0>=t1)
```

2. STL containers use the following idiom to generate different operators from user supplied `LESS_THAN` predicate, in order to avoid contradicting operator definitions. For example, we need to provide `LESS_THAN` predicate to `std::set` and `std::map`, however `EQUAL_TO` is needed inside the implementation of `std::map<KEY,VALUE>`, it is probably implemented as :

```
template<typename T> bool    MORE_THAN(const T& lhs, const T& rhs) { return  LESS_THAN(rhs,lhs); }
template<typename T> bool     EQUAL_TO(const T& lhs, const T& rhs) { return !LESS_THAN(lhs,rhs) && !MORE_THAN(lhs,rhs); }
template<typename T> bool NOT_EQUAL_TO(const T& lhs, const T& rhs) { return  LESS_THAN(lhs,rhs) ||  MORE_THAN(lhs,rhs); }
template<typename K, typename V> auto std::map::find(const K& key)
{
    // binary tree traversal using iter ...
    if (EQUAL_TO(key, iter->key)) return iter->value;
    // binary tree traversal using iter ...
}
```

The above idiom relies on the **trichotomy** assumption implying that one-and-only-one among `(lhs<rhs,lhs==rhs,lhs>rhs)` holds true, which is not always the case. For example, when `lhs=1.234` and `rhs=NaN`, then `(lhs<rhs,lhs==rhs,lhs>rhs)` should all be false. Besides, it implies that operators before C++20 is insufficient to handle different types of ordering, hence it should be enhanced.

3. Lexicographical order of aggregate is implemented as memberwise `LESS_THAN` comparisons, however this idiom invokes primitive operator on each member twice, which is inefficient.

```
struct T
{
    T0 m0; T1 m1; T2 m2; T3 m3;
    bool operator<(const T& rhs)
    {
        if (m0 < rhs.m0) return true; else if (m0 > rhs.m0) return false;
        if (m1 < rhs.m1) return true; else if (m1 > rhs.m1) return false;
        if (m2 < rhs.m2) return true; else if (m2 > rhs.m2) return false;
        if (m3 < rhs.m3) return true; else if (m3 > rhs.m3) return false;
        return false; // return false for lhs == rhs
    }
};
```

Solution step1 - New C++20 concepts

First of all, C++20 introduces the following concepts :

- spaceship operator `<=>` which returns one of the following ordering types (instead of merely a `bool`) ...
- strong ordering / weak ordering / partial ordering, which are defined as ...

```
// This is just my implementation, not necessarily true.
struct strong_order
{
    enum dummy { less = -1, equal = 0, greater = +1 };
    strong_order(dummy x) { m = x; }
    dummy m;
};

struct weak_order
{
    enum dummy { less = -1, equivalent = 0, greater = +1 };
    weak_order(dummy x) { m = x; }
    dummy m;
};

struct partial_order
{
    enum dummy { less = -1, equal = 0, greater = +1, unordered = 0xFF; };
    partial_order(dummy x) { m = x; }
    dummy m;
};

// ORDER_TYPE = strong_order or weak_order
template<typename ORDER_TYPE>
bool operator> (const ORDER_TYPE& ord, int x) { return ord.m >  x; }
bool operator==(const ORDER_TYPE& ord, int x) { return ord.m == x; }
bool operator< (const ORDER_TYPE& ord, int x) { return ord.m <  x; }
// specialization for partial_order
bool operator> (const partial_order& ord, int x) { if (ord.m == unordered) return false; else return (weak_order)(ord) >  x; }
bool operator==(const partial_order& ord, int x) { if (ord.m == unordered) return false; else return (weak_order)(ord) == x; }
bool operator< (const partial_order& ord, int x) { if (ord.m == unordered) return false; else return (weak_order)(ord) <  x; }
```

- together with the 6 operators, we now have 7, classified into 4 catergories, reversed and rewritten features are introduced

	equality	ordering	feature
primary	`==`	`<=>`	primary operator can be reversed (primary cannot be rewritten)
secondary	`!=`	`<,>,<=,>=`	secondary operator can be rewritten (secondary cant be reversed)

What is operator-reversed?

- *it is for primary operator only*
- *it is for heterogenous operator only (comparison of different types)*
- *if heterogenous operator* `bool operator==(const U& u, const T& t)` *is invoked and undefined*
  then compiler will evaluate it using existing `bool T::operator==(const U& u) const` *i.e. evaluate* `u==t` *as* `t==u`
- *if heterogenous operator* `strong_order operator<=>(const U& u, const T& t)` *is invoked and undefined*
  then compiler will evaluate it using existing `strong_order T::operator<=>(const U& u) const` *i.e. evaluate* `u<=>t` *as* `0 <=> (t<=>u)`
  Why's that?

	`t<=>u`	`0 <=> (t<=>u)`	undefined `u<=>t` is supposed to ...
if `u` is smaller	returns `greater`	returns `less`	returns `less`
if `u` equals to `t`	returns `equal`	returns `equal`	returns `equal`
if `u` is greater	returns `less`	returns `greater`	returns `greater`

What is operator-rewritten?

- *only secondary operator can be rewritten as primary operator*
- *if secondary operator* `bool T::operator!=(const T& rhs)` *is invoked and undefined*
  then compiler will evaluate it using existing `bool T::operator==(const T& rhs)` *i.e. evaluate* `lhs!=rhs` *as* `!(lhs==rhs)`
- *if secondary operator* `bool T::operator>=(const T& t)` *is invoked and undefined*
  then compiler will evaluate it using existing `bool T::operator<=>(const T& rhs)` *i.e. evaluate* `lhs>=rhs` *as* `(lhs<=>rhs) >= 0`
- *in general ... evaluate* `*this OP rhs` *as* `(*this<=>rhs) OP 0` *where* `OP` *is either one of* `<,>,<=,>=` *and* `OP` *is NOT* `==,!=`

Unlike template function, both operator-reversed and operator-rewritten do not generate new functions.

Solution step2 - What developer needs to define?
Secondly, whenever we declare a new `struct T`, which can be compared to an `int`, we need to offer 4 operators (instead of 18).

```
// member functions                        // member functions             // global functions for heterogenous comparison
bool T::operator==(const T& rhs)          bool T::operator==(int)
auto T::operator<=>(const T& rhs)         auto T::operator<=>(int)

// auto means either std::strong_ordering, std::weak_ordering, std::partial_ordering, for example :
auto T::operator<=>(const T& rhs)
{
    if      (this->m > rhs.m) return std::strong_ordering::greater;
    else if (this->m < rhs.m) return std::strong_ordering::less;
    else return std::strong_ordering::equal;
}
```

As far as I know (please check), the following is not guaranteed as they are defined in two separate operators :

```
    (lhs == rhs) == ((lhs <=> rhs) == 0)
```

User can then do one of the homogenous comparisons using :

```
T obj0(x0,y0,z0);
T obj1(x1,y1,z1);

// invoking operator==
if (obj0 == obj1) action(obj0, obj1);

// invoking operator<=>
auto result = (obj0 <=> obj1);
if (result == 0) action0(obj0,obj1);      if (result != 0) action1(obj0,obj1);
if (result >= 0) action2(obj0,obj1);      if (result >  0) action3(obj0,obj1);
if (result <= 0) action4(obj0,obj1);      if (result <  0) action5(obj0,obj1);
```

With secondary-operator-rewritten, we can do with better syntax, which triggers evaluation `(obj0<=>obj1) OP 0` :

```
// These statements cannot compile in C++17.
if (obj0 == obj1) action0(obj0,obj1);     if (obj0 != obj1) action1(obj0,obj1);
if (obj0 >= obj1) action2(obj0,obj1);     if (obj0 >  obj1) action3(obj0,obj1);
if (obj0 <= obj1) action4(obj0,obj1);     if (obj0 <  obj1) action5(obj0,obj1);
```

With primary-operator-reversed, we can do various heterogenous comparison :

```
if (obj0 == i) ...                        if (i == obj1) ...
if (obj0 != i) ...                        if (i != obj1) ...
if (obj0 >= i) ...                        if (i >= obj1) ...
if (obj0 >  i) ...                        if (i >  obj1) ...
if (obj0 <= i) ...                        if (i <= obj1) ...
if (obj0 <  i) ...                        if (i <  obj1) ...
```

How the three problems are solved?
Problem 1 is solved obviously as we able to simplify 18 operators into 4. Problem 2 is also solved as it does not reply on **trichotomy**. Problem 3 can be solved too as we can implement lexicographical order with less operator invocations :

```
struct T
{
    T0 m0; T1 m1; T2 m2; T3 m3;
    std::strong_ordering operator<=>(const T& rhs)
    {
        if (auto temp = m0 <=> rhs.m0; temp != 0) return temp;
        if (auto temp = m1 <=> rhs.m1; temp != 0) return temp;
        if (auto temp = m2 <=> rhs.m2; temp != 0) return temp;
        return m3 <=> rhs.m3;
    }
};
```

It is very likely that there are multi-operators matching an invocation. For example, user calls `t!=u`, there are 3 possible choices :

direct match candidate	→	`t.operator!=(u)`
with rewritten as `!(t==u)`	→	`!t.operator==(u)`
with rewritten as `!(t==u)` then reversed arguments `!(u==t)`	→	`!u.operator==(t)`
but never as (since no reversal for secondary operator `!=`)	×	`u.operator!=(t)`

Use of partial ordering

Comparison between double and NaN, returning partial ordering :

```
std::partial_ordering operator<=>(double lhs, double rhs)
{
    if (lhs == NaN || rhs == NaN) return std::partial_ordering::unordered;
    return ...;
}
```

Comparison between two rectangles, returning partial ordering :

```
struct rect
{
    double x;
    double y;
};

std::partial_ordering operator<=>(const RECT& lhs, const RECT& rhs)
{
    auto ans_x = (lhs.x <=> rhs.x);
    auto ans_y = (lhs.y <=> rhs.y);
    if (ans_x == ans_y) return (std::partial_ordering)(ans_x);
    return std::partial_ordering::unordered;
}
```

Difference between strong ordering and weak ordering

Strong ordering means if `x==y`, then it must imply `f(x)==f(y)` for all reasonable function `f`. Please note that `>=,>,<=,<` aren't considered. For example if we define a case sensitive word in lexicographical order. For all reasonable functions like `to_upper_case`, `to_lower_case` and `sum`, the property "`x==y` implies `f(x)==f(y)`" is fulfilled, so we can output `std::strong_ordering` in its comparison operator. Function `get_ptr` is not considered as it does not make sense in this context.

```
struct word
{
    std::strong_ordering operator<=>(const word& rhs)
    {
        for(int n=0; n!=size; ++n)
        {
            if (array[n] > rhs.array[n]) return std::strong_ordering::greater;
            if (array[n] < rhs.array[n]) return std::strong_ordering::less;
        }
        return std::strong_ordering::equal;
    }

    word to_upper_case(); // omit implementation for clarity
    word to_lower_case()
    {
        word output;
        for(int n=0; n!=size; ++n)
        {
            if (array[n] <= 'Z') output.array[n] = array[n]+32;
            else                 output.array[n] = array[n];
        }
        return output;
    }

    int sum()
    {
        int output = 0;
        for(int n=0; n!=size; ++n) output += array[n];
        return output;
    }

    auto get_ptr()
    {
        return array;
    }

    static const int size = 10;
    char array[size];
};
```

On the contrary, if the property "`x==y` implies `f(x)==f(y)`" cannot be fulfilled, then we should output `std::weak_ordering` and substitute `equal` by `equivalent`. If `struct word` above is modified to represent case insensitive word, then function `sum` does not fulfill the required property, hence we should output `std::weak_ordering` instead.

B. Range library

First of all, as range library usually involves some basic techniques, lets take a look at them first.

- class template argument deduction *CTAD*
- cascading operations on rvalue objects for pipe
- cast operator of lazy view as a rvalue container

The following works in `gcc`.

```
    // 1. CTAD to construct filter / transform object etc ...
    template<typename F> struct filter { F f; };
//  template<typename F> filter(F&) -> filter<F>; // CTAD works without explicit guide

    // All the following x,y,z work ...
    auto f = [](int x) { return x%3==0; };
    filter x([](int x) { return x%3==0; } );
    filter y(f);
    filter z(std::move(f));
    for(int n=0; n!=20; ++n) std::cout << x.f(n) << " " << y.f(n) << " " << z.f(n);

    // 2. cascading operations
    struct T
    {
        int m;
    };

    T fct(T&& x)
    {
        std::cout << "\nfct : " << x.m;
        return T{x.m+1};
    }

    T x(123);
    fct(fct(fct(fct(std::move(x))))); // prints 123, 124, 125 and 126

    // 3. cast operator returning std::vector
    struct T
    {
        operator std::vector<int>()
        {
            std::vector<int> output;
            output.push_back(123);
            output.push_back(124);
            output.push_back(125);
            output.push_back(126);
            return output;
        }
    };

    T t;
    std::vector<int> v = t; // invokes cast operator
    for(const auto& x:v) std::cout << "\n" << x; // prints 123, 124, 125 and 126
```

Range is a pair of iterators, so that :

- they come from the same container (protect against bug)
- they can be passed to algorithm as single argument, so cascading algorithms is possible
- range is the counterpart of range in Python, so it can be a list of numbers, pair of iterator etc ...

View is an ROI of container :

- defined by predicate (function returning bool)
- it is lazy, it does not do anything until dereference

```
    std::vector<std::uint32_t> vec0;
    for(std::uint32_t n=0; n!=50; ++n) vec0.push_back(rand()%100);

    // no calculation for pipe-operator, all calculations are delayed until copy-assignment to output vector
    auto vec1 = vec0 | std::views::filter   ([](const auto& x) { return x%2==1; } )
                     | std::views::transform([](const auto& x) { return x*2;    } )
                     | std::views::take(10);
    auto vec2 = vec0 | std::views::filter   ([](const auto& x) { return x%2==1; } )
                     | std::views::transform([](const auto& x) { return x*2;    } );

    decltype(vec0) vec3;
    for(const auto& x : vec0 | std::views::filter   ([](const auto& x) { return x%2==1; } )
                             | std::views::transform([](const auto& x) { return x*2;    } ))
    {
        vec3.push_back(x);
    }
```

A naive implementation of range library :

```
namespace alg
{
    // Just a wrapper supporting CTAD without explicity guide
    template<typename F> struct filter    { F impl; };
    template<typename F> struct transform { F impl; };
                         struct take      { std::uint32_t impl; };

    template<typename C> // C = container
    struct lazy_view
    {
    public:
        using T = typename C::value_type;
        lazy_view(C& container) : i0(container.begin()), i1(container.end()) {}

        // casting operator, invoke lazy calculation
        operator C()
        {
            // This is lazy calculation.
            C output;
            for(auto i=i0; i!=i1; ++i)
            {
                if (predicate(*i))
                {
                    output.push_back(transformation(*i));

                    ++taken_count;
                    if (taken_count == taken_total) return output;
                }
            }
            return output;
        }

        // we should offer begin(), end(), operator++(), operator==() and operator*() to support for-range

        typename C::iterator    i0;
        typename C::iterator    i1;
        std::function<bool(T)>  predicate;
        std::function<T(T)>     transformation;
        std::uint32_t           taken_count;
        std::uint32_t           taken_total;
    };

    template<typename C, typename F> lazy_view<C> operator | (C& lhs, filter<F> rhs)
    {
        lazy_view<C> output(lhs);
        output.predicate = rhs.impl;
        return output;
    }

    template<typename C, typename F> lazy_view<C> operator | (lazy_view<C> lhs, filter<F> rhs)
    {
        lhs.predicate = rhs.impl;
        return lhs;
    }

    template<typename C, typename F> lazy_view<C> operator | (lazy_view<C> lhs, transform<F> rhs)
    {
        lhs.transformation = rhs.impl;
        return lhs;
    }

    template<typename C> lazy_view<C> operator | (lazy_view<C> lhs, take rhs)
    {
        lhs.taken_count = 0;
        lhs.taken_total = rhs.impl;
        return lhs;
    }
}

std::vector<std::uint32_t> vec0;
for(std::uint32_t n=0; n!=50; ++n) vec0.push_back(rand()%100);

// My algo implementation does not support "auto" vec4.
std::vector<std::uint32_t> vec4 = vec0 | alg::filter   ([](const auto& x) { return x%2==1; } )
                                       | alg::transform([](const auto& x) { return x*2;    } )
                                       | alg::take(10);
```

C. Concepts and constraints

SFINAE syntax is not human-readable, its compilation error is cumbersome. Concepts comes to help. There are two steps :

- defining concepts
- applying concepts to template class or template function

The following 4 terminologies are different things :

- concepts
- constraints
- `requires` expression (with keyword `requires`) used in defining concepts
- `requires` clause (also with keyword `requires`) used in applying concepts to template class or template function

Convention used in naming traits and concepts :

- `std::is_xxx_able<T>` is a traits
- `std::is_xxx_able_v<T>` is a short cut to `std::is_xxx_able_t<T>::value`
- `std::is_xxx_able_t<T>` is a short cut to `typename std::is_xxx_able_t<T>::type`
- `std::   xxx_able<T>` is a concepts

Defining concepts

Concepts is a template *(ended with semicolon)*, formed by either :

- conjunction of constraints
- disjunction of constraints or
- atomic constraint

where atomic constraint is an expression returning prvalue type of `bool`, which can either be :

- instantiation of concepts
- value traits defined by class template (refer to `C++01 Fundamental.doc G2:2.1`)
- value traits defined by variable template (refer to `C++01 Fundamental.doc G2:2.3`)
- value traits mapped by variable template (refer to `C++01 Fundamental.doc G2:2.4`)
- `requires` expression

while `requires` expression is a function-liked syntax, taking optional arguments and has a braced-body :

- type requirement
- simple requirement
- compound requirement (braced simple requirement, with optional `noexcept` and optional return type)
- nested requirement (which is a `requires` clause)

```
// type requirement
template<typename C> concept container =
requires // no bracketed-argument is needed if no object is involved in the following definition ...
{
    typename C::value_type;
    typename C::iterator_type;
    typename C::const_iterator_type;
};

// simple requirement
template<typename T> concept addable =
requires(T x0, T x1)
{
    x0 + x1;
};

template<typename T, typename U> concept swappable =
requires(T x0, U x1)
{
    std::swap(std::forward<T>(x0), std::forward<U>(x1));
    std::swap(std::forward<U>(x1), std::forward<T>(x0));
};

// compound requirement
template<typename T, typename U> concept cross_addable =
requires(T x0, U x1)
{
    { x0 + x0 } -> std::same_as<T>;        // std::same_as<T>        is another concept with 2 template parameters.
    { x0 + x1 } -> std::convertible_to<T>; // std::convertible_to<T> is another concept with 2 template parameters.
};

template<typename T> concept hashable =
requires(T x)
{
    { std::hash<T>{}(x) } -> std::convertible_to<std::size_t>; // while {}-> is requires-expression syntax
};
```

```
// nested requirement
template<typename P> concept hashable_ptr =
requires (P ptr)
{
    typename P::value_type;                               // This is type     requirement.
    { *ptr } -> std::same_as<typename P::value_type>;     // This is compound requirement.
    requires hashable<P>;                                 // This is nested   requirement.
    requires cross_addable<P, std::size_t>;               // This is nested   requirement.
};
```

Here is an example trying to include all syntax (with conjunction, disjunciton) in single concepts.

```
// conjunction and disjunction should return prvalue type of bool as a whole
template<typename T> concept my_algo_requirement =
some_concepts<T>            &&                // instantiation of other concepts
std::is_xxx_able  <T>::value  &&              // value traits defined by class    template (C++01 G2:2.1)
std::is_yyy_able_v<T>       &&                // value traits defined by variable template (C++01 G2:2.3)
requires (T x)                                // require expression
{
    typename T::value_type;                   // require expression - type     requirement
      x.f0();                                 // require expression - simple   requirement
    { x.f1() } -> std::convertible_to<std::uint32_t>;  // require expression - compound requirement
    requires hashable<P>;                     // require expression - nested   requirement
};
```

Applying concepts

There are three ways to apply concepts to template class or template function :

- replace `typename` with one concepts (however it does ***NOT*** support conjunction nor disjunction)
- `requires` clause after template arguments (support conjunction nor disjunction)
- `requires` clause after function declaration (support conjunction nor disjunction)

► *when using the first approach, i.e. replacing `typename` with concepts :*
- *the concepts is instantiated with all-but-the-first template parameters*
- *hence it takes one less parameter than the definition of concepts, see example below*
- *the 1st template parameter in concept is removed and considered as the desired template parameter for the template class or function*

Here are two examples, using the three approaches respectively (the latter two are called `requires` clause) :

```
// Concepts with 1 template parameter
template<hashable T> void fct(const T&);                       // for this syntax, fill 1 less template para for hashable

template<typename T> requires hashable<T>                      // for this syntax, fill in all template para for hashable
void fct(const T&);

template<typename T>
void fct(const T&)   requires hashable<T>;                     // for this syntax, fill in all template para for hashable

// Concepts with 2 template parameters
template<std::derived_from<BASE> D> void fct(const D&);        // again, one less parameter in concepts

template<typename D> requires std::derived_from<D,BASE>
void fct(const D&);

template<typename D>
void fct(const D&)   requires std::derived_from<D,BASE>;

// Concepts with N template parameters
template<std::invocable<ARG0,ARG1,ARG2> F> void fct(const F&);   // again, one less parameter in concepts

template<typename F> requires std::invocable<F,ARG0,ARG1,ARG2>
void fct(const F&);

template<typename F>
void fct(const F&)   requires std::invocable<F,ARG0,ARG1,ARG2>;
```

where `requires` clause can be a conjunction or disjunction of atomic constraint, which can be (copied from previous part) :

- instantiation of concepts
- value traits defined by class template
- value traits defined by variable template (when using variable template in applying concept place it in bracket)
- value traits mapped by variable template (when using variable template in applying concept place it in bracket)
- `requires` expression
- ***OR*** any `constexpr` bool expression inside bracket

Please note that :

- `requires` expression contains an optional argument list and a braced body `{}`
- `requires` clause contains no braced body `{}`, but conjunction or disjunction of atomic constraint

Here is an example of applying `std::invocable`, it can accept function, member function, lambda and `std::function` :

```
#include<math.h>
template<typename R, typename G, typename B>
struct rgb
{
    // fct can be std::function   (which is slow)
    // fct can be lambda directly (without conversion to std::function)
    template<std::invocable<R,G,B> F>
    void to_grey(F&& fct) { std::cout << "\nRGB to grey " << fct(r,g,b); }

    // Why so complicated? Why not use this version? Slow binding to std::function?
    void to_grey2(std::function<double(R,G,B)>&& fct) { std::cout << "\nRGB to grey " << fct(r,g,b); }

    R r; G g; B b;
};

inline double rgb2grey(std::uint32_t r, std::uint32_t g, std::uint32_t b)
{
    return 0.6*r + 0.2*g + 0.2*b;
}

// *** Test program *** //
std::function f = [](std::uint32_t r, std::uint32_t g, std::uint32_t b){ return (r+g+b)/3; };

rgb<std::uint32_t, std::uint32_t, std::uint32_t> x(240,160,80);
x.to_grey (rgb2grey);     x.to_grey (f);
x.to_grey2(rgb2grey);     x.to_grey2(std::move(f));
x.to_grey ([](auto r, auto g, auto b){ return sqrt(r*r+g*g+b*b)/3; });
x.to_grey2([](auto r, auto g, auto b){ return sqrt(r*r+g*g+b*b)/3; });
```

More complicated examples for approach 2&3 (supporting conjunction and disjunction) :

```
// Woo!!! The following together is just a function prototype ...
template<typename T, typename U>
requires incrementable<T>         &&  // instantiation of other concepts
         decrementable<U>::value  &&  // value traits defined by class template
        (are_compatiable_v<T,U>)  &&  // value traits defined by variable template inside bracket

requires (T x, U y)                   // require expression
{
     typename T::value_type;
     { x.f() } -> std::same_as<U>;
     { y.f() } -> std::same_as<T>;
     requires hashable<T>;
     requires hashable<U>;
};
vod fct(const T&);
```

Implementation of two commonly used concept

In this section, we are going to implement two bivariate and multivariate concepts :

- `std::derived_from`
- `std::invocable`

```
// 1. std::is_base_of is value traits, while std::derived_from is concepts
// 2. std::is_base_of<base, derived>,  while std::derived_from<derived, base>
// 3. Please refer to C++04 Fundamental.doc Part F1 for implementation of std::is_base_of

template<typename DERIVED, typename BASE>
concept std::derived_from = std::is_base_of_v<BASE, DERIVED> &&
                            std::is_convertible_v<const volatile DERIVED*,
                                                  const volatile BASE*>;

template<typename FCT, typename... ARGS>
concept std::invocable = requires(FCT&& fct, ARGS&&... args)
{
    std::invoke(std::forward<FCT> (fct),
                std::forward<ARGS>(args)...);
};
```

Compare SFINAE with concepts

Compare the complicated syntax for `f` (which uses SFINAE) with the simple syntax for `g` (which uses concepts).

```
// Stub
class base {};
class derived : public base {};
class non_derived {};

// Method 1 - SFINAE
template<typename T, typename std::enable_if<!std::is_base_of<base, T>::value, int>::type = 0> void f(const T&) {}
template<typename T, typename std::enable_if< std::is_base_of<base, T>::value, int>::type = 0> void f(const T&) {}

// Method 2 - concepts
template<typename T>                  void g(const T&) {}
template<std::derived_from<base> T> void g(const T&) {}

// Test
derived x0;
non_derived x1;

f(x0);
f(x1);
g(x0);
g(x1);
```

Please read `C++04 Fundamental.doc part F1` for a simple implementation of value traits `is_base_of<BASE,DERIVED>`.

All in one test

Here is a complete test for concept syntax. If one of the 7 requirement is missing (commented out), the generic version is invoked.

```
struct A {};
struct B {};

struct type
{
    using value_type = void;                 // Requirement 1
    operator A() { return A{}; }             // Requirement 2
    operator B() { return B{}; }             // Requirement 3
    void          f() const {}               // Requirement 4
    std::uint32_t g() const { return 0UL; }   // Requirement 5
};

template<typename T> struct is_xxx        { static const bool value = false; };
template<>           struct is_xxx<type>  { static const bool value = true;  }; // Requirement 6

template<typename T> struct is_yyy        { static const bool value = false; };
template<>           struct is_yyy<type>  { static const bool value = true;  }; // Requirement 7
template<typename T> constexpr bool is_yyy_v = is_yyy<T>::value;

template<typename T> concept my_requirements =
std::convertible_to<T,A>  &&
is_xxx<T>::value          &&
is_yyy_v<T>               &&
requires (T x)
{
    typename T::value_type;
      x.f();
    { x.g() } -> std::same_as<std::uint32_t>;
    requires std::convertible_to<T,B>;
};

template<typename T> void f(const T& x) { std::cout << "\ngeneric version f"; }
template<typename T> void g(const T& x) { std::cout << "  generic version g"; }
template<typename T> void h(const T& x) { std::cout << "  generic version h"; }

template<my_requirements T>
void f(const T& x) { std::cout << "\nspecial version f"; }

template<typename T> requires my_requirements<T>
void g(const T& x) { std::cout << "  special version g"; }

template<typename T>
void h(const T& x) requires my_requirements<T>
{ std::cout << "  special version h"; }

// Test programme
type x;
f(x); g(x); h(x); // Special version is invoked. If one of the above requirements is missing, generic verion is invoked.
```

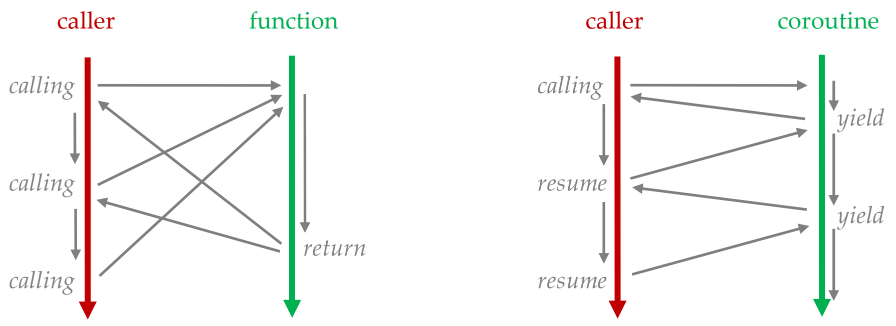
D.1 Coroutine introduction
3 concepts about coroutine :

- stackless function
- cooperative vs preemptive
- concurrency vs parallelism

Why stackless function?

- coroutine is a generalized function
- coroutine can be called / suspended / resumed / return
- function can be called / return only
- calling coroutine is like calling a function, which returns a handle
- resuming coroutine is like calling the cached handle
- it is about the *flow of execution* between caller and coroutine :
- when coroutine is suspended, it gives up the execution control (i.e. yielding cpu resource) to caller
- when coroutine is resumed by caller, it should start from where it left last time
- coroutine states (activation frame) live between suspension and resumption
- coroutine states (activation frame) are thus stored in heap (hence need `destroy` in C++), they cannot be nested in call-stack

Here is the *flow of execution* in function and coroutine for single thread :



How to suspend and resume coroutine?

- suspension is done inside coroutine (willingly, via various yield operators)
- `yield x` in python, or equivalently, `co_yield x` in C++20, which means output `x` and then suspend
- `x = (yield)` in python, or equivalently, `x = co_await awaitable` in C++20, which means suspended until input `x` is ready
- resumption is done outside coroutine (by caller, via coroutine handle)
- caller, on calling coroutine, should get a coroutine handle, which is returned from coroutine
- caller can then resume coroutine anytime by invoking the handle, starting from where it is last suspended
- up to this moment, there is no data transfer between caller and coroutine, we will see how it happens later

What are cooperative and preemptive scheduling?
Scheduling is needed because there are plenty of tasks to do, but only a limited amount of cpu resources are available. Cooperative scheduling means a task (running either a function or a coroutine) tries to cooperate with other tasks by giving up its cpu resources willingly, when it doesn't need them temporarily, for the sake of the whole system, hoping the others will do the same later. On the contrary, preemptive scheduling means a task will not cooperate with the other, unless a scheduler interrupts and orders it to do so. Cooperative scheduling is achieved by ***yielding***, which means surrender / grant / giving up cpu resources temporarily, for the sake of mutual benefit.

For C++20 :

- `std::this_thread::yield()` means yielding from `std::this_thread` to other threads
- `co_yield` means yielding from coroutine to caller, as coroutine has produced a product for the caller
- `co_await` means yielding from coroutine to caller, and coroutine is expecting for a product from the caller

What are concurrency and parallelism?
Multiple coroutines can be joint together to form a linear pipeline or a complicated *DAG* :

- `yield` in Python (or `co_yield` in C++) is for producer (source)
- `(yield)` in Python (or `co_await` in C++) is for consumer (sink)
- of course, a coroutine can be a producer and a consumer at the same time, forming intermediate node in *DAG*
- each node in the *DAG* yields some cpu resources (via cooperation, not by scheduler) and makes progress within finite time
- this apparent-parallelism by time-interleaving strategy is called concurrency (as opposed to real parallelism)

D.2 Python generator & coroutine
Lets start with Python coroutine, then move to C++20 coroutine.

For Python :
- function calling `yield x` automatically becomes generator
- function calling `x = (yield)` automatically becomes coroutine (in C++, both generator & coroutine are called coroutines)
- both generator and coroutine return handle implicitly (no need to specify in generator & coroutine definition)
- caller should cache handle for later resumption, resumption is invoked by `next` or `send` :

- `next` returns product `x` to caller

- `send` takes product `x` from caller

```
h = generator_that_produces()
while(cond) :
    x = next(h)

h = coroutine_that_consumes()
while(cond) :
    h.send(x)
```

Now lets try cascading multiple coroutines in two ways to achieve the same task (Brownian / stock model / moving average)
- cascading generators that `yield`
- cascading coroutines that `(yield)`

- no explicit `return` of coroutine handle is needed in following `def`

```
### Cascading generator approach ###
def gen_Brownian():
    zt = 0
    while True:
        yield zt
        zt = zt + (numpy.random.rand()-0.5) * 0.02

def gen_stock(mu, source):
    St = 0
    while True:
        yield St
        St = St + mu + next(source)

def gen_moving_average(source):
    count = 0
    total = 0
    while True:
        if count > 0 : yield total/count
        else : yield total
        count = count + 1
        total = total + next(source)

### Main ###
gen0 = gen_Brownian()
gen1 = gen_stock(0.1, gen0)
gen2 = gen_moving_average(gen1)
for n in range(30) : print(next(gen2))
print()

### Cascading coroutine approach ###
def crt_moving_average():
    count = 0
    total = 0
    while True:
        count = count + 1
        total = total + (yield)
        print(total/count)

def crt_stock(mu, sink):
    St = 0
    next(sink) # cannot sink.send() without next(sink)
    while True:
        St = St + mu + (yield)
        sink.send(St)

def crt_Brownian(sink):
    zt = 0
    next(sink)
    while True:
        zt = zt + ((yield)-0.5) * 0.02
        sink.send(zt)

### Main ###
crt2 = crt_moving_average()
crt1 = crt_stock(0.1, crt2)
crt0 = crt_Brownian(crt1)
next(crt0)
for n in range(30) : crt0.send(numpy.random.rand())
```

These two approaches are completely symmetrical.

Here are the steps involved in the two approaches :

```
                         |  Cascading       | Cascading       |
                         |  generator       | coroutine       |
-------------------------+-----------------+-----------------+
ultimate producer = main |                 |  rand   + send  |
-------------------------+-----------------+-----------------+
producer      = Brownian | rand + yield    | (yield) + send  |
intermediater = stock    | next + yield    | (yield) + send  |
consumer      = mov-aver | next + yield    | (yield) + print |
-------------------------+-----------------+-----------------+
ultimate consumer = main | next + print    |                 |
-------------------------+-----------------+-----------------+
composition of coroutine | Brown b         | M.Ave m         |
                         | Stock s(b)      | Stock s(m)      |
                         | M.Ave m(s)      | Brown b(s)      |
-------------------------+-----------------+-----------------+
```

Here is the timeline :

```
    cascading-generator             cascading-coroutine

    Brown   Stock   M.Ave   main    Brown   Stock   M.Ave
-------------------------------------------------------------
      |       |       |       v       |       |       |
      |       |       | next  v       |       |       |
      |       |       .<<<<<<<.       |       |       |
      |       |       v       |       |       |       |
      |       | next  v       | rand  |       |       |
      |       .<<<<<<<.       v       |       |       |
      |       v       |       v send  |       |       |
      | next  v       |       .>>>>>>>.       |       |
      .<<<<<<<.       |       |       v       |       |
rand  v       |       |       |       v send  |       |
      v yield |       |       |       .>>>>>>>.       |
      .>>>>>>>.       |       |       |       v       |
      |       v       |       |       |       v send  |
      |       v yield |       |       |       .>>>>>>>.
      |       .>>>>>>>.       |       |       |       v print
      |       |       v       |       |       | await v
      |       |       v yield |       |       .<<<<<<<.
      |       |       .>>>>>>>.       |       v       |
      |       |       |       v       | await v       |
      |       |       | print v       .<<<<<<<.       |
      |       |       |       |       v       |       |
      |       |       |       | await v       |       |
      |       |       |       .<<<<<<<.       |       |
      |       |       |       v       |       |       |
      |       |       |       v       |       |       |
```

D.3 C++20 coroutine

C++20 coroutine is so clumsy, there are 6 steps, 6 components and 3 yielding operators in total.
Reference : "My tutorial and take on C++20 coroutines" by David Mazieres.

As compared to python, in C++20 coroutine :

- there is no distinction between generator and coroutine, both are called coroutine
- C++20 decouples coroutine return from coroutine handle
  - coroutine does not return coroutine handle, instead it returns a customizable object, called future
  - coroutine handle cannot be customized, it is a predefined template class `std::coroutine_handle<promise_type>`
- C++20 decouples coroutine resumption from data transmission
  - `x = next(h)` in python is replaced by handle invocation `h()` returning `void`
  - `h.send(x)` in python is replaced by handle invocation `h()` taking nullary argument
  - hence the transmission of product `x` in both directions are **NOT** done via handle, instead done via an object called promise
- C++20 allows customization of `co_await` return value
  - `yield x` in python is retained in C++20 as `co_yield x`
  - `x = (yield)` in python is generalised in C++20 as `x = co_await a`
  - where `a` is an object called awaitable, which defines the type of `x` that `co_await` returns (i.e. data from caller to coroutine)
- C++20 requires owner of coroutine handle to destroy it explicitly when it is no longer needed (as its resources are in heap)
  - by calling `std::coroutine_handle<promise_type>::destroy()`

The 6 steps for coroutine in python and C++20 :

	Python	*C++20*	

creation of handle	`h = coroutine()`	`auto f = coroutine();`
		`auto h = f.get_handle();`
			`// where f is future`

resume generator handle	`x = h.next()`	`h(); x = p.get_x();`	`// where p is promise`
resume coroutine handle	`h.send(x)`	`p.set_x(x); h();`	
suspend in generator	`yield x`	`co_yield x;`	
suspend in coroutine	`x = (yield)`	`x = co_await a;`	`// where a is awaitable`
destroy handle	*no explicit destroy, its auto*	`h.destroy();`	

The 6 components in C++20 coroutine (there is no future / promise / awaitable in python, their roles in C++ are *highlighted*) :

• caller	
• coroutine	which can be suspended (by operator `co_yield` / `co_await` / `co_return`), resumed (by handle invocation)
• coroutine handle	which points to heap location where coroutine states are stored, can be invoked to resume coroutine
• future	which is the *return object* from coroutine to caller
• promise	which is the *factory* of future, the *owner* of product `x`
• awaitable	which is the *mediater* between caller and coroutine in flow of execution (see time line in `expt.0`) awaitable offers customization of the behaviour of `co_await` operator awaitable objects offered by STL library : `std::suspend_never` and `std::suspend_always`

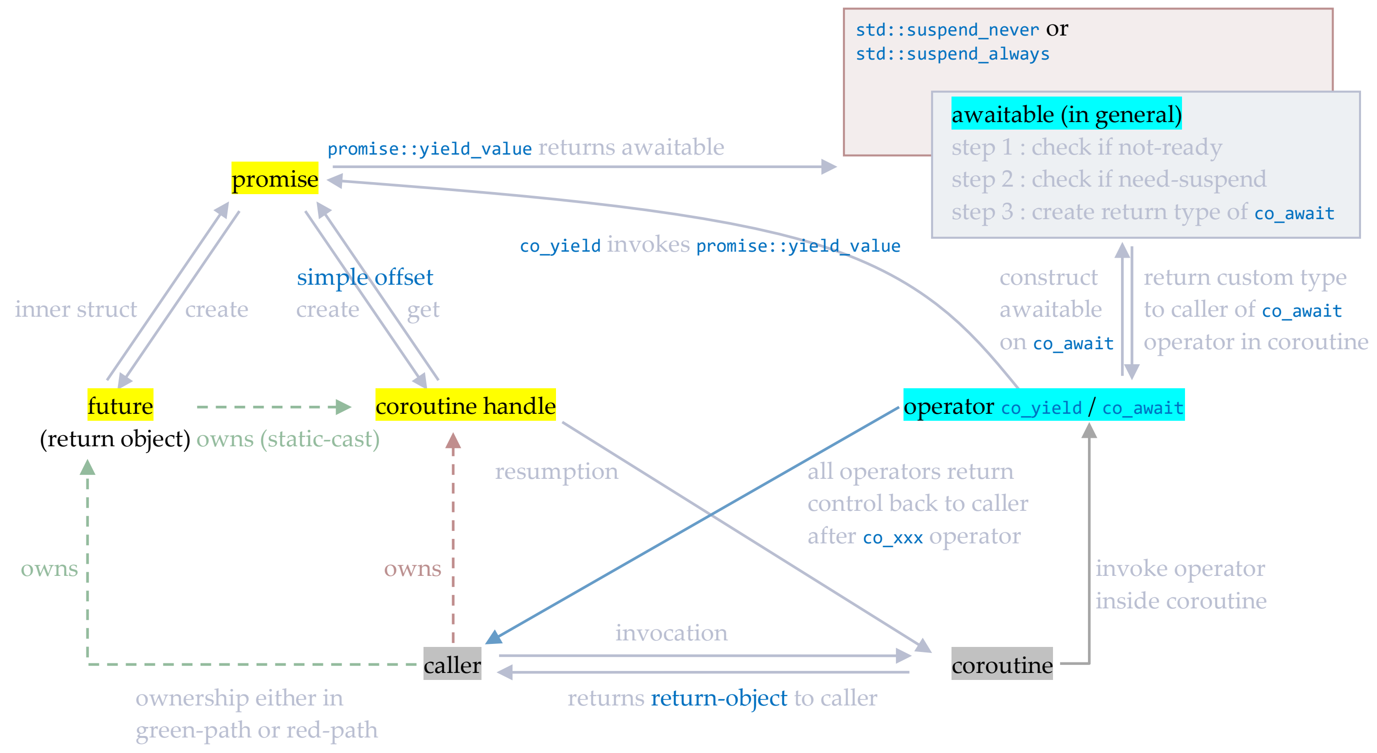
The 3 yielding operators in C++20 coroutine (all perform cooperative scheduling) :

- `co_await` an awaitable object, returns a data object (from caller to coroutine)
- `co_yield` a data object (from coroutine to caller)
- `co_return` to terminate coroutine

Please note that future is not `std::future`, while promise is not `std::promise`. Future / promise / coroutine handle are closely related :

- future defines promise (but not owns)
- promise creates future (by factory)
- promise is privately owned by coroutine handle (with a hidden offset in address), thus ...
  - promise can be obtained from handle by `promise_type& std::coroutine_handle<promise_type>::promise()`
  - handle can be constructed from promise by `auto std::coroutine_handle<promise_type>::from_promise(const promise_type&)`
  - both functions above are member functions of `std::coroutine_handle<promise_type>`
- besides, `std::coroutine_handle<promise_type>` can be implicitly converted to `std::coroutine_handle<void>`
  - the latter cannot be converted to promise anymore

DAG relating all components
Solid lines stands for a must-relationship, dotted lines stands for optional-relationship (just a common practice).



Now we are going to look into the following topics :
- 2 methods to get coroutine handle
- 2 methods to pass data between caller and coroutine
- compiler generated code by expanding awaitable : for customization of `co_await` operator
- compiler generated code by expanding promise : for customization of coroutine body : (see `expt.4`)
- `promise_type::initial_suspend()` is used to ensure exception safety on construction of handle
- `promise_type::final_suspend()` is used to protect the lifetime of handle on terminating coroutine

Since C++20 coroutine does not return coroutine handle, it returns future instead, we need to get a handle by 2 methods :
- pass a handle pointer as coroutine argument, which is initialized by the coroutine with the help of awaitable (see `expt.0`)
- keep a handle pointer as a future public member, which is initialized by the promise with the help of future factory (see `expt.1`)

Since C++20 coroutine invocation does not pass data between caller and coroutine, we need to do it via promise by 2 methods :
- keep data as promise public member, then either : (see `expt.2`)
- access it in caller / coroutine by `h.promise().data` where `h` is a handle, or
- access it in caller / coroutine by `p.data` where `p` is a promise
- keep data as promise public member, then : (see `expt.3`)
- update it in coroutine by `co_yield x` where `x` is a data
- method 1 works for passing data from coroutine to caller and from caller to coroutine
- method 2 works for passing data from coroutine to caller only

Experiment 0 - Method 1 to get coroutine handle

Before defining coroutine, we need to define future, as it is the return value of coroutine. Future should define an inner class called `promise_type` (don't mistakenly type as ~~`promise`~~) which has a few required member functions :

`get_return_object()`	creator of future (i.e. factory)
`unhandled_exception()`	handle exception after first resumption
`initial_suspend()`	determine whether to suspend on entry to coroutine body, return `std::suspend_never` for now
`final_suspend()`	determine whether to suspend on exit from coroutine body, return `std::suspend_never` for now

As the future below is default-constructible, we can simply `return{}` inside `get_return_object()`. After that, we define `awaitable` which has a few required member functions :

`await_ready()`	which is called right after coroutine is suspended when one of the yield-operators is invoked
`await_suspend()`	which is called only if `await_ready()` returns false (skipped otherwise), see the logic below
`await_resume()`	which is called right after coroutine is resumed when coroutine handle is invoked

This is how the types are binded :

- coroutine return type defines the future
- `typename future::promise_type` defines the promise
- operand type of `co_await` operator defines the awaitable

This is how the flow of execution happens :

- caller and coroutine cooperate, both yield to each other along time line, for the sake of mutual benefit
- awaitable is involved in every switch in flow of execution between caller and coroutine, thus it is a mediator
- future and promise is involved only when coroutine is called for the first time (not involved in yielding & resumption)

How do we get a handle in `expt.0`? All the blue lines are the answer.

```
struct future
{
    struct promise_type
    {
        promise_type()                     {           } // step 1
        future get_return_object()         { return{}; } // step 2
        void unhandled_exception()         {           }
        std::suspend_never initial_suspend() { return{}; } // step 4
        std::suspend_never   final_suspend() { return{}; }
    };

    future() {} // step 3
};

struct awaitable
{
    explicit awaitable(std::coroutine_handle<>* h_ptr_) : h_ptr(h_ptr_) {} // step 5

    bool await_ready()  const noexcept            { return false; } // step 6
    void await_suspend(std::coroutine_handle<> h) { *h_ptr = h;   } // step 7
    void await_resume() const noexcept            {               } // step 9

    std::coroutine_handle<>* h_ptr;
};

future coroutine(std::coroutine_handle<>* h_ptr)
{                                        // <--- invoke step 1-4
    awaitable a{ h_ptr };                // <--- invoke step 5
    for(std::uint32_t n=0;; ++n)
    {
        co_await a;                      // <--- invoke step 6-7
        std::cout << "I am coroutine."; // step 10
    }
}

// *** Expt 0 *** //
std::coroutine_handle<> h;
coroutine(&h);
for(int n=0; n<1000; ++n)
{
    std::cout << "I am caller."; // step 8
    h();                         // <--- invoke step 9
}
```

```
Time line

In coroutine entrance
step 1 : promise constructor
step 2 : create future by promise
step 3 : future constructor
step 4 : check if init suspension needed

In coroutine body
step 5 : awaitable constructor ⇑ once
---------------------------------------
                               ⇓ looping
In awaitable
step 6 : check if ready to move forward
         if true, do not yield
         if false, prepare to yield
-> 6.1 : create handle, time consuming
-> 6.2 : pass it to await_suspend()
step 7 : allow user to cache the handle
-> 7.1 : yielding to caller

In caller
step 8 : caller body

In awaitable
step 9 : resumption of coroutine
         starts with await_resume()

In coroutine body
step 10 : continue until next co_await

Keep looping the following
step 6,7,8,9,10
step 6,7,8,9,10
step 6,7,8,9,10
...

step 6,7,8,9,10
        ^--- step 8 is caller
            ^--- step 10 is coroutine

step 6,7,9 are awaitable, i.e. mediator
```

Explanation
What happen on yielding by `co_await` operator?

- `co_await` operator is a conditional yield, the condition is customizable inside awaitable
- `co_await` operator implements the logic, which skips time consuming backup of coroutine states, if yielding is not needed

```
if (!a.await_ready()) // please customize the condition inside await_ready() function
{
    std::coroutine_handle<promise_type> h = __backup_coroutine_states_in_activation_frame();
    a.await_suspend(h);
    yield_to_caller();
}
else continue_with_coroutine();
```

What happen on invoking coroutine handle?

- `a.await_suspend()` is called first, before running the code below `co_await a;`
- `a.await_suspend()` is customizable and returns data
- which can be received if we rewritten the line as `auto data = co_await a;`

Experiment 1 - Method 2 to get coroutine handle
The second method to get coroutine handle to caller :

- the coroutine handle is owned by future as its public member
- the coroutine handle is initialized on future construction, via the factory inside promise
- now the future is a POD with one member, hence it can be brace-constructed like the blue line
- no custom awaitable object is needed (as no `expt.0 step.7` is needed), `std::suspend_always` offered by STL is good enough

```
struct future
{
    struct promise_type
    {
        promise_type()                       {             }
        future get_return_object()           { return future { std::coroutine_handle<promise_type>::from_promise(*this) }; }
        void unhandled_exception()           {             }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never   final_suspend() { return {}; }
    };

    std::coroutine_handle<promise_type> h;
};

[[nodiscard]] future coroutine()
{
    for(std::uint32_t n=0;; ++n)
    {
        co_await std::suspend_always{};
        std::cout << "I am coroutine.";
    }
}

// *** Expt 1 *** //
future f = coroutine();
for(int n=0; n<1000; ++n)
{
    std::cout << "I am caller.";
    f.h();
}
```

Experiment 2 - Method 1 to transmit data

This data transmission works in both way : from caller to coroutine and from coroutine to caller. New codes are *blue highlighed*.

- housekeep data in promise, we have `T` for caller to coroutine data and `U` for coroutine to caller data (we can merge them)
- housekeep data pointer in awaitable, which will point to corresponding data (in next step)
- like what we did in `expt.0`, initialize data pointer in awaitable, pointing to data in promise
- caller can access data through `future.promise.data`
- coroutine access data through `promise.resume` which returns data to `co_await` operator

```
// T = data from caller to coroutine
// U = data from coroutine to caller
template<typename T, typename U>
struct future
{
    struct promise_type
    {
        promise_type()                        {            }
        future<T,U> get_return_object()       { return future<T,U>{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        void unhandled_exception()            {            }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never   final_suspend() { return {}; }

        T data_T;
        U data_U;
    };

    explicit future(std::coroutine_handle<promise_type> h_) : h(h_) {}
    operator std::coroutine_handle<promise_type>() const { return h; }  // *** Conversion operator *** //

    std::coroutine_handle<promise_type> h;
};

template<typename T, typename U>
struct awaitable
{
    awaitable() : data_T_ptr(nullptr), data_U_ptr(nullptr) {}

    bool await_ready() const noexcept { return false; }
    bool await_suspend(std::coroutine_handle<typename future<T,U>::promise_type> h)
    {
        data_T_ptr = &(h.promise().data_T);
        data_U_ptr = &(h.promise().data_U);
        return true;
    }
    std::pair<T*,U*> await_resume() const noexcept
    {
        return std::make_pair(data_T_ptr, data_U_ptr);
    }

    // Keep data-pointer instead of handle-pointer
    T* data_T_ptr;
    U* data_U_ptr;
};

template<typename T, typename U>
[[nodiscard]] future<T,U> coroutine()
{
    for(std::uint32_t n=0;; ++n)
    {
        // Read t from caller to coroutine
        // Send u from coroutine to caller
        auto [T_ptr,U_ptr] = co_await awaitable<T,U>{}; // wait for caller until t is ready
        U_ptr->a = T_ptr->a;
        U_ptr->b = T_ptr->a + T_ptr->n;
        U_ptr->c = T_ptr->a + T_ptr->n * 2;
        co_await std::suspend_always{}; // u is ready and wait for caller
    }
}

// Sample POD
struct pod_T { char a; std::uint16_t n; };
struct pod_U { char a; char b; char c;  };

inline std::ostream& operator<<(std::ostream& os, const pod_T& t) { os << t.a << "_" << t.n; return os; }
inline std::ostream& operator<<(std::ostream& os, const pod_U& u) { os << u.a << u.b << u.c; return os; }

// *** Expt 2 *** //
std::coroutine_handle<future<pod_T,pod_U>::promise_type> h = coroutine<pod_T,pod_U>();
auto& p = h.promise();
for(int n=0; n<8; ++n)
{
    p.data_T.a = static_cast<char>('A' + n);
    p.data_T.n = n;
    std::cout << "\ncaller inputs  " << p.data_T;     h();
    std::cout << "\ncaller outputs " << p.data_U;     h();
}
```

Experiment 3 - Method 2 to transmit data

However, if we aim at transmission of data from coroutine to caller only, the transmission can be made simpler using `co_yield`. This method does not involve customization of awaitable.

- housekeep data in promise, same as `expt.2`
- transmit data from coroutine to caller through `co_yield` operator
- consequence of `co_yield` operator is defined in `promise_type::yield_value`

```
template<typename T>
struct future
{
    struct promise_type
    {
        promise_type()                      {           }
        future get_return_object()          { return future{ std::coroutine_handle<promise_type>::from_promise(*this)}; }
        void unhandled_exception()          {           }
        std::suspend_never initial_suspend() { return{}; }
        std::suspend_never   final_suspend() { return{}; }
        std::suspend_always  yield_value(const T& x)
        {
            data = x; // deep copy
            return {};
        }

        T data;
    };

    explicit future(std::coroutine_handle<promise_type> h_) : h(h_) {}
    operator std::coroutine_handle<promise_type>() const { return h; } // *** Conversion operator *** //

    std::coroutine_handle<promise_type> h;
};

template<typename T>
[[nodiscard]] future<T> coroutine()
{
    for(std::uint16_t n=0;; ++n)
    {
        T t{ static_cast<char>('A'+n), n };
        co_yield t;
        std::cout << "\ncoroutine::iteration " << n << ", " << t;
    }
}.

std::coroutine_handle<future<pod_T>::promise_type> h = coroutine<pod_T>();
auto& p = h.promise();
for(int n=0; n<8; ++n)
{
    std::cout << "\ncaller outputs " << p.data;
    h();
}
```

These two functions are used to customize whether to have extra suspension on entering or exiting coroutine. The former is for exception safety while the latter is for postponing lifetime of coroutine handle on exiting the coroutine. The latter is used in my `jthreadpool` *example in next section.*

Summary

In short, this is how we inter-change between coroutine handle and promise :

```
// From promise to handle
auto handle = std::coroutine_handle<future::promise_type>::from_promise(promise);

// From handle to promise
std::coroutine_handle<future::promise_type> handle = ...; // by some means
auto promise = handle.promise();
```

In short, promise defines the return of coroutine, while awaitable defines the return of `co_await` operator :

```
// Return value of coroutine
struct future
{
    struct promise_type
    {
        future get_return_object();
    };
};

// Return value of co_await operator
struct awaitable
{
    R await_resume() const noexcept;
};
```

E. Cooperative cancellation with `std::stop_source` and `std::jthread`

We have two cooperative concepts :

- `std::this_thread::yield()` implements cooperative scheduling, which gives up CPU resources willingly to other threads
- `std::coroutine_handle` implements cooperative scheduling, which gives up CPU resources willingly to caller
- `std::stop_source` implements cooperative cancellation, which stopping current thread willingly

- In this section, we will implement a `jthreadpool` for `std::coroutine_handle` with cooperative cancellation.

Background of stop source

The story starts with a program crash in threadpool. With `gdb` backtrace, we find that a crash happens when threadpool constructor calls its destructor, implying that it is an exception thrown in constructor and triggers stack unwinding. It is a desirable effect that if any problem found during construction, `std::runtime_error` exception should be thrown with human readable messages. However if extra problems happen inside destructor, the program has to be `std::terminate()`. For threadpool, when destructor tries to destroy a vector of `std::thread` when they are still running, `std::terminate()` is called.

Thus we need a thread to do two things before it is destroyed :

- cooperative stopping of the task (we hope to achieve it by `std::stop_source`)
- automatic joining of the thread (we hope to achieve it by `std::jthread`)

About stop source and stop token

For thread running a finite loop, there is no stop-issue. However for thread running an infinite loop, we need to take care of stop. It should be done with a well managed stop button. `std::stop_source` and `std::stop_token` are publication pattern of an atomic stop flag, like the `std::promise` and `std::future` pair. One stop source can connect to multiple stop tokens, while each thread should listen to its own token for cooperative cancellation. It is purely cooperative, which means it is ***NOT*** forced to do so.

```
std::stop_source s_source;
std::vector<std::thread> threads;
for(std::uint32_t n=0; n!=5; ++n)
{
    threads.push_back(std::thread([s_token = s_source.get_token()](std::uint32_t thread_id)
    {
        std::uint32_t m = 0;
        while(!s_token.stop_requested())
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(400 + 40 * thread_id));
            ++m;
        }
    }, n));
}
std::this_thread::sleep_for(std::chrono::seconds(4));

// *** Stop and join should be done explicitly *** //
s_source.request_stop();
for(auto& x:threads)
{
    if (x.joinable()) x.join(); // It is joinable if it is not joined before. To avoid double join, which crashes.
}
```

There is an internal stop source in each `std::jthread`, which is `request_stop()` automatically on destruction. Like that :

```
~jthread()
{
    if (joinable()) { internal_stop_source.request_stop(); join(); }
}
```

With `std::jthread`, we can simplify above test. Instead of single `std::stop_source`, we now have one stop for each thread :

```
std::vector<std::jthread> threads;
for(std::uint32_t n=0; n!=5; ++n)                      ⇩ std::stop_token must be the 1st argument, which ...
{                                                      ⇩ is binded to internal stop source automatically.
    threads.push_back(std::jthread([](std::stop_token s_token, std::uint32_t thread_id)
    {
        std::uint32_t m = 0;
        while(!s_token.stop_requested())
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(400 + 40 * thread_id));
            ++m;
        }
    }, n)); // <--- No need to bind other things.
}
std::this_thread::sleep_for(std::chrono::seconds(4));
// On destruction, request_stop() and join() are called automatically
```

About stop callback and condition variable
However not all threads can continuously listening to the stop token, such as threads waiting for a condition variable in threadpool. To solve this problem, wrapper `std::condition_variable_any` is introduced, its wait function takes the stop token as an extra argument, so that the condition variable will be notified if stop is requested for that stop token, and the waiting thread will be woken. Besides, this wait function returns a boolean :

- `true` denoting that the thread is woken because the waiting condition is fulfilled (it should remain in `1st loop`) or
- `false` denoting that the thread is woken because the stop source got a stop request (it should move to `2nd loop`)

Our threadpool can thus be rewritten in an elegant way :

```
class jthreadpool_cv
{
public:
    explicit jthreadpool_cv(std::uint32_t num_threads)
    {
        for(std::uint32_t n=0; n!=num_threads; ++n)
        {
            jthreads.emplace_back(std::jthread(&jthreadpool_cv::fct, this, s_source.get_token(), n));
        }
    }

    // Explicit thread-join no longer needed.
    ~jthreadpool_cv() { stop(); }

    // Emit stop signal to all threads (they are listening willingly)
    void stop() { s_source.request_stop(); }

public:
    void add_task(const std::function<void()>& task) // This function is unchanged.
    {
        {
            std::lock_guard<std::mutex> lock(mutex);
            tasks.push(task);
        }
        condvar.notify_one();
    }

private:
    void fct(std::stop_token s_token, std::uint32_t id)
    {
        try
        {
            // *** 1st loop *** //
            while(!s_source.get_token().stop_requested())
            {
                std::function<void()> task;
                {
                    std::unique_lock<std::mutex> lock(mutex);
                    if (!condvar.wait(lock, s_token, [this]()
                    {
                        return !tasks.empty();
                    //         || out_of_scope.load(); // No long needed. Predicate is decoupled from stop-request.
                    }))
                    {
                        break;
                    }

                //  if (out_of_scope.load()) break; // No long needed. It is handled by the return of condvar.wait() already.
                    task = std::move(tasks.front());
                    tasks.pop();
                }
                task();
            }

            // *** 2nd loop *** //
            while(!tasks.empty())
            {
                std::function<void()> task;
                {
                    std::lock_guard<std::mutex> lock(mutex);
                    task = std::move(tasks.front());
                    tasks.pop();
                }
                task();
            }
        }
        catch(std::exception& e) { std::cout << "exception caught in " << id << e.what(); }
    }

private:
    std::vector<std::jthread> jthreads;
    std::queue<std::function<void()>> tasks;
    mutable std::mutex mutex;
    std::condition_variable_any condvar;   // Replace condvar by condvar_any.
    std::stop_source s_source;             // Replace out_of_scope by stop-source.
};
```

Further study : What happen if we have multi stop source for the same condition variable? Given :

- 8 threads waiting on the same condition variable
- however 4 threads invoke the wait with `token0`, like `if (!cv.wait(lock, token0, [this](){ return !tasks.empty(); })) break;`
- whereas 4 threads invoke the wait with `token1`, like `if (!cv.wait(lock, token1, [this](){ return !tasks.empty(); })) break;`
- where `token0` and `token1` are generated from two stop sources `src0` and `srcn1` respectively
- which threads will be woken if I call `src0.request_stop()`?

Generalizing threadpool to coroutine

We can generalize our threadpool to coroutine by the following changes :

- replace `std::function<void()>` task by `std::coroutine_handle<>`
- if a task is not completed (`!task.done()`), insert it back to task queue
- while constructor, destructor, `stop()`, `add_task()` are kept unchanged

```
void fct(std::stop_token s_token, std::uint32_t id)
{
    try
    {
        // *** 1st loop *** //
        while(!s_source.get_token().stop_requested())
        {
            std::coroutine_handle<> task;
            {
                std::unique_lock<std::mutex> lock(mutex);
                if (!condvar.wait(lock, s_token, [this]() { return !tasks.empty(); }))
                {
                    break;
                }

                task = std::move(tasks.front());
                tasks.pop();
            }
            task();

            // *** Unfinished coroutine *** //
            if (!task.done()) add_task(task);
        }

        // *** 2nd loop *** //
        while(!tasks.empty())
        {
            std::coroutine_handle<> task;
            {
                std::lock_guard<std::mutex> lock(mutex);
                task = std::move(tasks.front());
                tasks.pop();
            }
            task();

            // *** Unfinished coroutine *** //
            if (!task.done()) add_task(task);
        }
    }
    catch(std::exception& e) { std::cout << "exception caught " << id << e.what(); }
}

private: std::queue<std::coroutine_handle<>> tasks;
```

There is a contraint on the coroutine used in this threadpool. As we need to check if the coroutine is completed by `!task.done()`, this function will crash if the coroutine handle no longer live if it is done. Thus we need to postpone its lifetime by ensuring :

```
struct sample_future
{
    struct promise_type
    {
        sample_future get_return_object()
        {
             return sample_future { std::coroutine_handle<promise_type>::from_promise(*this) };
        }
        void unhandled_exception(){}
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always  final_suspend() { return {}; }
    };

    std::coroutine_handle<promise_type> h;
};
```

Please read : `std::jthread` *and cooperative cancellation, nextptr.com*