Serial and parallel pipelines in modern C++

Bogusław Cyganek

cyganek@agh.edu.pl

CppNorth, Toronto, Canada, July 22nd 2025

A pipeline architecture in C++ - AN OVERVIEW

- Introduction to pipes and pipeline in software engineering.
- Going functional in C++ (pipeline, pure functions, function currying).
- How to overload the pipe `operator |`.
- How to build and use our own pipeline.
- New kid on the block – `std::expected`.
- Parallel versions of the pipeline framework.

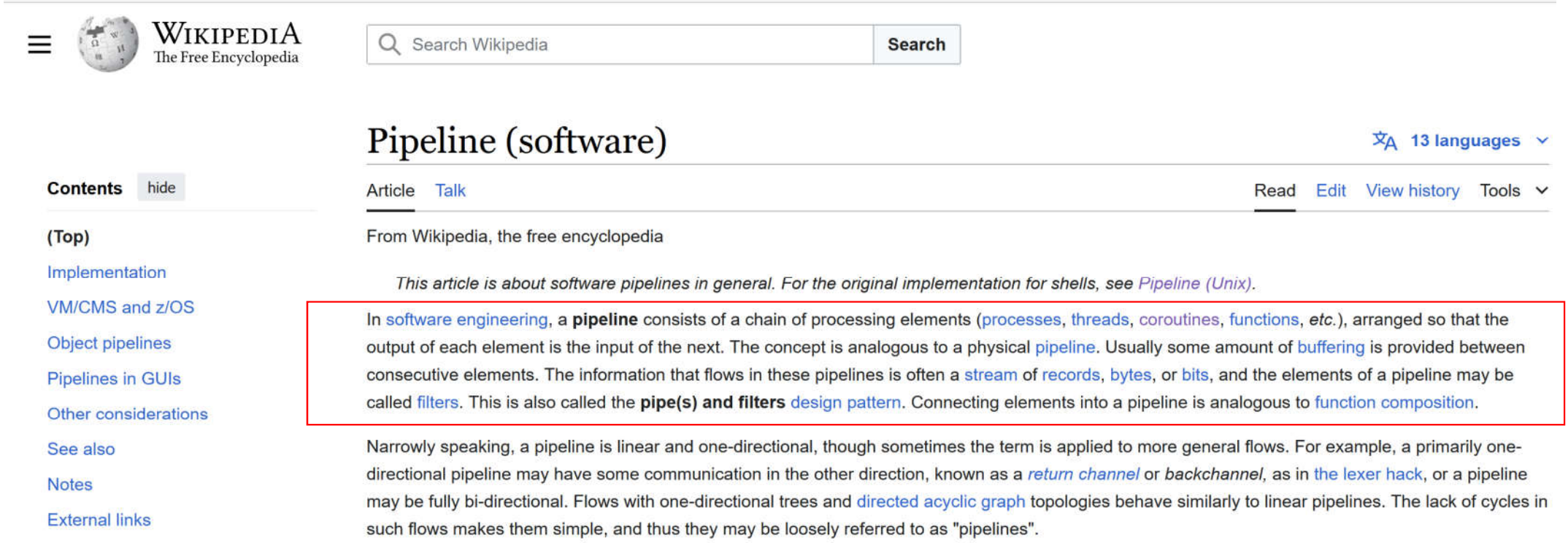
A pipeline architecture in C++

What is a pipe and a pipeline?

https://pixabay.com

A pipeline architecture in C++

What is a pipe and a pipeline?



WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Pipeline (software)

13 languages

Contents hide

(Top)
Implementation
VM/CMS and z/OS
Object pipelines
Pipelines in GUIs
Other considerations
See also
Notes
External links

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article is about software pipelines in general. For the original implementation for shells, see Pipeline (Unix).

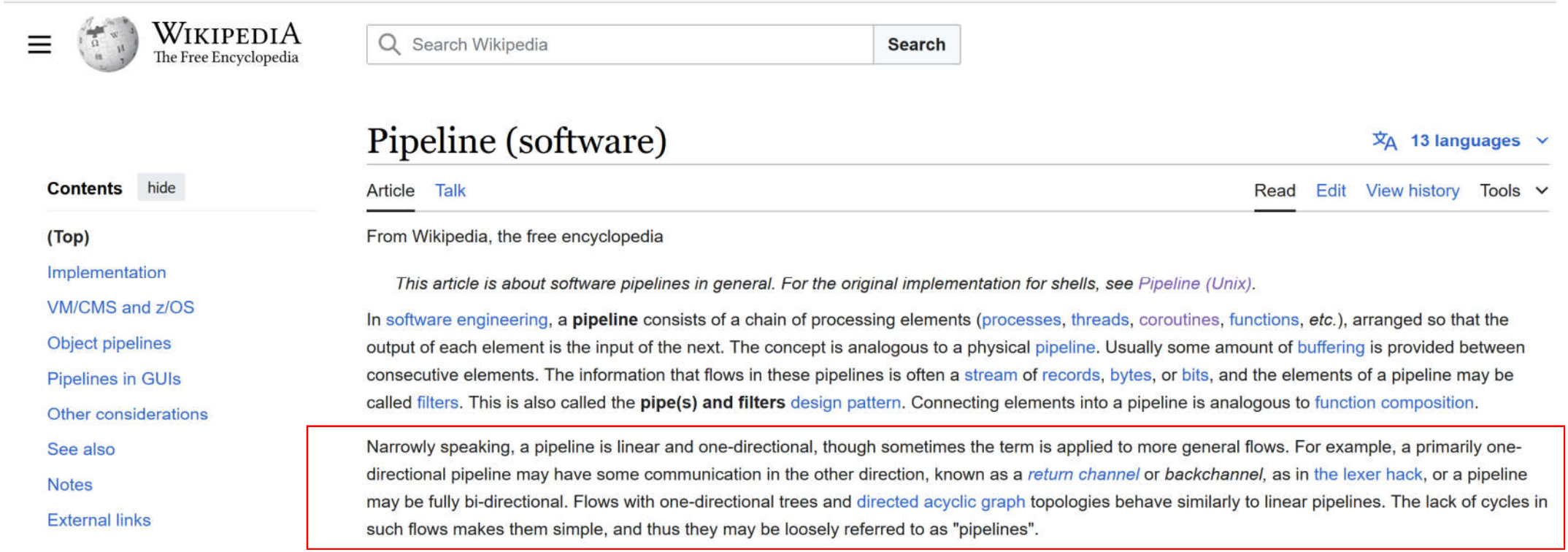
In software engineering, a **pipeline** consists of a chain of processing elements (processes, threads, coroutines, functions, *etc.*), arranged so that the output of each element is the input of the next. The concept is analogous to a physical pipeline. Usually some amount of buffering is provided between consecutive elements. The information that flows in these pipelines is often a stream of records, bytes, or bits, and the elements of a pipeline may be called filters. This is also called the **pipe(s) and filters** design pattern. Connecting elements into a pipeline is analogous to function composition.

Narrowly speaking, a pipeline is linear and one-directional, though sometimes the term is applied to more general flows. For example, a primarily one-directional pipeline may have some communication in the other direction, known as a *return channel* or *backchannel*, as in the lexer hack, or a pipeline may be fully bi-directional. Flows with one-directional trees and directed acyclic graph topologies behave similarly to linear pipelines. The lack of cycles in such flows makes them simple, and thus they may be loosely referred to as "pipelines".

https://en.wikipedia.org/wiki/Pipeline_(software)

A pipeline architecture in C++

What is a pipe and a pipeline?



WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Pipeline (software)

13 languages

Contents hide

(Top)
Implementation
VM/CMS and z/OS
Object pipelines
Pipelines in GUIs
Other considerations
See also
Notes
External links

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article is about software pipelines in general. For the original implementation for shells, see Pipeline (Unix).

In software engineering, a **pipeline** consists of a chain of processing elements (processes, threads, coroutines, functions, *etc.*), arranged so that the output of each element is the input of the next. The concept is analogous to a physical pipeline. Usually some amount of buffering is provided between consecutive elements. The information that flows in these pipelines is often a stream of records, bytes, or bits, and the elements of a pipeline may be called filters. This is also called the **pipe(s) and filters** design pattern. Connecting elements into a pipeline is analogous to function composition.

Narrowly speaking, a pipeline is linear and one-directional, though sometimes the term is applied to more general flows. For example, a primarily one-directional pipeline may have some communication in the other direction, known as a *return channel* or *backchannel*, as in the lexer hack, or a pipeline may be fully bi-directional. Flows with one-directional trees and directed acyclic graph topologies behave similarly to linear pipelines. The lack of cycles in such flows makes them simple, and thus they may be loosely referred to as "pipelines".

https://en.wikipedia.org/wiki/Pipeline_(software)

Pipes in software engineering

A pipeline architecture in C++

A pipeline concept is well known from UNIX (and Linux) – invented by Douglas McIlroy '73

A pipeline is a way for inter-process communication with message passing.

A pipeline means a number of processes chained together by their streams – the output of the predecessor process is passed as input to its successor.

Commands are separated by the vertical bar – the pipe operator |

```
ls -l | grep 'myFile' > log.txt
```

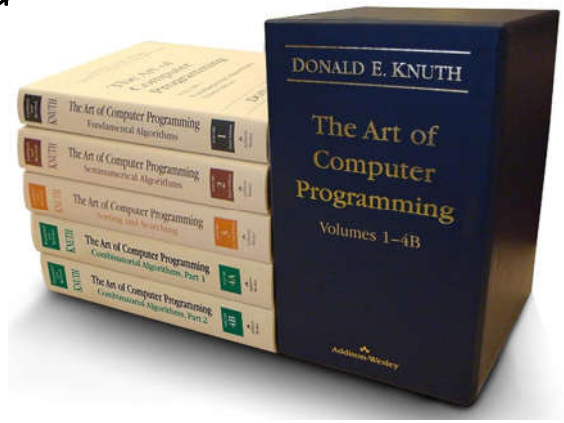
https://en.wikipedia.org/wiki/Pipeline_(Unix)

A pipeline architecture in C++

An interesting story happend in 1986 ...

The famous computer science professor Donald Knuth was asked by John Bentley (the editor of Programming Pearls in Communications of the ACM) to implement a procedure for finding *the most frequently occurring words in a text*.

Prof. Knuth took up the challenge and after a few weeks presented his solution.



His implementation was detailed but rather low-level, using advanced techniques such as *hash trie*, and the publication of the code took many pages (Knuth also wanted to show his idea of *Literate Programming* approach).

A pipeline architecture in C++

An interesting story happend in 1986 ...

A clever response came from Doug McIlroy, who provided a Unix shell script of just 6 commands for the word count task.

```
tr –cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq –c |
sort –rn |
sed ${1}q
```

A pipeline architecture in C++

Just a warm-up on a function composition

A(R) is equivalent to R | A

```
std::vector vals { 10, 11, 2, -3, 4, 55, 0, 0, -4, 10, -8 };


for( auto p :  std::views::transform(
                      std::views::filter( vals,    []( auto a ){ return a > 0; } ),
                                                   []( auto a ){ return 2 * a; } ) )
       std::cout << p << ", ";
```

```
20, 22, 4, 8, 110, 20,
```

The nice thing in the std::ranges library is the pipe operator | for pipelining the operations

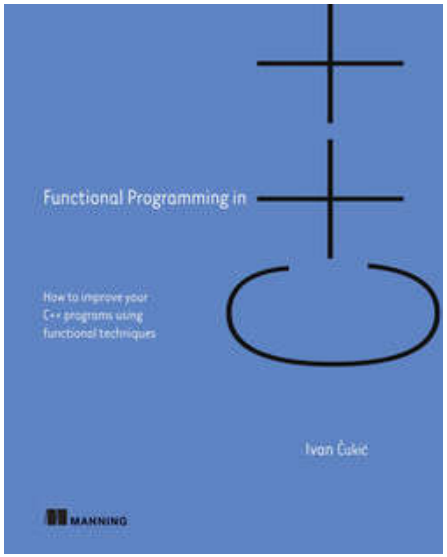
```
for( auto p : vals    | std::views::filter( []( auto a ){ return a > 0; } )
                      | std::views::transform( []( auto a ){ return 2 * a; } ) )
       std::cout << p << ", ";
```

A pipeline architecture in C++

I started to look up how the pipe operator is implemented but also how it can be used in a custom C++ code organized in the pipeline fashion.

https://www.youtube.com/watch?v=L_bomNazb8M

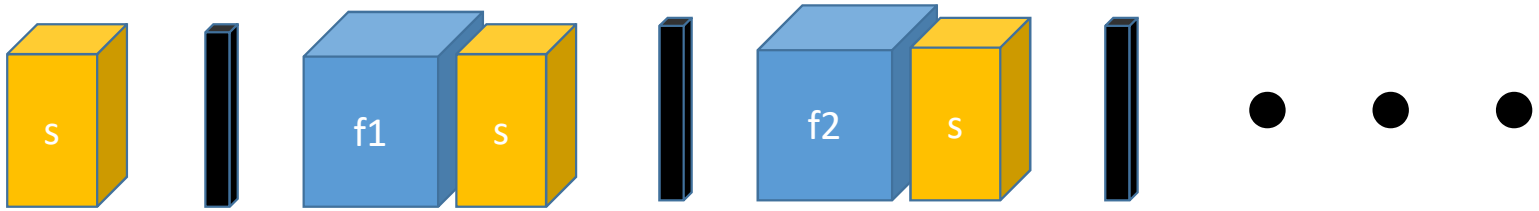
Functional Programming in C++
by Ivan Cukic



A pipeline architecture in C++

```
using Function = std::function< std::string( std::string && ) >;
```

```
auto operator | ( std::string && s, Function f ) -> std::string
{
        return f( std::move( s ) );
}
```



- The | operator simply calls f providing it with s.
- s needs to be std::move'd, since it is a named object here.
- The callable f must be able to accept std::string && as its parameter and return std::string

A pipeline architecture in C++

- The callable `f` must be able to accept `std::string &&` as its parameter and return `std::string`

```
std::string StringProc_1( std::string && s )
{
        s += " proc by 1,";
        std::cout << "I'm in StringProc_1, s = " << s << "\n";
        return s;
}

std::string StringProc_2( std::string && s )
{
        s += " proc by 2,";
        std::cout << "I'm in StringProc_2, s = " << s << "\n";
        return s;
}

std::string StringProc_3( std::string && s )
{
        s += " proc by 3,";
        std::cout << "I'm in StringProc_3, s = " << s << "\n";
        return s;
}
```

A pipeline architecture in C++

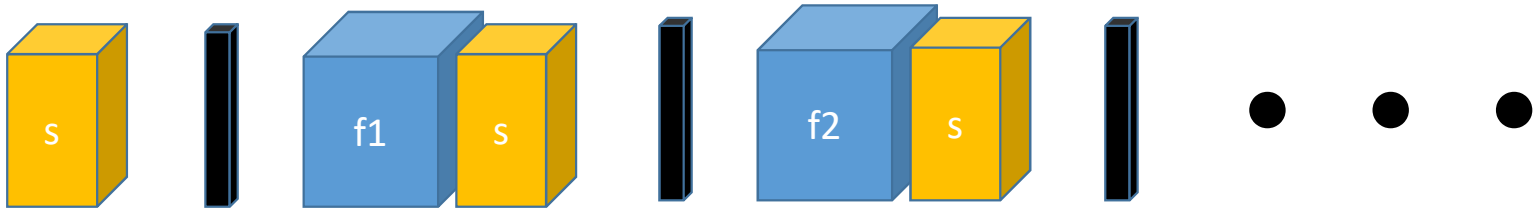
```
void SimplePipeTest()
{
      std::string start_str( "Start string " );
      std::cout << ( std::move( start_str ) | StringProc_1 | StringProc_2 | StringProc_3 );
}
```

```
I'm in StringProc_1, s = Start string  proc by 1,
I'm in StringProc_2, s = Start string  proc by 1, proc by 2,
I'm in StringProc_3, s = Start string  proc by 1, proc by 2, proc by 3,


Start string  proc by 1, proc by 2, proc by 3,
```

A pipeline architecture in C++

```
using Function = std::function< std::string( std::string && ) >;
```

```
auto operator | ( std::string && s, Function f ) -> std::string
{
        return f( std::move( s ) );
}
```

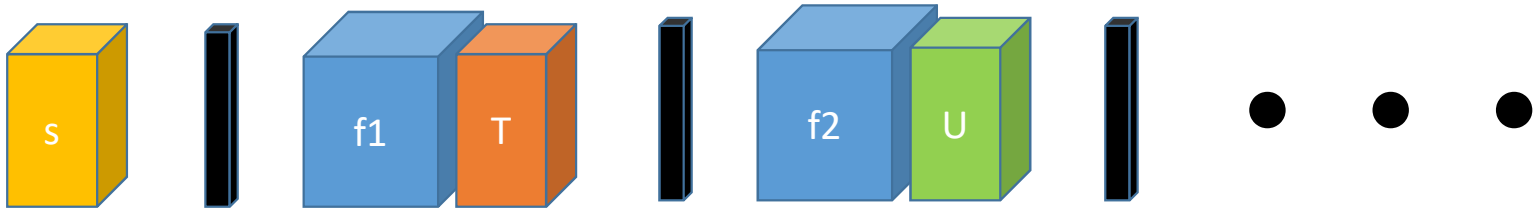


- The | operator simply calls f providing it with s.
- s needs to be std::move'd, since it is a named object here.
- The callable f must be able to accept std::string && as its parameter and return std::string

There is more on pipelines…

```
using Function = std::function< std::string( std::string && ) >;
```

```
auto operator | ( std::string && s, Function f ) -> std::string
{
        return f( std::move( s ) );
}
```



But what if we wish to have DIFFERENT INTERFACES on each function?

A pipeline architecture in C++

The improved version has to distinguish itself from the ranges pipe operator.

```
template < typename T, typename Function >
        requires ( not std::ranges::range< T > && std::invocable< Function, T > )
        constexpr auto operator |     ( T && t, Function && f )
                                      -> typename std::invoke_result_t< Function, T >
{
        return std::invoke( std::forward< Function >( f )( std::forward< T >( t ) ) );
}
```

A pipeline architecture in C++

The improved version has to distinguish itself from the ranges pipe operator.

```
template < typename T, typename Function >
        requires ( not std::ranges::range< T > && std::invocable< Function, T > )
        constexpr auto operator |     ( T && t, Function && f )
                                      -> typename std::invoke_result_t< Function, T >
{
        return std::invoke( std::forward< Function >( f )( std::forward< T >( t ) ) );
}
```

- A concept has been added that requires:
  - That the provided type `T` is not a range, that is not of type `std::ranges::range<T>`. This is to avoid clashes with `operator |` defined in `std::ranges`.
  - That `Function` parameter can be invoked with a parameter of type `T`.

A pipeline architecture in C++

The improved version has to distinguish itself from the ranges pipe operator.

```
template < typename T, typename Function >
        requires ( not std::ranges::range< T > && std::invocable< Function, T > )
        constexpr auto operator |     ( T && t, Function && f )
                                      -> typename std::invoke_result_t< Function, T >
{
        return std::invoke( std::forward< Function >( f )( std::forward< T >( t ) ) );
}
```

- `constexpr`, so the `operator |` can be called and executed at compile time, if its arguments are also available at that time.

A pipeline architecture in C++

The improved version has to distinguish itself from the ranges pipe operator.

```
template < typename T, typename Function >
        requires ( not std::ranges::range< T > && std::invocable< Function, T > )
        constexpr auto operator |     ( T && t, Function && f )
                                      -> typename std::invoke_result_t< Function, T >
{
        return std::invoke( std::forward< Function >( f )( std::forward< T >( t ) ) );
}
```

- The return type is defined with the helper, `std::invoke_result_t< Function, T >` which deduces the return type at compile time.

A pipeline architecture in C++

The improved version has to distinguish itself from the ranges pipe operator.

```
template < typename T, typename Function >
        requires ( not std::ranges::range< T > && std::invocable< Function, T > )
        constexpr auto operator |     ( T && t, Function && f )
                                      -> typename std::invoke_result_t< Function, T >
{
        return std::invoke( std::forward< Function >( f )( std::forward< T >( t ) ) );
}
```

- Calls `std::invoke` that invokes the callable object `f` with the parameter `t`. The benefit of using , instead of a direct call `f( t )`, is that the former works with any callable, such as a function pointer, a reference to a function, a lambda function, a member function pointer, a functional object (i.e. the one with operator() on board), or a pointer to member data. In other words, the callable `f` has to satisfy the Callable concept (*https://en.cppreference.com/w/cpp/concepts*).

A pipeline architecture in C++

A PROBLEM… what to do if one link in the above pipeline cannot complete its operation and transmit its result because an error occurred?

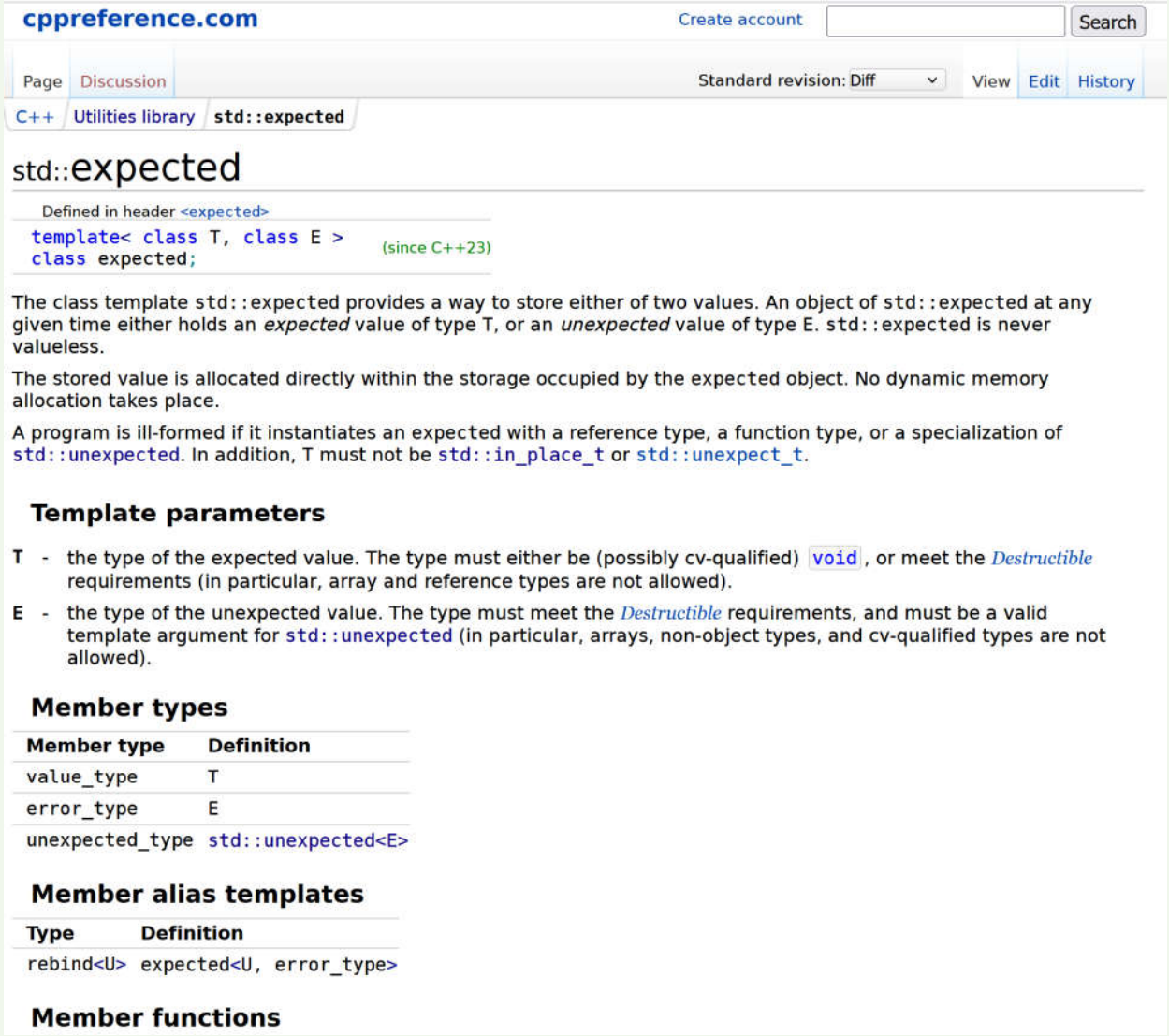
We can `throw` an exception and interrupt the entire operation.

We can use `std::optional` to express whether the operation was successful and we have the result, or whether we have a situation in which the calculations failed for some reason and we simply cannot provide any result.

But if you have a C++23 compiler, an even better option is to use **`std::expected`**. Unlike `std::optional`, which has been available since C++17, in the event of a calculation failure, it allows you to pass an error code and not just state the failure.

std::expected

An intro to std::expected



cppreference.com Create account Search

Page Discussion Standard revision: Diff View Edit History

C++ Utilities library std::expected

std::expected

Defined in header <expected>

```
template< class T, class E >
class expected;
```
(since C++23)

The class template std::expected provides a way to store either of two values. An object of std::expected at any given time either holds an *expected* value of type T, or an *unexpected* value of type E. std::expected is never valueless.

The stored value is allocated directly within the storage occupied by the expected object. No dynamic memory allocation takes place.

A program is ill-formed if it instantiates an expected with a reference type, a function type, or a specialization of std::unexpected. In addition, T must not be std::in_place_t or std::unexpect_t.

Template parameters

T - the type of the expected value. The type must either be (possibly cv-qualified) void, or meet the *Destructible* requirements (in particular, array and reference types are not allowed).

E - the type of the unexpected value. The type must meet the *Destructible* requirements, and must be a valid template argument for std::unexpected (in particular, arrays, non-object types, and cv-qualified types are not allowed).

Member types

Member type	Definition
value_type	T
error_type	E
unexpected_type	std::unexpected<E>

Member alias templates

Type	Definition
rebind<U>	expected<U, error_type>

Member functions

https://en.cppreference.com/w/cpp/utility/expected

An intro to std::expected

Used to store *exactly one value* at the time: *expected* or *unexpected*. Hence, it can be used in all situations where we have a 'normal' result or an error.

- It is a wrapper type that contains either of two values: an *expected* of type T, or an *unexpected* (error) of type E.
- Either value is stored directly in the `std::expected` object, i.e. there is no dynamic memory allocation.
- `std::unexpected` is a template class to represent an unexpected value E.
- T cannot be a reference type, a function type, or a specialization of `std::unexpected`.
- Before trying to access the expected value always check with `has_value()` or `bool()` if it is available, otherwise UB.
- Available monadic operations.
- Can be used instead of `std::optional`.

An intro to std::expected

Member	Description
`constructor`	Constructs the `std::expected<T,E>` object with zero or a number of parameters. Can use `std::in_place` to ensure in place construction. Can be created from a single `std::unexpected` object.

An intro to std::expected

Member	Description
`constructor`	Constructs the `std::expected<T,E>` object with zero or a number of parameters. Can use `std::in_place` to ensure in place construction. Can be created from a single `std::unexpected` object.
`operator bool` `has_value`	Returns true if `std::expected` contains a valid object, `false` otherwise.

An intro to std::expected

Member	Description
`constructor`	Constructs the `std::expected<T,E>` object with zero or a number of parameters. Can use `std::in_place` to ensure in place construction. Can be created from a `single std::unexpected` object.
`operator bool` `has_value`	Returns true if `std::expected` contains a valid object, false otherwise.
`operator ->` `operator *`	Access the expected value. UB if has_value returns false.

An intro to std::expected

Member	Description
`constructor`	Constructs the `std::expected<T,E>` object with zero or a number of parameters. Can use `std::in_place` to ensure in place construction. Can be created from a single `std::unexpected` object.
`operator bool` `has_value`	Returns true if `std::expected` contains a valid object, false otherwise.
`operator ->` `operator *`	Access the expected value. UB if has_value returns false.
`value`	Returns a reference to the expected value, if present. Returns nothing if T is void. Otherwise throws `std::bad_expected_access`

An intro to std::expected

Member	Description
`constructor`	Constructs the `std::expected<T,E>` object with zero or a number of parameters. Can use `std::in_place` to ensure in place construction. Can be created from a single `std::unexpected` object.
`operator bool` `has_value`	Returns true if `std::expected` contains a valid object, false otherwise.
`operator ->` `operator *`	Access the expected value. UB if has_value returns false.
`value`	Returns a reference to the expected value, if present. Returns nothing if T is void. Otherwise throws `std::bad_expected_access`
`error`	Returns a reference to the unexpected value, if present. UB if has_value returns true.

An intro to std::expected

Member	Description
`constructor`	Constructs the `std::expected<T,E>` object with zero or a number of parameters. Can use `std::in_place` to ensure in place construction. Can be created from a single `std::unexpected` object.
`operator bool` `has_value`	Returns true if std::expected contains a valid object, false otherwise.
`operator ->` `operator *`	Access the expected value. UB if has_value returns false.
`value`	Returns a reference to the expected value, if present. Returns nothing if T is void. Otherwise throws `std::bad_expected_access`
`error`	Returns a reference to the unexpected value, if present. UB if has_value returns true.
`value_or( d )`	Returns the contained value if present, otherwise returns d. Not declared if T is void.

An intro to std::expected

Member	Description
`constructor`	Constructs the `std::expected<T,E>` object with zero or a number of parameters. Can use `std::in_place` to ensure in place construction. Can be created from a single `std::unexpected` object.
`operator bool` `has_value`	Returns true if `std::expected` contains a valid object, false otherwise.
`operator ->` `operator *`	Access the expected value. UB if has_value returns false.
`value`	Returns a reference to the expected value, if present. Returns nothing if T is void. Otherwise throws `std::bad_expected_access`
`error`	Returns a reference to the unexpected value, if present. UB if has_value returns true.
`value_or( d )`	Returns the contained value if present, otherwise returns d. Not declared if T is void.
`operator ==`	Compare optional objects

A simple example with `std::expected`

An intro to std::expected

To present `std::expected<T,E>` in action let's write a function to compute a normalized representation of a mathematical vector whose values can be stored in an arbitrary container, such as `std::vector` or `std::array`

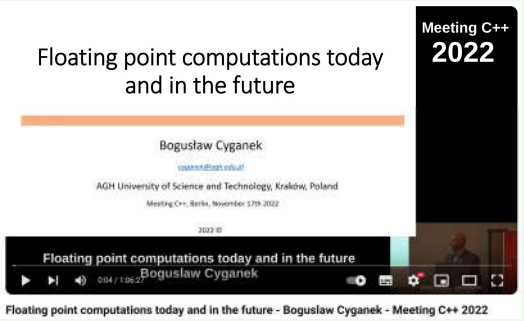
A normalized version of a vector w is obtained by dividing by its module, that is

$$\bar{\mathbf{w}} = \frac{\mathbf{w}}{|\mathbf{w}|} \qquad |\mathbf{w}| = \sqrt{\mathbf{w} \cdot \mathbf{w}}$$

```
enum class ENormErr { kEmptyVec, kZeroSum, kWrongVals };

template < typename T, template < typename > typename Cont = std::vector >
using NormExpected = std::expected< Cont< T >, ENormErr >;
```

An intro to std::expected

```
template < std::floating_point T, auto kThresh = 1e-76 >
auto normalize( std::vector< T > v ) -> NormExpected< T, std::vector >
{
        if( not v.size() )
                return std::unexpected( ENormErr::kEmptyVec );

        auto denom = std::inner_product(    v.begin(), v.end(), v.begin(),
                                            decltype( v )::value_type() );

        if( denom < kThresh )           // check the denominator
                return std::unexpected( ENormErr::kZeroSum );
        else if( std::isinf( denom ) || std::isnan( denom ) )
                return std::unexpected( ENormErr::kWrongVals );

        std::transform(  v.begin(), v.end(), v.begin(),
                        [ sd = std::sqrt( denom ) ] ( auto x ) { return x / sd; } );
        return v;
}
```



Floating point computations today and in the future - Bogusław Cyganek - Meeting C++ 2022

An intro to std::expected

A test function:

```
void ExpectedNormalizeTest()
{
	using DVec = std::vector< double >;		// also good for 'float'
	for( auto v : {	DVec { -1, 0, 1 },
			DVec {  },		// empty
			DVec { 0, 0, 0 },	// too small magnitude
			DVec { 2, 4, 1.f/std::sin(0.f) },	// almost good but inf
			DVec { 5, -5, 7, -7 } } )
	{
```

An intro to std::expected

A test function (ctd.):

```
        if( auto env = normalize( std::move( v ) ); env.has_value() )
        {
                std::copy( env->begin(), env->end(),
                        std::ostream_iterator<DVec::value_type>(std::cout, ", ") );
                std::cout << "\n";
        }
        else switch( env.error() )
        {
                using enum ENormErr;  // introducing ENormErr to this scope
                case kEmptyVec: std::cout << "Error: vec is empty\n";     break;
                case kZeroSum:  std::cout << "Error: module is 0\n";      break;
                case kWrongVals:std::cout << "Error: wrong vals\n";       break;
                default : assert( false );                                break;
        }
    }
}
```

An intro to std::expected

A test function – the results:

```
void ExpectedNormalizeTest()
{
	using DVec = std::vector< double >;		// also good for 'float'
	for( auto v : {	DVec { -1, 0, 1 },
			DVec {  },		// empty
			DVec { 0, 0, 0 },	// too small magnitude
			DVec { 2, 4, 1.f/std::sin(0.f) },// almost good but inf
			DVec { 5, -5, 7, -7 } } )
	{
		if( auto env = normalize( std::move( v ) ); env.has_value() )
		// ...
```

```
-0.707107, 0, 0.707107,

Error: vec is empty

Error: module is 0

Error: wrong vals

0.410997, -0.410997, 0.575396, -0.575396,
```

Returning to the pipeline architecture in C++

A pipeline architecture in C++

A PROBLEM… what to do if one link in the above pipeline cannot complete its operation and transmit its result because an error occurred?

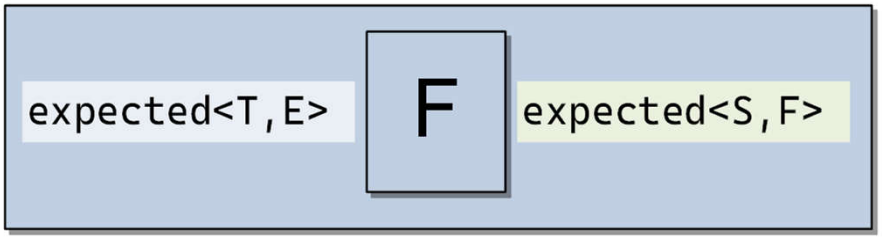
We can throw an exception and interrupt the entire operation.

~~We can use std::optional to express whether the operation was successful and we have the result, or whether we have a situation in which the calculations failed for some reason and we simply cannot provide any result.~~

But if you have a C++23 compiler, an even better option is to use **`std::expected`**. Unlike std::optional, which has been available since C++17, in the event of a calculation failure, it allows you to pass an error code and not just state the failure.

A pipeline architecture in C++

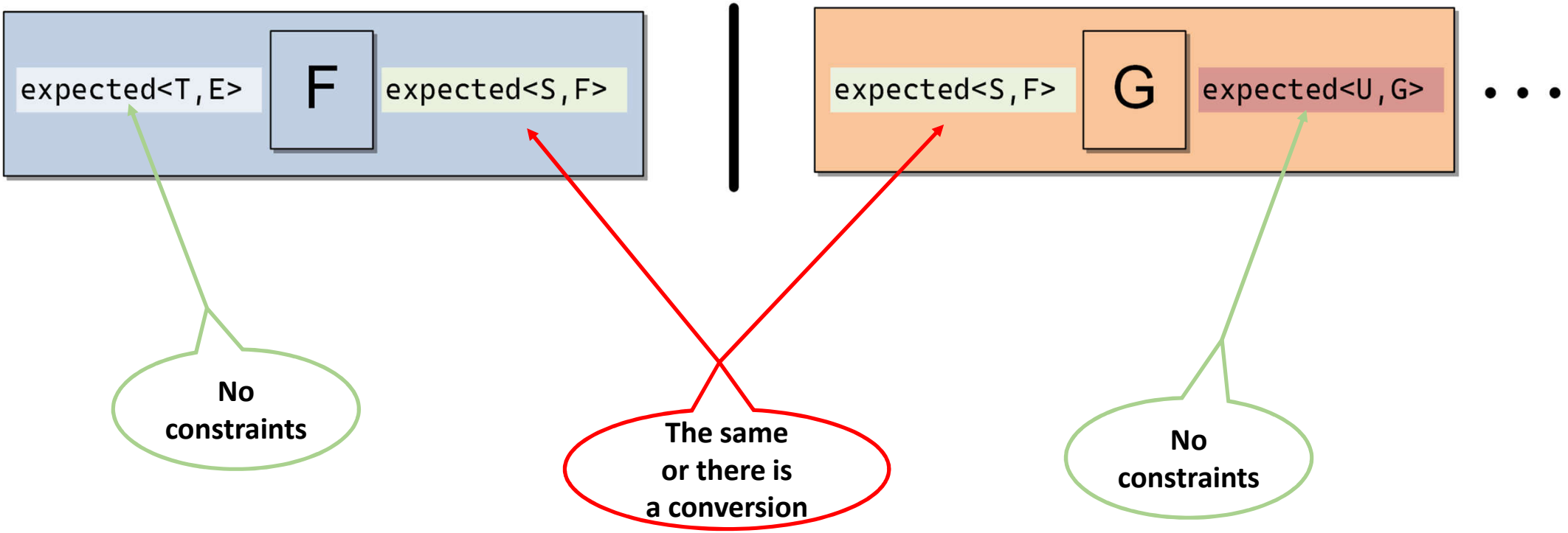
The main idea is to allow any type of `std::expected` on the function input and output



and then to connect such functions in a chain, i.e. a pipeline, with a custom `operator |`

A pipeline architecture in C++

The main idea is to allow any type of `std::expected` on the function input and output



and then to connect such functions in a chain, i.e. a pipeline, with a custom `operator |`

A pipeline architecture in C++

Custom overloaded pipe operator for pipe-line with `std::expected`

```
std::expected< S, F >
```

```
template < typename T, typename E, typename Function >
   requires std::invocable< Function, std::expected< T, E > >
         && is_expected< typename std::invoke_result_t< Function, std::expected< T, E > > >
constexpr auto operator | ( std::expected< T, E > && ex, Function && f )
                          -> typename std::invoke_result_t< Function, std::expected< T, E > >
{
   return std::invoke( std::forward< Function >( f ),
                       std::forward< std::expected< T, E > >( ex ) );
}
```

```
// ... an overloaded version
constexpr auto operator | ( const std::expected< T, E > & ex, Function && f )
                          -> typename std::invoke_result_t< Function, std::expected< T, E > >
{
   return std::invoke( std::forward< Function >( f ), ex );
}
```

A pipeline architecture in C++

Let's define a concept to check for `std::expected`

```
template < typename T >
concept is_expected = requires( T t )
{




};
```

The parameter t of type T is introduced.

A pipeline architecture in C++

Let's define a concept to check for `std::expected`

```
template < typename T >
concept is_expected = requires( T t )
{
        typename T::value_type;       // type requirement – nested member name exists
        typename T::error_type;       // type requirement – nested member name exists




};
```

The parameter t of type T is introduced.
It is checked that type T defines `value_type`, as well as `error_type`.

A pipeline architecture in C++

Let's define a concept to check for `std::expected`

```
template < typename T >
concept is_expected = requires( T t )
{
        typename T::value_type;       // type requirement – nested member name exists
        typename T::error_type;       // type requirement – nested member name exists

        requires std::is_constructible_v< bool, T >;
        requires std::same_as< std::remove_cvref_t< decltype(*t) >, typename T::value_type >;

        requires std::constructible_from< T, std::unexpected< typename T::error_type > >;
};
```

The parameter t of type T is introduced.
It is checked that type T defines `value_type`, as well as `error_type` on line.
The series of three nested requirements begins

A pipeline architecture in C++

Let's define a concept to check for `std::expected`

```
template < typename T >
concept is_expected = requires( T t )
{
        typename T::value_type;       // type requirement – nested member name exists
        typename T::error_type;       // type requirement – nested member name exists

        requires std::is_constructible_v< bool, T >;
        requires std::same_as< std::remove_cvref_t< decltype(*t) >, typename T::value_type >;

        requires std::constructible_from< T, std::unexpected< typename T::error_type > >;
};
```

The parameter t of type T is introduced.
It is checked that type T defines value_type, as well as error_type on line.
The series of three nested requirements begins

What is characteristic of them is the first word requires. Inserting the keyword `requires` forces the compiler *to check what the value of this expression* actually is – if it is `true`, then the concept is fulfilled.

Returning to the pipeline architecture in C++

A pipeline architecture in C++

Word2Vec

A problem – how to find similar texts or similar images?

Convert text to the semantic representation ➔ embedding vectors (Large Language Models).

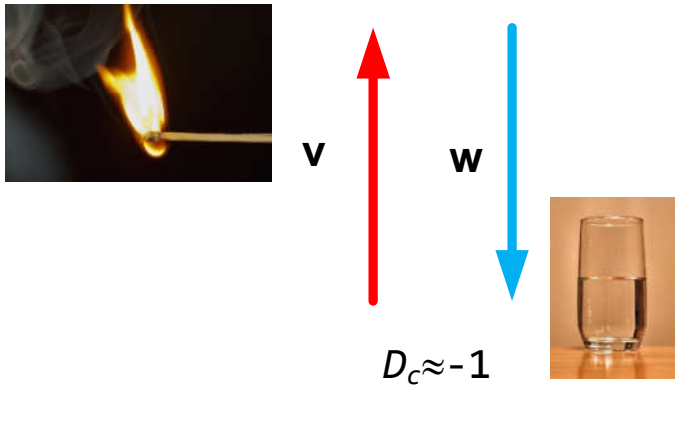
Convert images to the semantic representations ➔ embedding vectors (Visual Transformer).

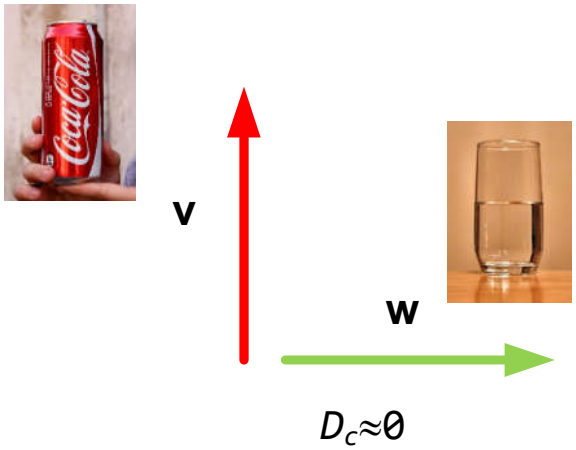
Then – compare vectors, e.g. with the cosine measure:

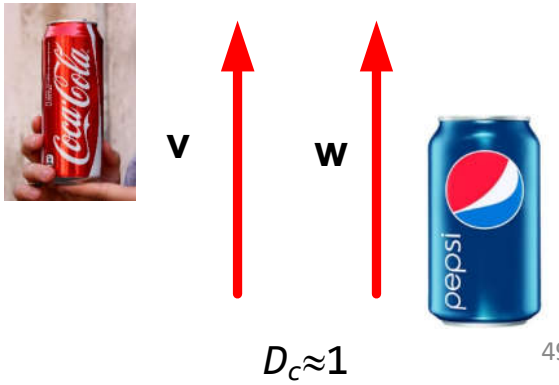
$$D_c(\mathbf{v},\mathbf{w}) = \cos(\mathbf{v},\mathbf{w}) = \frac{\mathbf{v}\cdot\mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \left(\frac{\mathbf{v}}{|\mathbf{v}|}\right)\cdot\left(\frac{\mathbf{w}}{|\mathbf{w}|}\right) = \bar{\mathbf{v}}\cdot\bar{\mathbf{w}}$$

normalized

The inner product between normalized vectors conveys a similarity measure D_c

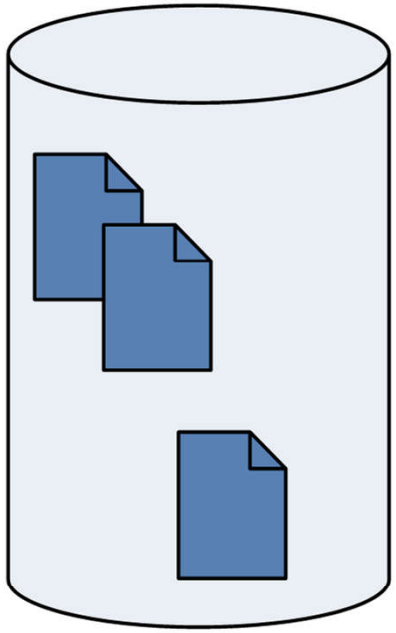






A pipeline architecture in C++

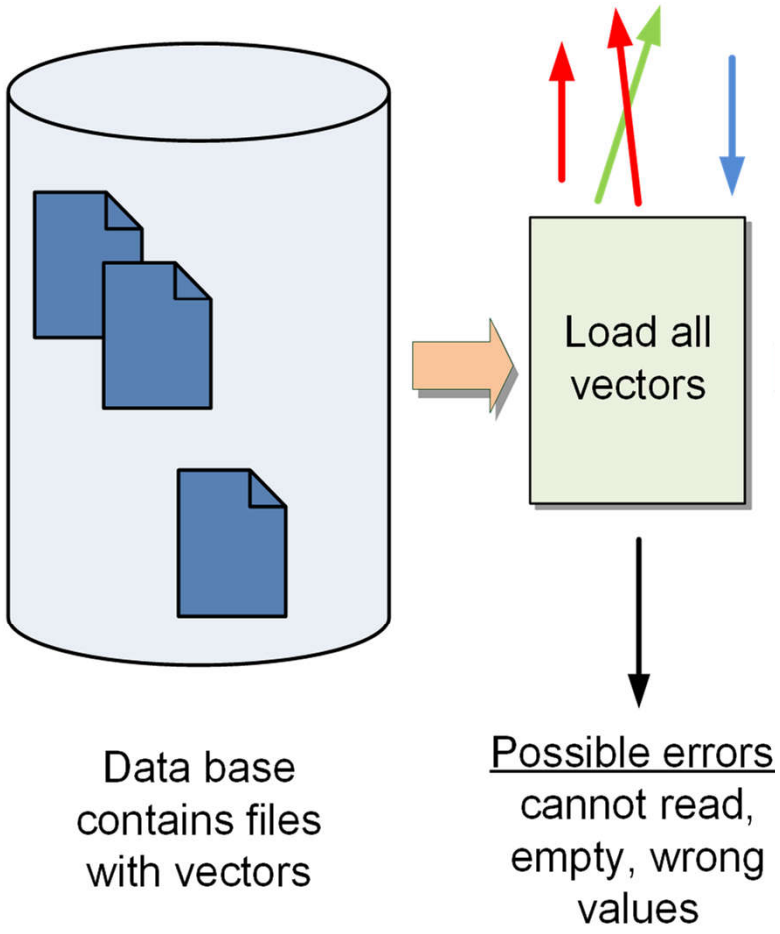
After computing **word embeddings** (vectors) our framework may look as follows...



Data base
contains files
with vectors

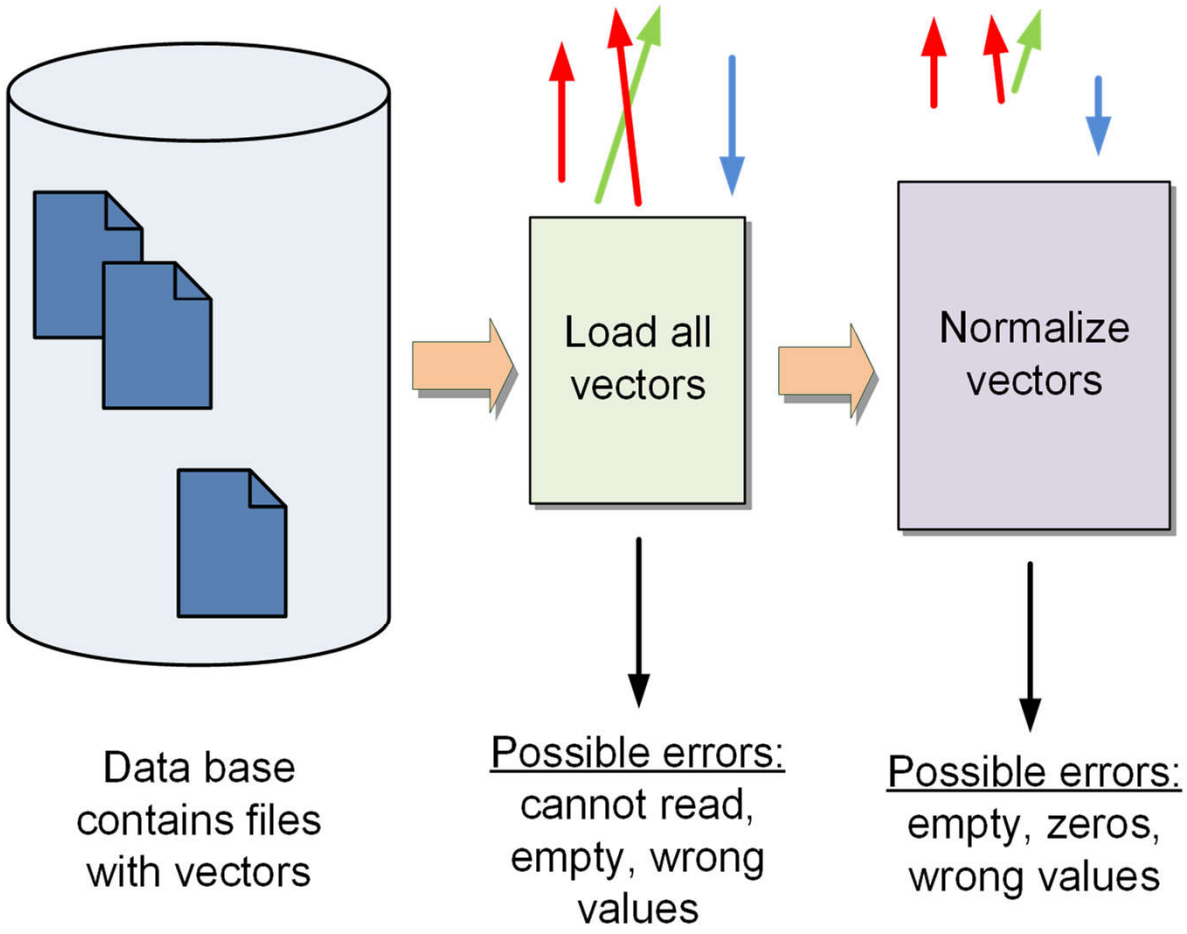
A pipeline architecture in C++

After computing **word embeddings** (vectors) our framework may look as follows...



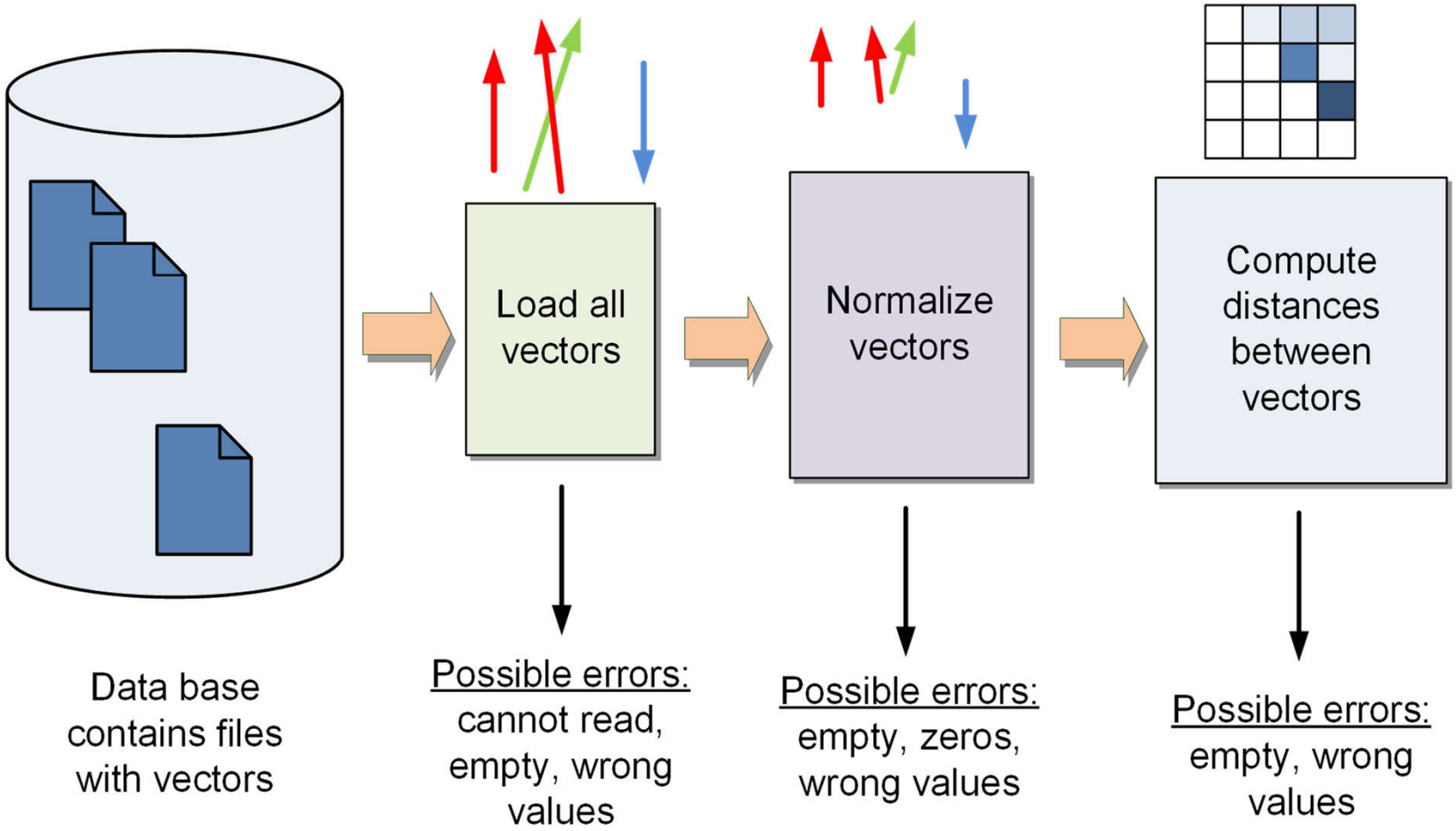
A pipeline architecture in C++

After computing **word embeddings** (vectors) our framework may look as follows...



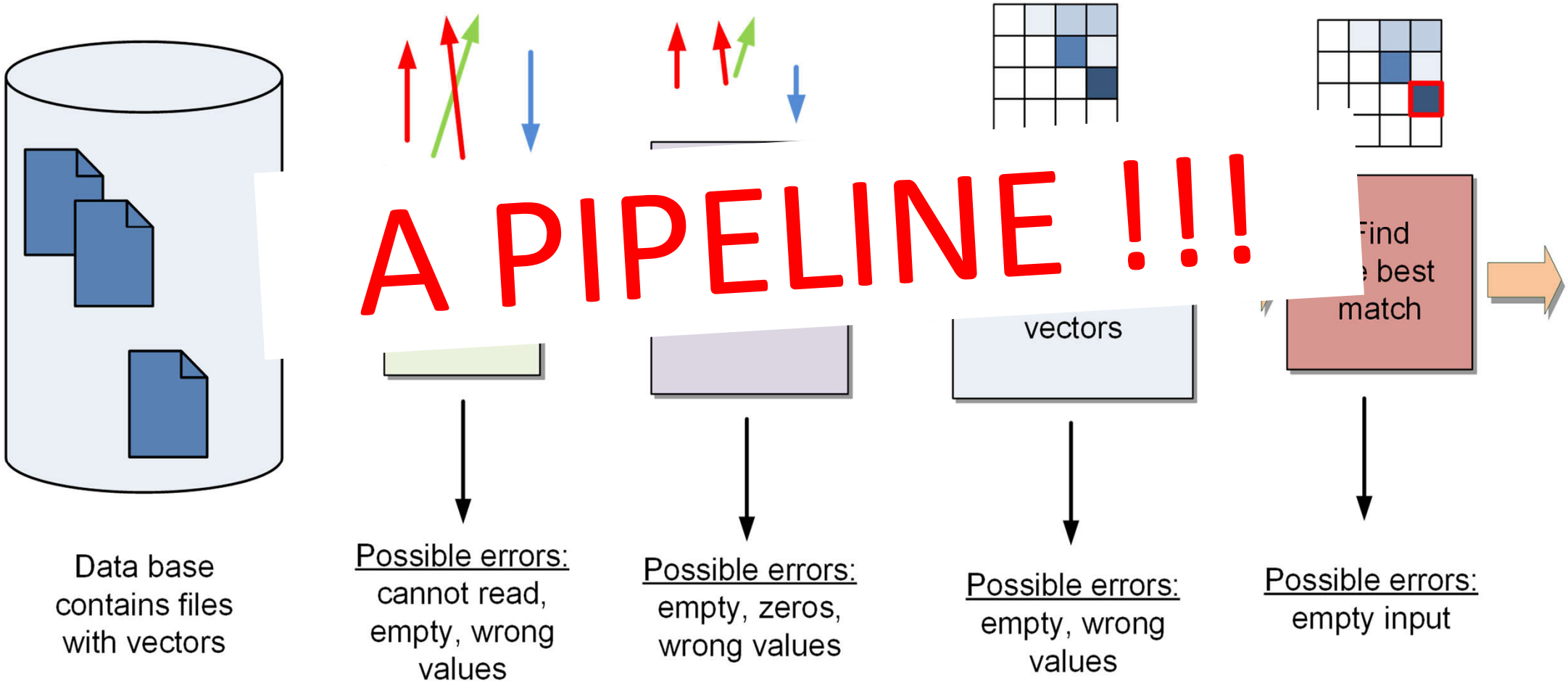
A pipeline architecture in C++

After computing **word embeddings** (vectors) our framework may look as follows...



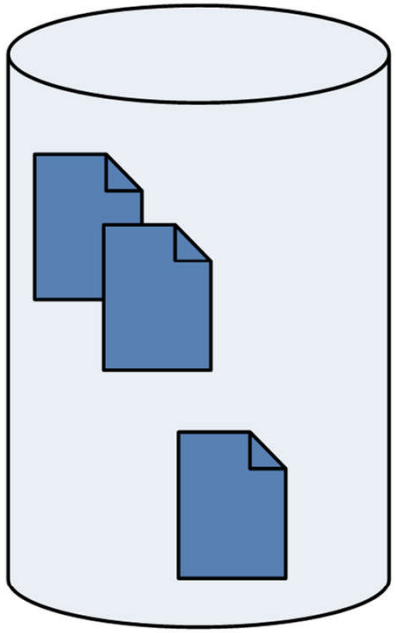
A pipeline architecture in C++

After computing **word embeddings** (vectors) our framework may look as follows...



Our pipelined code framework may look as follows...

After computing **word embeddings** (vectors) our framework may look as follows...



Data base
contains files
with vectors

Our pipelined code framework may look as follows...

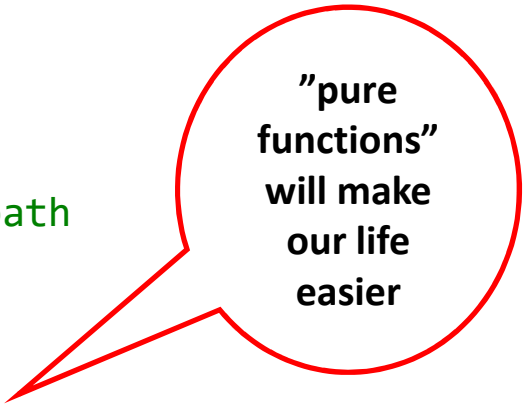
Our code may look as follows...

```
enum class PathErr { kEmpty };        // no path provided
using path_exp = std::expected< std::filesystem::path, PathErr >;


enum class LoadErr { kNoData, kWrongPath, kCannotOpen, kWrongData };
using load_exp = std::expected< PathVec, LoadErr >;
```

Our pipelined code framework may look as follows...

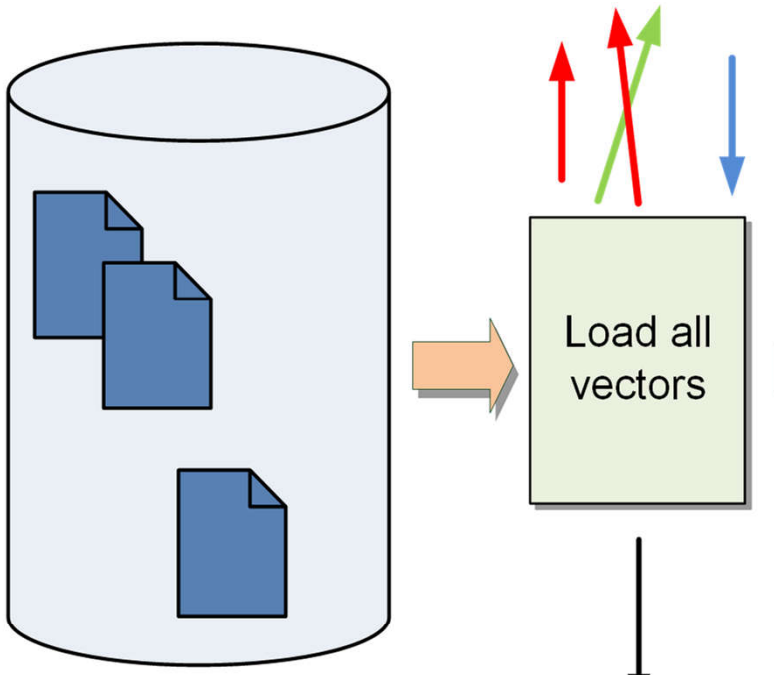
Our code may look as follows...

```
// traverse and collect all paths in this directory of files with the "accept_ext" extension
load_exp load_paths( path_exp && pe, const std::filesystem::path & accept_ext )
{
   if( ! pe )// if no objects to process, then exit passing an error
      return std::unexpected( LoadErr::kWrongData );

   if( ! fs::exists( * pe ) || ! fs::is_directory( * pe ) )
      return std::unexpected( LoadErr::kWrongPath );// exit if wrong path

   // Iterate through the directory
   load_exp retExp {};
   for( const auto & file_obj : fs::directory_iterator( * pe ) )
      if( fs::is_regular_file( file_obj ) )
         if( file_obj.path().extension().string().contains( accept_ext.string() ) )
            retExp->push_back( file_obj.path() );

   return retExp->size() > 0 ? retExp : std::unexpected { LoadErr::kNoData };
}
```



Our pipelined code framework may look as follows...

After computing **word embeddings** (vectors) our framework may look as follows...



Our pipelined code framework may look as follows...

Our code may look as follows...

```
using vec_vec_exp = std::expected< VecOfVec, ENormErr >;
```

Our pipelined code framework may look as follows...

```
// open all files and read the vectors
vec_vec_exp load_vectors( load_exp && le )
{
   if( ! le )   // if no objects to process, then exit passing an error
      return std::unexpected( ENormErr::kEmptyVec );

   // open each file and read vectors
   VecOfVec retVecs;
   for( const auto & f : * le ) {
      if( std::ifstream inFile( f ); inFile.is_open() ) {
         for( std::string str; std::getline( inFile, str ) && str.length() > 0; ) {
            std::istringstream istr( str );
            using DType_Iter = std::istream_iterator< DType >;

            retVecs.emplace_back( std::make_move_iterator( DType_Iter{ istr } ),
                                  std::make_move_iterator( DType_Iter{} ) );
         }
      }
   }

   return retVecs.size() > 0 ? vec_vec_exp {retVecs} : std::unexpected(ENormErr::kEmptyVec);
}
```

A pipeline architecture in C++

But let's stop for a moment and think about the interface…

- PROBLEM: We have a sequence of functions in a pipeline, each with a different interface, so how do we communicate error information? How do we make these interfaces compatible? Especially the branch that carries a potential error In C++ `enum class` cannot be inherited ...

- SOLUTION: Use `std::variant` to hold a union of error types for the entire pipeline.

A pipeline architecture in C++

`std::variant` allows us to hold any type, but only one at a time - and that's what we need

```
using common_errors= std::variant< PathErr, LoadErr, ENormErr, DistErr >;

using path_com_exp    = std::expected< std::filesystem::path,     common_errors >;
using load_com_exp    = std::expected< PathVec,                   common_errors >;
using vec_vec_com_exp = std::expected< VecOfVec,                  common_errors >;
using dist_com_exp    = std::expected< Matrix,                    common_errors >;
using max_com_exp     = std::expected< index_val,                 common_errors >;
```

So we need to redesign our interfaces a bit - so let's go back to the features we just saw and analyze what...

Our pipelined code framework may look as follows...

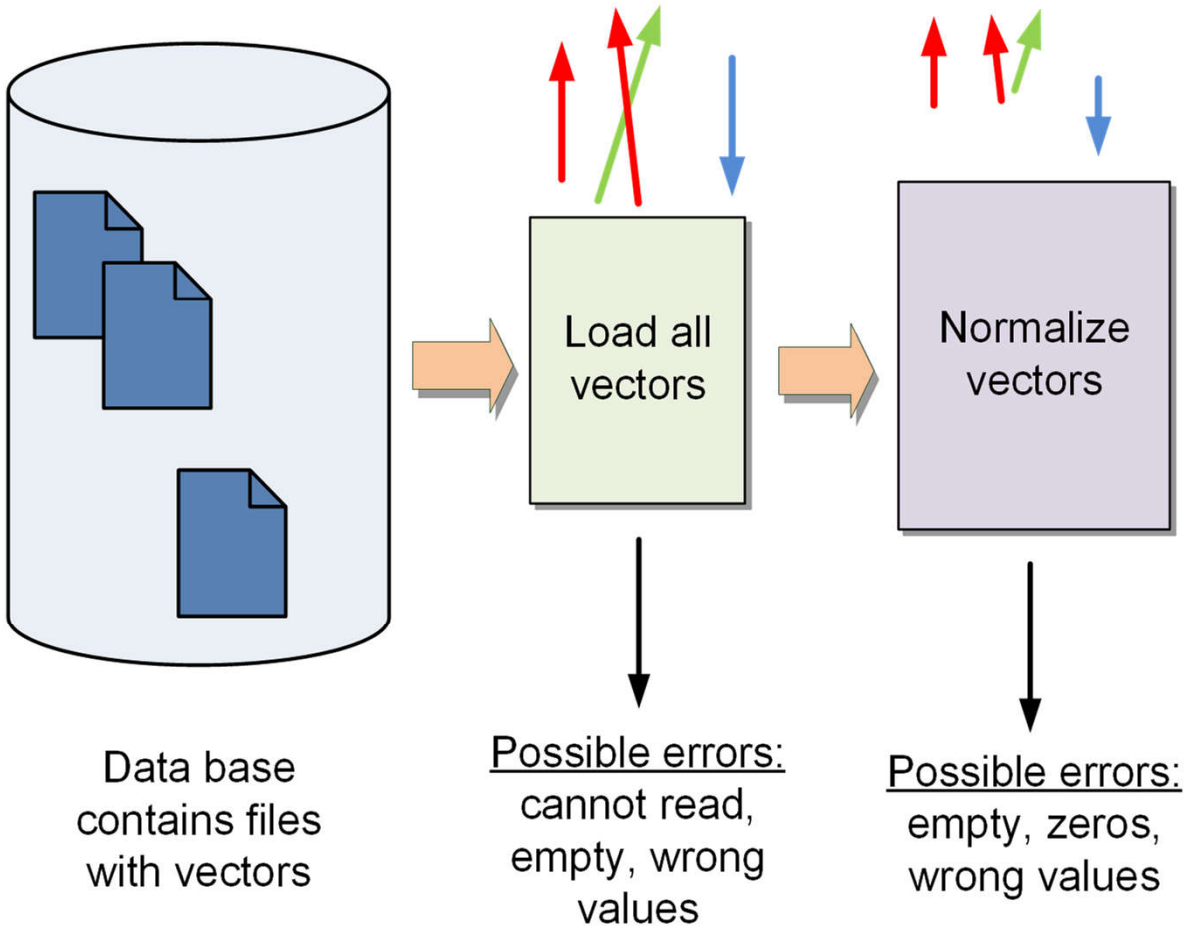
Our code may look as follows...

```
// traverse and collect all paths in this directory of files with the "accept_ext" extension
load_exp load_paths( path_exp && pe, const std::filesystem::path & accept_ext )
{
   // ...
```

```
// traverse and collect all paths in this directory of files with the "accept_ext" extension
load_com_exp load_paths( path_com_exp && pe, const std::filesystem::path & accept_ext )
{
   // ...
```

Our pipelined code framework may look as follows...

After computing **word embeddings** (vectors) our framework may look as follows...



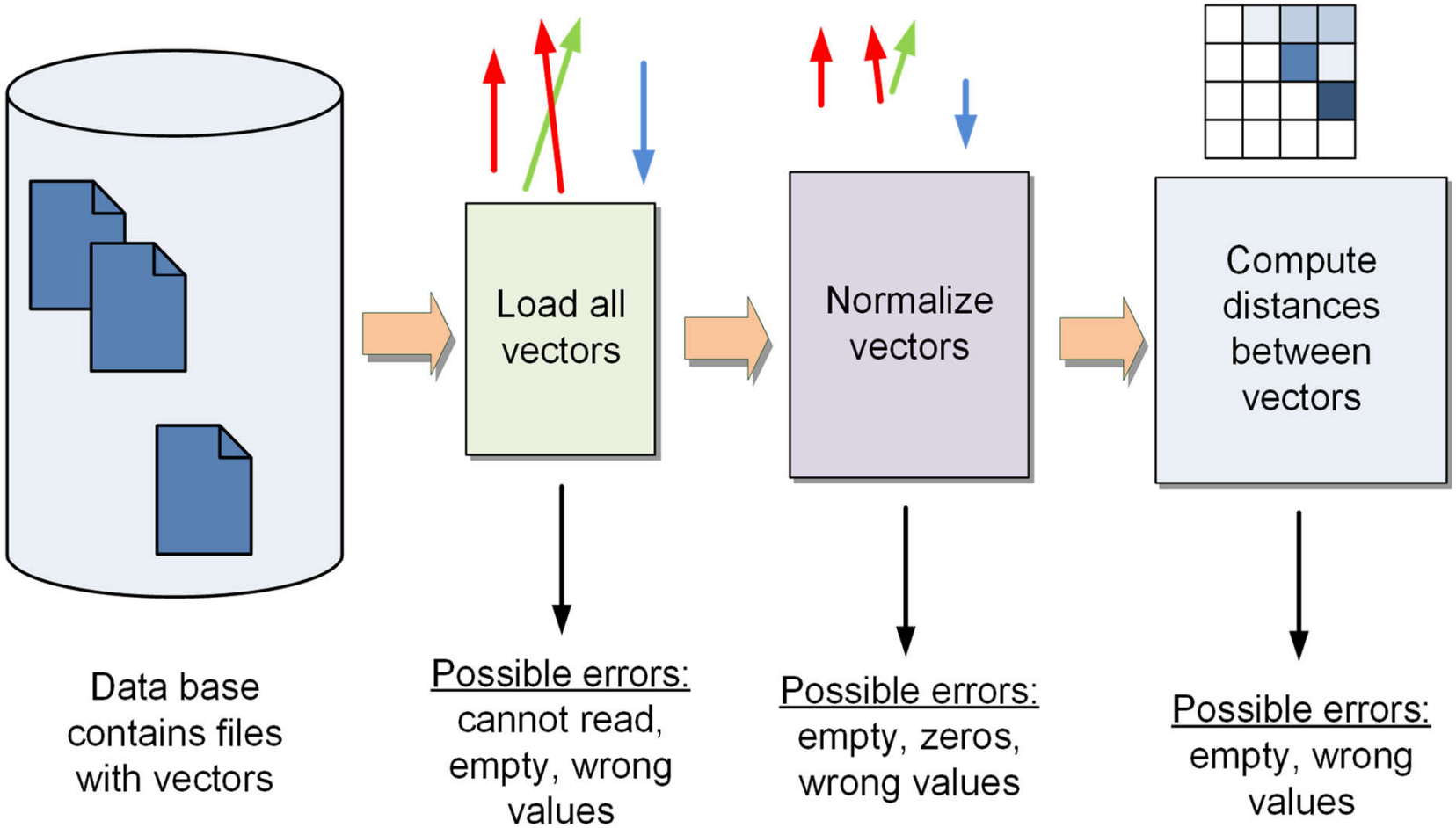
Our pipelined code framework may look as follows...

Our code may look as follows...

```
vec_vec_com_exp vec_normalize(vec_vec_com_exp && vve )
{
   if( ! vve )// if no objects to process, then pass the error
      return std::unexpected( vve.error() );

   for( auto && v : * vve )
      if( auto env = normalize< DVec::value_type, std::vector >( v ); env.has_value() )
         v = env.value();
      else
         return std::unexpected( env.error() );// stop immediately and return the error

   return vve;
}
```

Our pipelined code framework may look as follows...

After computing **word embeddings** (vectors) our framework may look as follows...

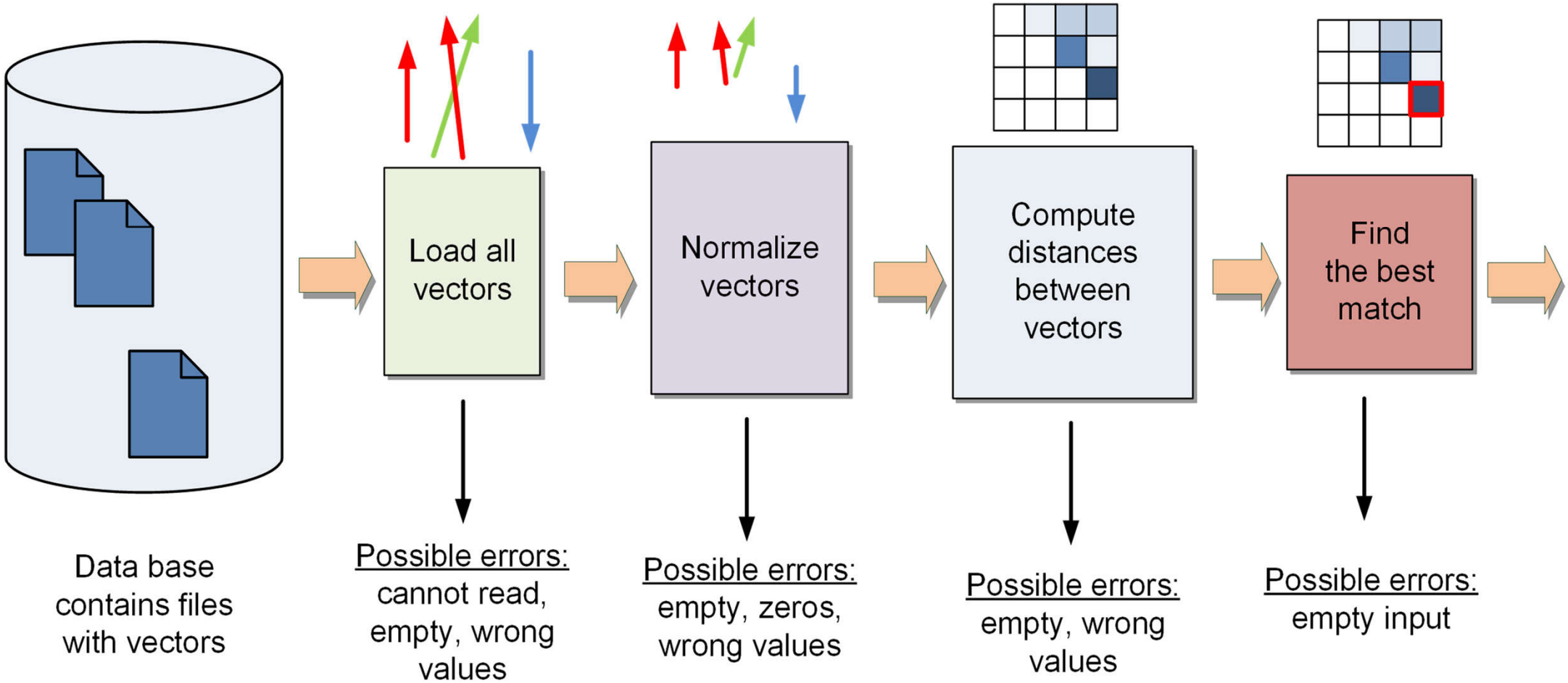


Our pipelined code framework may look as follows...

```
// Computes a cosine distance between vectors
// We assume that the input vectors are already normalized
dist_com_exp comp_distance(vec_vec_com_exp && vve )
{
   if( ! vve )// if no objects to process, then exit passing an error
      return std::unexpected( DistErr::kWrongData ); // THIS TIME WE ISSUE AN ERROR

   const auto kColsRows { vve->size() }; // it's a square matrix
   if( kColsRows == 0 )
      return std::unexpected( DistErr::kZeroLen );

   Matrix distances( kColsRows, DVec( kColsRows, DVec::value_type {} ) );

   for( auto r { DVec::size_type {} }; r < kColsRows - 1; ++ r ) {
      const auto & v = ( * vve )[ r ];
      for( auto c { r + 1 }; c < kColsRows; ++ c ) {
         distances[ r ][ c ] = std::inner_product( v.begin(), v.end(),
                  ( * vve )[ c ].begin(), std::remove_cvref_t< decltype(v) >::value_type {} );
      }
   }
   return distances;
}
```

Our pipelined code framework may look as follows...

After computing **word embeddings** (vectors) our framework may look as follows...



Our pipelined code framework may look as follows...

Our code may look as follows...

```
using index_val = std::tuple< Matrix::size_type, Matrix::size_type, DType >;

using max_com_exp = std::expected< index_val, common_errors >;
```

Our pipelined code framework may look as follows...

```
max_com_exp find_max( dist_com_exp && de )
{
   if( ! de ) // if no objects to process, then exit passing an error
      return std::unexpected( de.error() );

   const auto kColsRows { de->size() }; // it's a square matrix
   assert( kColsRows > 0 );
   assert( (*de)[ 0 ].size() > 0 );

   constexpr DType kNoneVal { std::numeric_limits< DType >::lowest() };
   index_val ret { 0u, 0u, kNoneVal };
   for( Matrix::size_type r {}; r < kColsRows - 1; ++ r ) {
      for( Matrix::size_type c { r + 1 }; c < kColsRows; ++ c ) {
         auto val = (*de)[ r ][ c ];
         assert( val >= -1.1 && val <= +1.1 );

         if( std::get< 2 >( ret ) < val )
            ret = { r, c, val };
      }
   }
   return std::get<2>(ret) != kNoneVal?max_com_exp{ret} : std::unexpected(DistErr::kWrongData);
}
```

Our pipelined code framework may look as follows...

Our main function with the pipeline.

```
const auto kFileExt { "txt"sv };
 void GenPipeTest()
 {
    const auto kFExt { "txt"sv };

    // Here we create our CUSTOM PIPE
    auto result = path_exp( ".\\..\\data"sv )
              | [ ext = kFExt] ( auto && pe ) { return load_paths( std::move( pe ), ext ); }
              | load_vectors
              | vec_normalize
              | comp_distance
              | find_max
              | [] ( auto && exp_2_check ) { // Check the final result
                    if( exp_2_check )
                       // ...
```

Function currying

Haskell Brooks Curry

https://en.wikipedia.org/wiki/Currying

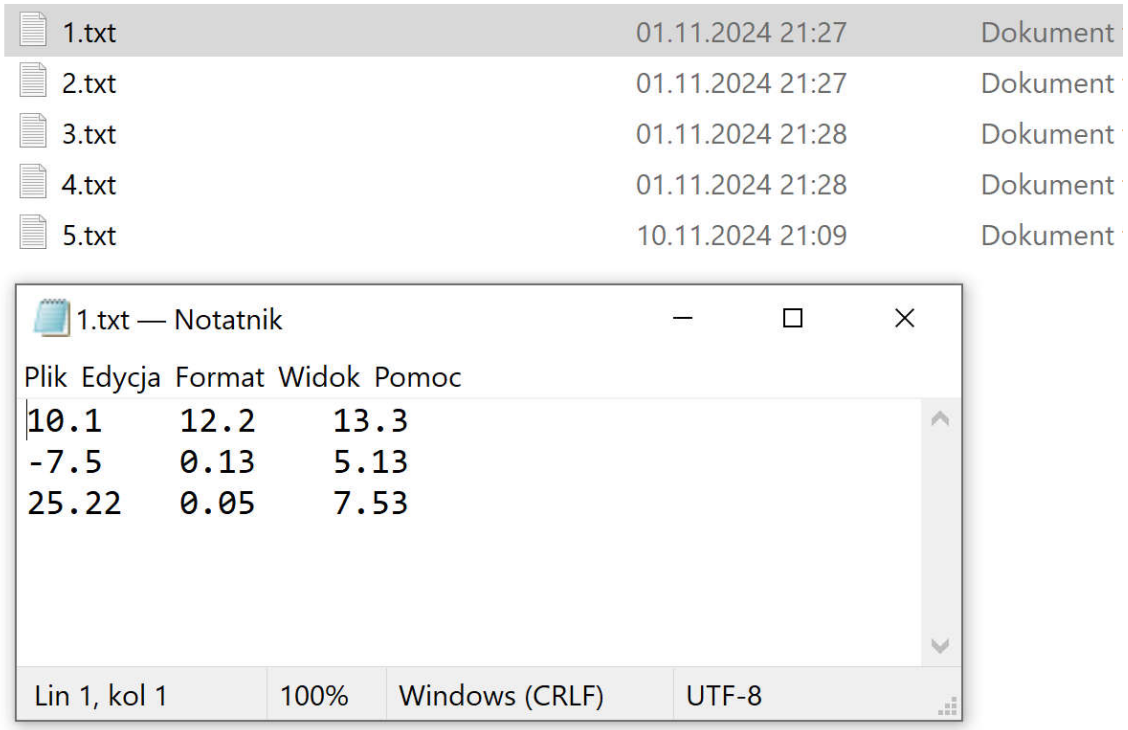
Our pipelined code framework may look as follows...

```
    // ...
        | [] ( auto && exp_2_check ) { // Check the final result
        if( exp_2_check ) {
                auto [ x, y, v ] = * exp_2_check;
                std::println( "success @ idx=({},{}; val={:.3f})\n", x, y, v );
        }
        else {
                std::visit( overloaded {
                  []( PathErr pe )  { std::println( "PathErr #{}",  static_cast<int>(pe) ); },
                  []( LoadErr le )  { std::println( "LoadErr #{}",  static_cast<int>(le) ); },
                  []( ENormErr ne ) { std::println( "ENormErr #{}", static_cast<int>(ne) ); },
                  []( DistErr de )  { std::println( "DistErr #{}",  static_cast<int>(de) ); }
                }, exp_2_check.error() );
     }

return exp_2_check;
}
```

```
template<class... Ts>
struct overloaded : Ts...
{ using Ts::operator()...; };
```

A pipeline architecture in C++

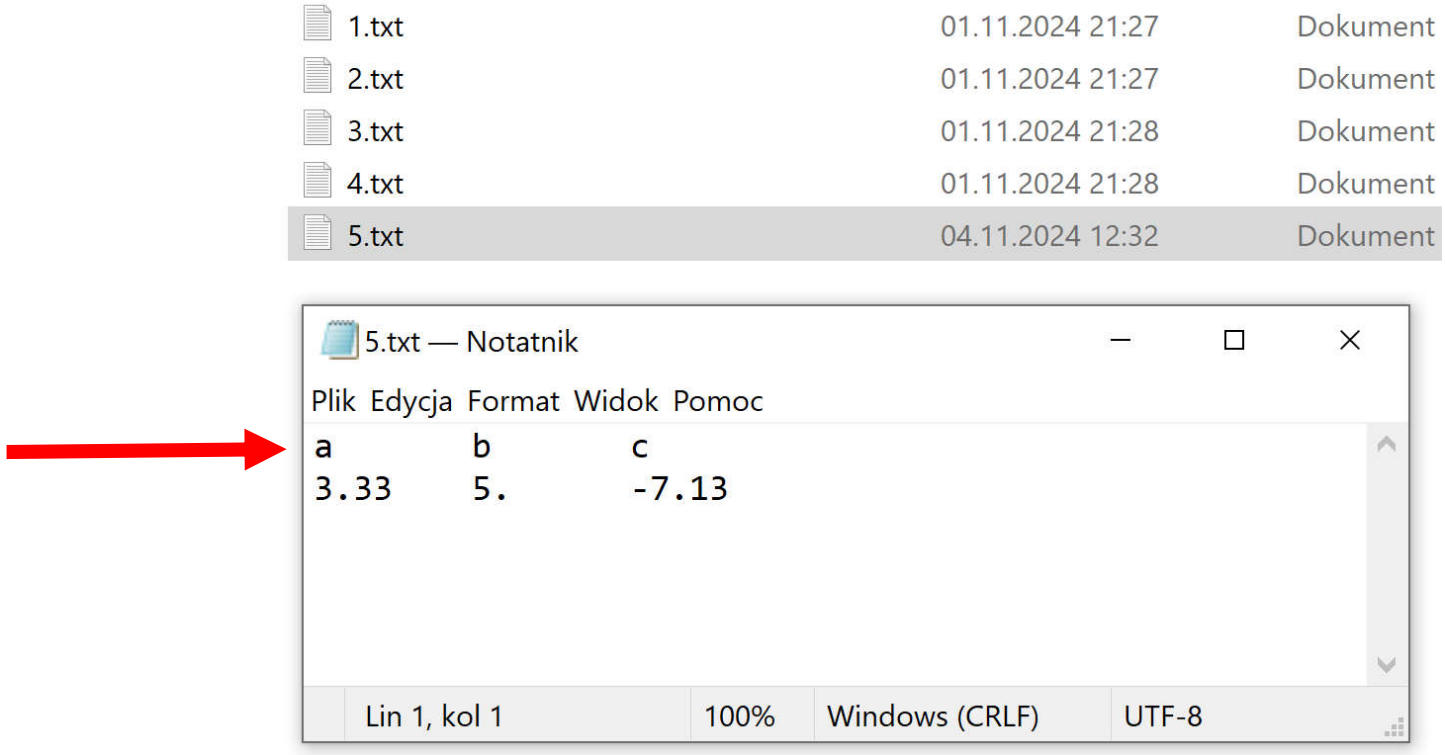
An application and its results ...




```
success @ idx=(3,4; val=0.993)

typeid( result ).name() == class std::expected<class std::tuple<unsigned __int64,unsigned __int64,double>,enum VectorsPipeTest::DistErr>
```

A pipeline architecture in C++

An application and its results ...

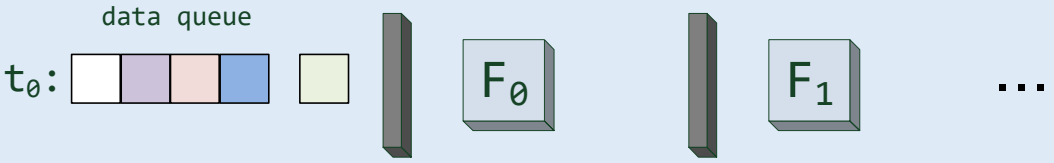


`DistErr::kWrongData`

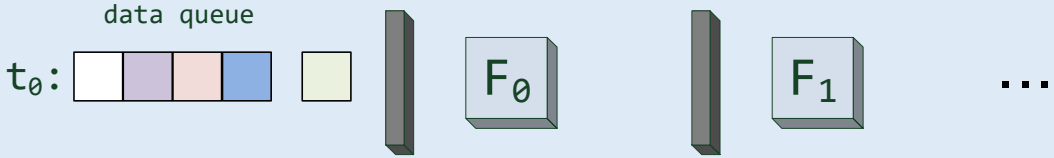
There is more on pipelines…
a parallel version

Parallel pipelines

Switching from serial

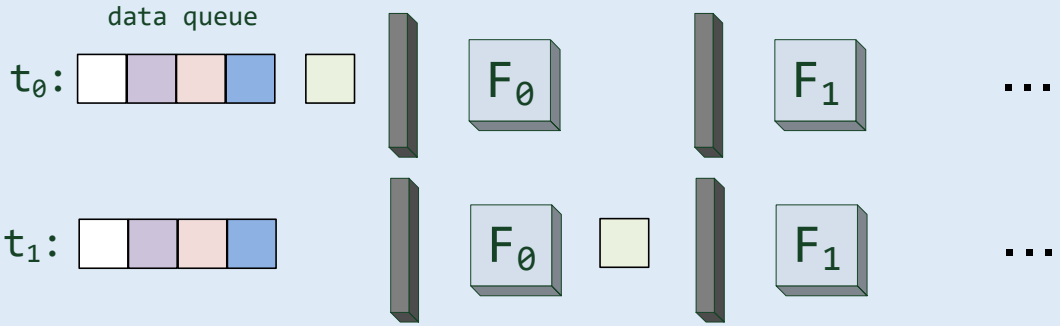


to parallel …

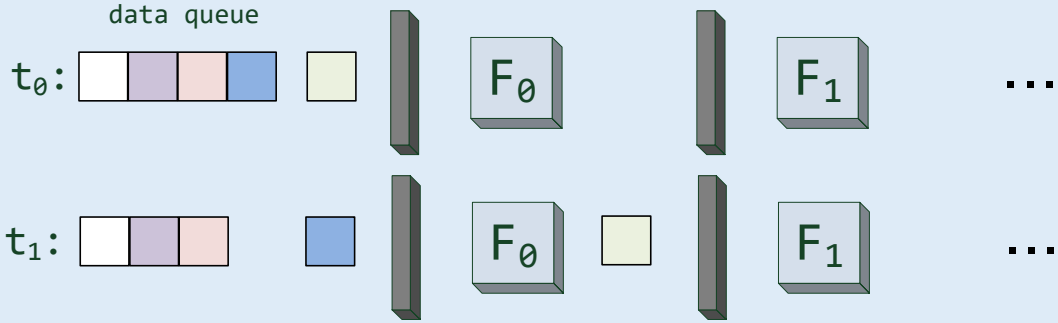


Parallel pipelines

Switching from serial

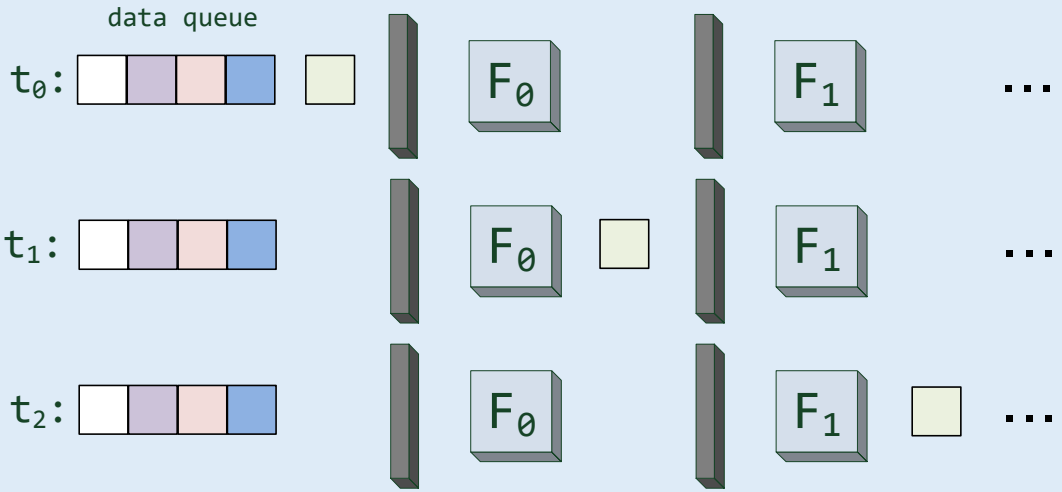


to parallel …

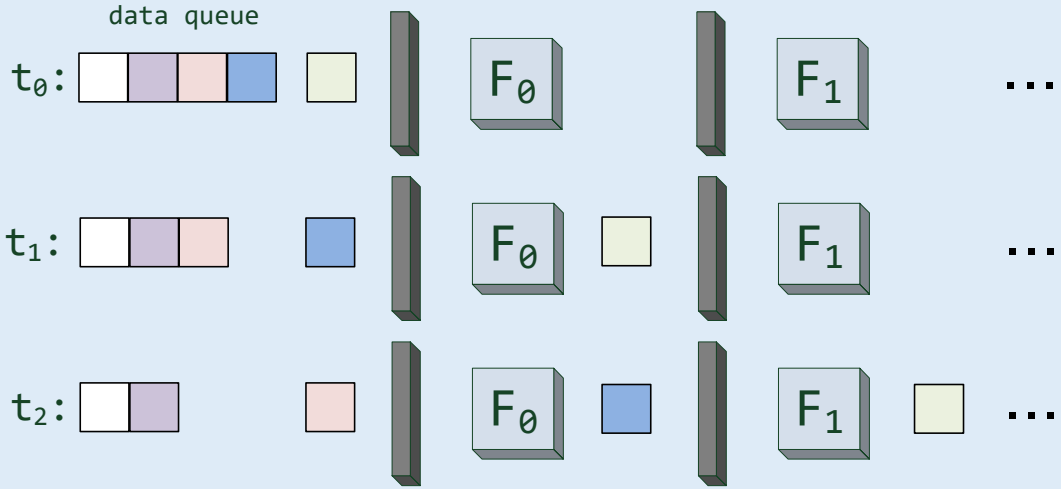


Parallel pipelines

Switching from serial

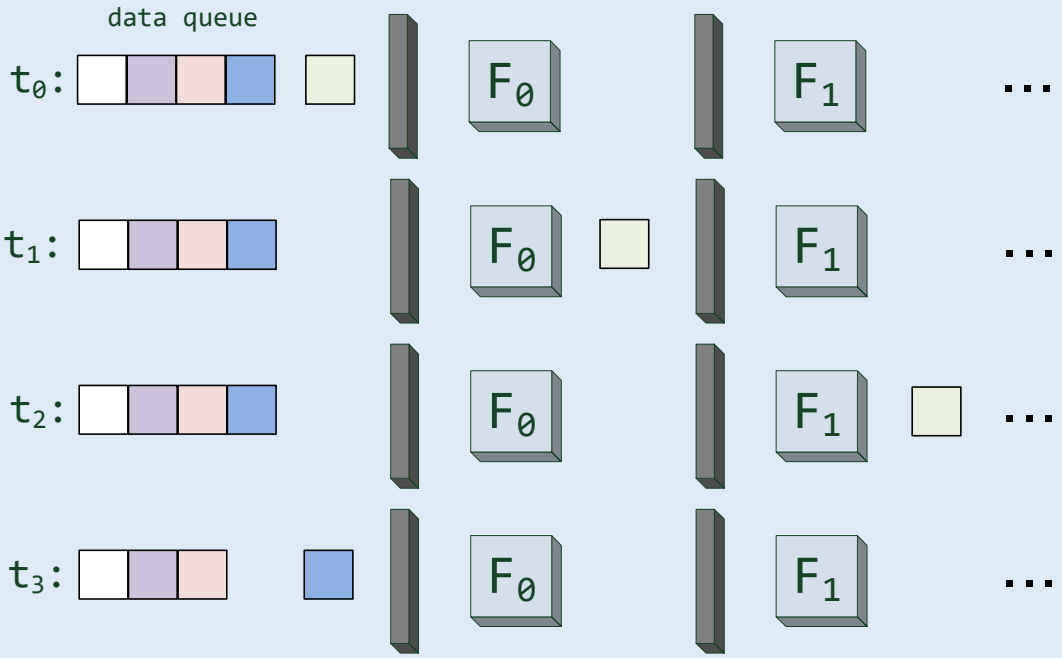


to parallel …

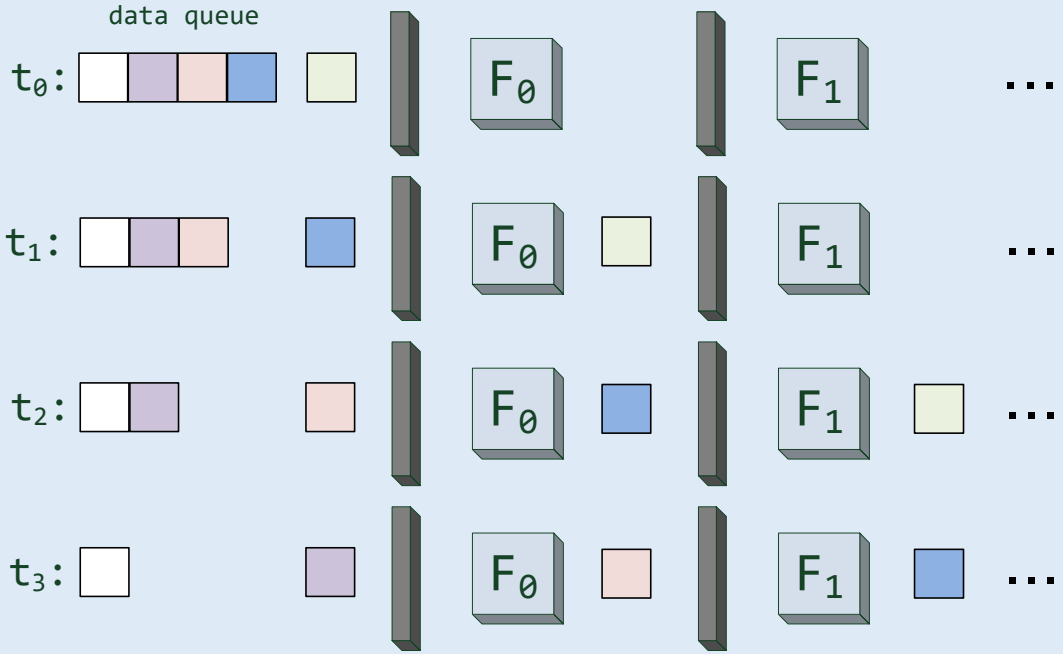


Parallel pipelines

Switching from serial

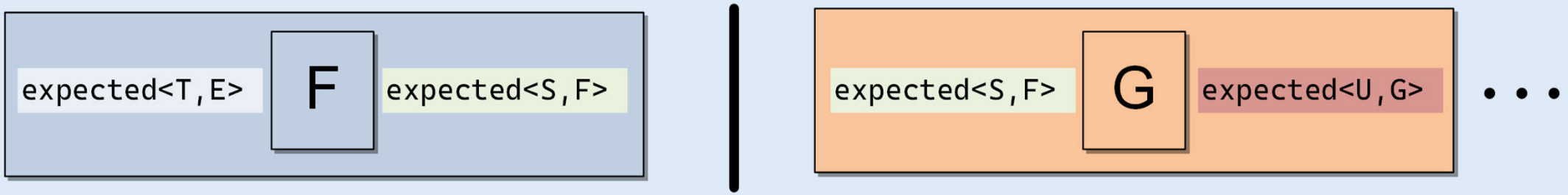


to parallel …



Parallel pipelines

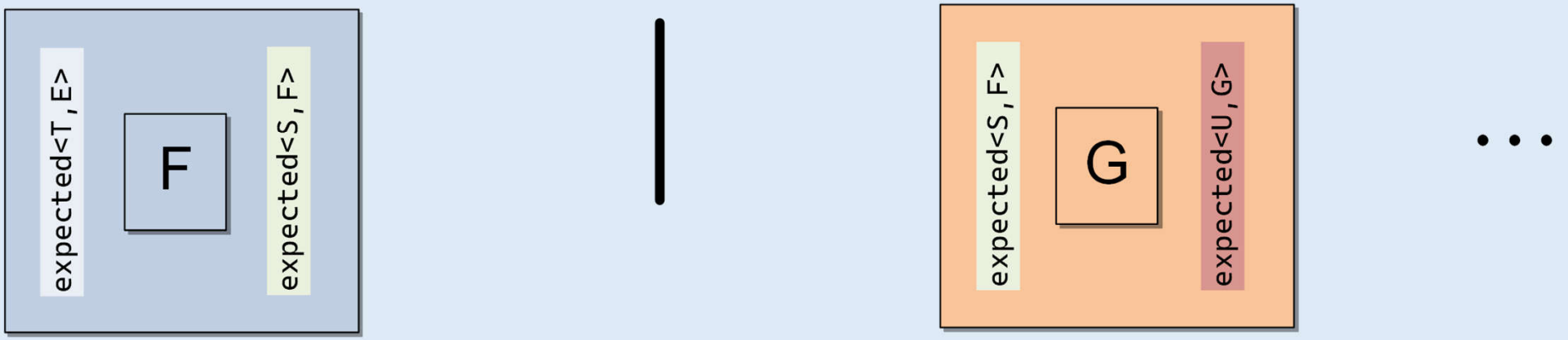
The serial version looked as follows



`operator |`

Parallel pipelines

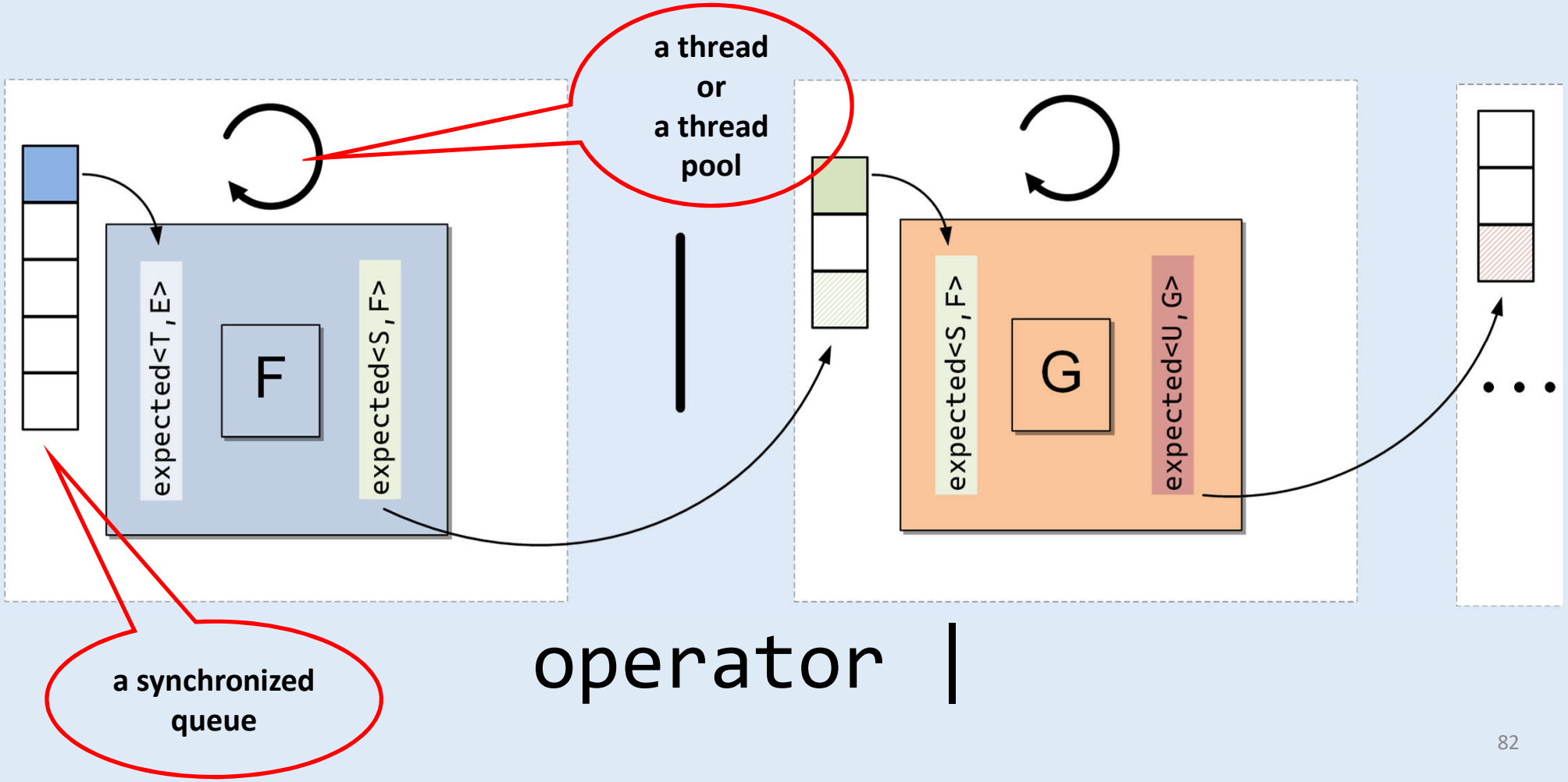
Let’s re-organize our architecture ...



`operator |`

Parallel pipelines

The parallel version can be organized as follows



Parallel pipelines

We have at least THREE things to solve

- THE OVERLOADED PIPE **`operator |`**: This time it has to deal with threads and queues and user defined operations.

- THE THREAD(S): For each node in the parallel pipeline there need to be a thread or threads that perform a task provided by the user.

- THE QUEUE(S): Each thread in a node has an access to two queues: INPUT and OUTPUT, and so on. The queues must be synchronized.

Parallel pipelines

We have at least THREE things to solve

- THE OVERLOADE PIPE **`operator |`**: This time it has to deal with threads and queues and user defined operations.

- THE THREAD(S): For each node in the parallel pipeline there need to be a thread or threads that perform a task provided by the user.

- THE QUEUE(S): Each thread in a node has an access to two queues: INPUT and OUTPUT, and so on. The queues must be synchronized.

Parallel pipelines – operator |

```
template < typename T, typename E, typename CartFun >
   requires std::invocable< CartFun, std::expected< T, E > >
            && is_expected< std::invoke_result_t< CartFun, std::expected< T, E > > >
auto operator | ( std::shared_ptr< TExpectSynchroQueue< std::expected< T, E > > > in_queue_sp,
                  CartFun && cf )
-> std::shared_ptr< TExpectSynchroQueue< std::invoke_result_t< CartFun,
                                                              std::expected< T, E > > > >
{
   assert( in_queue_sp );
   //                                                  out std::expected type
   auto out_queue_sp( std::make_shared< TExpectSynchroQueue<
                      std::invoke_result_t< CartFun, std::expected< T, E > > >() );

   // Pass cf by value (a copy) - if more threads each should have its own version
   std::jthread theThread( [ in_queue_sp, out_queue_sp, cf ] ()
                           { ParPipe_Fun( in_queue_sp, out_queue_sp, std::move( cf ) ); } );
   theThread.detach(); // Let it run separately
   return out_queue_sp;
}
```

This lambda, being const by default, deals well with shared_ptr since the object reference counter is a different object

Parallel pipelines

We have at least THREE things to solve

- THE OVERLOADE PIPE `operator |`: This time it has to deal with threads and queues and user defined operations.

- THE THREAD(S): For each node in the parallel pipeline there need to be a thread or threads that perform a task provided by the user.

- THE QUEUE(S): Each thread in a node has an access to two queues: INPUT and OUTPUT, and so on. The queues must be synchronized.

Decisssion we make:
Only ONE thread per NODE

Parallel pipelines – thread management

A possible implementation of the thread function

```
template < typename InQ_SP, typename OutQ_SP, typename CartFun >
void ParPipe_Fun( InQ_SP in_q, OutQ_SP out_q, CartFun && theCartridgeFun )
{
   try
   {
      for( ;; )
      {
         auto in_expected_elem = in_q->pop_wait_can_block();

         if( in_expected_elem )
         {
            out_q->push_elem( std::invoke( std::forward< CartFun >( theCartridgeFun ),
                              std::move( * in_expected_elem ) ) );
         }
         else
         {
```

Parallel pipelines – thread management

A possible implementation of the thread function (ctd.)

```
            else
            {
               // it should be a sentinel
               assert( in_expected_elem.error() == EQueueErr::kSentinelElem );
               out_q->push_sentinel();

               break;// and exit this thread
            }

         }
      }
      catch( ... )
      {
         if( out_q )
            out_q->push_sentinel();
      }

}
```

Parallel pipelines

We have at least THREE things to solve

- THE OVERLOADE PIPE **`operator |`**: This time it has to deal with threads and queues and user defined operations.

- THE THREAD(S): For each node in the parallel pipeline there need to be a thread or threads that perform a task provided by the user.

- THE QUEUE(S): Each thread in a node has an access to two queues: INPUT and OUTPUT, and so on. The queues must be synchronized.

We need to add at least TWO add-ons

Parallel pipelines – thread management

ADD-ON 1: Implementation of the overloaded **operator |** to synchronize the pipeline

```
class SynchroSentinel
{
};


template < typename T, typename E >
auto operator | ( std::shared_ptr<
                    TExpectSynchroQueue< std::expected< T, E > > >  in_queue_sp,
                    SynchroSentinel )
{
   assert( in_queue_sp );
   in_queue_sp->wait_for_sentinel_can_block();
   // Now we are synchronized - we can run a function on the queue and return a new queue
   return in_queue_sp;
}
```

Parallel pipelines – thread management

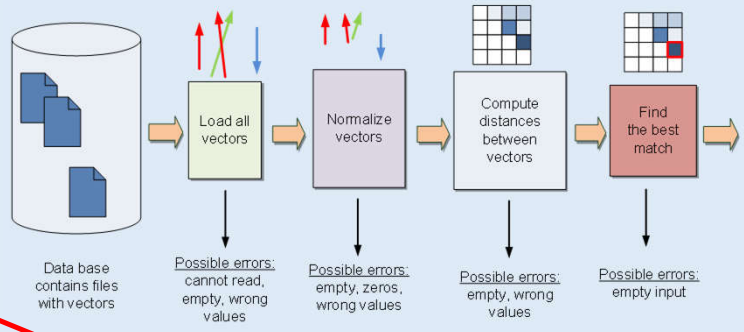
ADD-ON 2: Implementation of the overloade **operator |** to process a queue

```
template < typename T, typename E, typename QueueFun >
auto operator | ( std::shared_ptr<
                   TExpectSynchroQueue< std::expected< T, E > > > in_queue_sp,
                  QueueFun && qf )
// can return any type (not only queue)
-> std::invoke_result_t< QueueFun,
        std::shared_ptr< TExpectSynchroQueue< std::expected< T, E > > > >
{
   assert( in_queue_sp );

   // run a function on the queue and return a new queue
   return std::invoke( std::forward< QueueFun >( qf ), in_queue_sp );
}
```

Parallel pipelines – the vector example again

```
void GeneralParallelTestSpace_Test()
{
   // Create the first queue with the paths to process
   auto in_q = create_queue_of_paths( ".\\..\\data2"sv, "txt"sv );
   in_q->push_sentinel();

   auto result = in_q
                 | load_vectors
                 | vec_normalize
                 | SynchroSentinel()
                 | comp_distance< VecOfVec, common_errors >
                 | find_max;

   if( result ) {
      auto [ x, y, v ] = * result; // Process valid data
      std::println( "success @ idx=({},{}; val={:.3f})\n", x, y, v );
   }
}
```

Finally, an example



Practically compatible with the serial version

Parallel pipelines

We have at least THREE things to solve

- THE OVERLOADE PIPE **`operator |`**: This time it has to deal with threads and queues and user defined operations.

- THE THREAD(S): For each node in the parallel pipeline there need to be a thread or threads that perform a task provided by the user.

- THE QUEUE(S): Each thread in a node has an access to two queues: INPUT and OUTPUT, and so on. The queues must be synchronized.

Parallel pipelines – the SYNCHRONIZED QUEUE

```
enum class EQueueErr { kElemNotAvailable, kSentinelElem };


// Elem denotes the type of objects that can be stored in this queue.
// These are additionally wrapped by the std::expected< Elem, EQueueErr >
template < typename Elem >
class TExpectSynchroQueue
{
public:

   using value_type = Elem;

   using ExpectedElem = std::expected< Elem, EQueueErr >;

public:

   // Push an object onto the queue.
   // Can block the calling thread for awhile if the queue is already locked.
   void push_elem( ExpectedElem && in_elem );
```

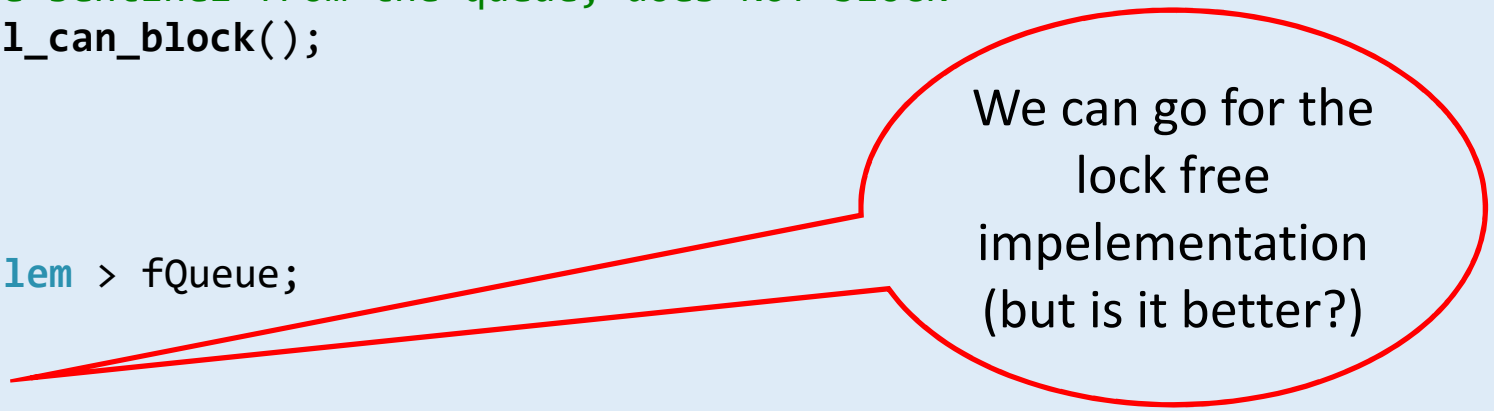
Observe that Elem itself can be std::expected

Parallel pipelines – the SYNCHRONIZED QUEUE

```
// Push a sentinel object onto the queue (i.e. set to EQueueErr::kSentinelElem)
// Can block the calling thread for awhile if the queue is already locked.
void push_sentinel( void );


// Returns ExpectedElem which can hold:
// - a valid object (check calling has_value() for true)
// - a signalling object, i.e. has_value() returns false,
//   then call error() and check for:
//-- EQueueErr::kElemNotAvailable - means the queue is empty
//-- EQueueErr::kSentinelElem - means the sentinel (last) element in the queue
// This function does NOT put a calling thread into a blocking state.
ExpectedElem pop_dont_block( void );


// Returns ExpectedElem which can hold:
// - a valid object (check calling has_value() for true)
// - a signalling object, i.e. has_value() returns false,
//   then call error() and check for:
//-- EQueueErr::kSentinelElem - means the sentinel (last) element in the queue
// This function CAN block the calling thread until the queue is NOT empty.
ExpectedElem pop_wait_can_block( void );
```

Parallel pipelines – the SYNCHRONIZED QUEUE

```
    // Checks if the top of the queue contains a sentinel object, and returns
    // - true if there is a sentinel
    // - false otherwise
    // Does NOT remove the sentinel from the queue, does NOT block
    bool check_for_sentinel_dont_block();

    // Checks if the top of the queue contains a sentinel object, and returns
    // - true if there is a sentinel
    // - false otherwise
    // Does NOT remove the sentinel from the queue, does NOT block
    void wait_for_sentinel_can_block();

private:

    std::queue< ExpectedElem > fQueue;

    std::mutex fMutex;

    std::condition_variable fCondVar;
```



Parallel pipelines – the SYNCHRONIZED QUEUE

A possible implementation (simplified)

```
// Push an object onto the queue.
// Can block the calling thread for awhile if the queue is already locked by other thread.
void push_elem( ExpectedElem && in_elem )
{
   assert( in_elem ); // allow only valid objects

   {
      std::unique_lock theLock( fMutex );
      fQueue.push( std::move( in_elem ) );
   }

   fCondVar.notify_one();// we call notify_one when the mutex is already
   // released - otherwise the notified thread
}  // can be woken up only to try to lock still locked mutex
```

Parallel pipelines – the SYNCHRONIZED QUEUE

```
// Push a sentinel object onto the queue (i.e. set to EQueueErr::kSentinelElem)
// Can block the calling thread for awhile if the queue is already locked by other thread.
void push_sentinel( void )
{
   {
      std::unique_lock theLock( fMutex );

      if( fQueue.empty() || fQueue.back() )
         fQueue.push( ExpectedElem { std::unexpected( EQueueErr::kSentinelElem ) } );
      else
         assert( ( fQueue.back().error() == EQueueErr::kSentinelElem ) &&
             "If not empty AND not a value, then error AND this error is kSentinelElem" );
   }

   fCondVar.notify_one(); // in some situations this can notify falsely
              // (e.g. if there already is a sentinel in the queue), however this is ok
}
```

If there is an element, then get access and dereference from std::expected

Parallel pipelines – the SYNCHRONIZED QUEUE

```
// Returns ExpectedElem which can hold:
// - a valid object (check calling has_value() for true)
// - a signalling object, i.e. has_value() returns false,
//   then call error() and check for:
//-- EQueueErr::kElemNotAvailable - means the queue is empty
//-- EQueueErr::kSentinelElem - means the sentinel (last) element in the queue
// This function does NOT put a calling thread into a blocking state.
ExpectedElem pop_dont_block( void )
{
   std::unique_lock theLock( fMutex );

   if( fQueue.empty() )
      return ExpectedElem { std::unexpected( EQueueErr::kElemNotAvailable ) };

   assert( not fQueue.empty() );

   // OK, we have something to pop and to return
   auto out_elem = fQueue.front();
   assert( out_elem.has_value() || out_elem.error() == EQueueErr::kSentinelElem );
   fQueue.pop();
   return out_elem;
}
```

Parallel pipelines – the SYNCHRONIZED QUEUE

```
// Returns ExpectedElem which can hold:
// - a valid object (check calling has_value() for true)
// - a signalling object, i.e. has_value() returns false,
//   then call error() and check for:
//-- EQueueErr::kSentinelElem - means the sentinel (last) element in the queue
// This function CAN block the calling thread until the queue is NOT empty.
ExpectedElem pop_wait_can_block( void )
{
   std::unique_lock theLock( fMutex );

   // I possessed the mutex. However, if I stay here we ALL WOULD BE BLOCKED,
   // since no other thread can push anything to this queue.
   // The solution is to give up my thread and realease the mutex
   // until the condition is true - in this case, the queue is not longer empty.
   fCondVar.wait( theLock, [ this ]() { return not fQueue.empty(); } );

   // OK, we have something to pop and to return
   auto out_elem = fQueue.front();
   assert( out_elem.has_value() || out_elem.error() == EQueueErr::kSentinelElem );
   fQueue.pop();
   return ExpectedElem( out_elem );
}
```

Parallel pipelines – thread management

Parallel pipelines – example

To measure serial vs. parallel I also needed to reach back to the story from 1986 ...

A simple ”refactorign” of Doug McIlroy’s Unix shell script of just 6 commands for the word count task.

```
tr –cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq –c |
sort –rn |
sed ${1}q
```

Parallel pipelines – example

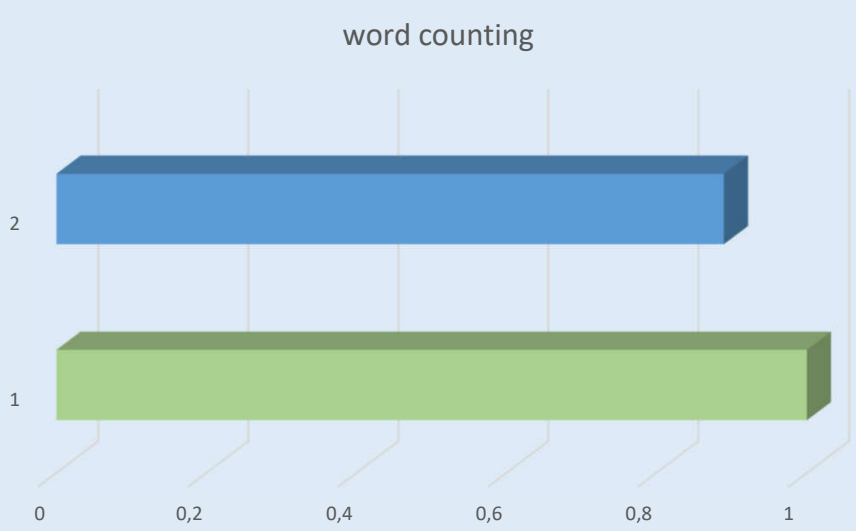
To measure serial vs. parallel I also needed to reach back to the story from 1986 ...

```
void test_parallel_pipe_word_count( std::filesystem::path data_path = ".\\..\\data3"sv,
                    std::filesystem::path ext = ".txt"sv, int num_topmost = 5 )
{
   auto in_q = create_queue_of_paths( data_path, ext );
   in_q->push_sentinel();

   // Here we create our CUSTOM PIPE
   in_q | [] ( path_com_exp pe )
              { if(pe){ std::println("path_name = {}",(*pe).string() ); } return pe; }
        | OpenFile_ToLower_FreqMap
        | Create_TupleVec
        | Sort_TupleVec
        | [ m = num_topmost ] ( tuple_vec_com_exp a ) { return Output_TopResults( a, m ); }
        | Print_Results
        | SynchroSentinel();
}
```

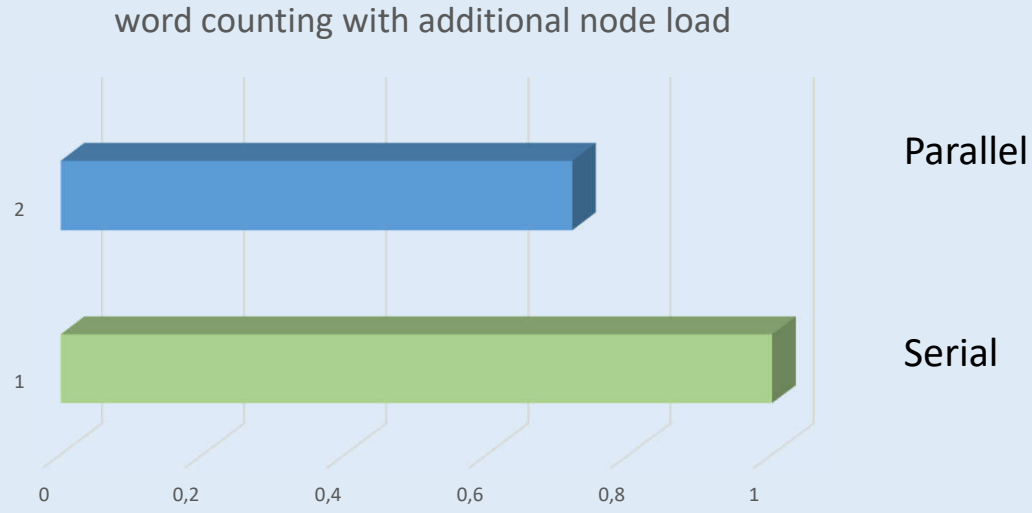
Parallel pipelines – example, time measurement

To measure serial vs. parallel I also needed to reach back to the story from 1986 ...

- The Project Gutenberg eBook of **Ulysses, by James Joyce** – a text file in UTF-8, ~100MBytes
- 100 copies to process
- Repeated 10 times
- Average time measured



Speed up ~11%



Speed up ~28%

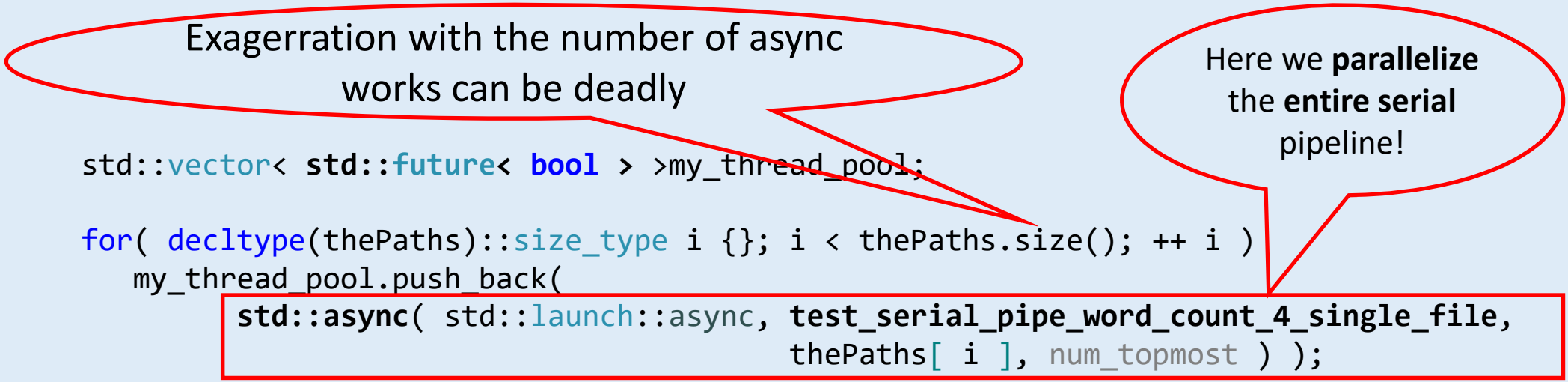
Parallel pipelines – example

However, this problem can be easily parallelized taking advantage of separate data files, e.g.

```
bool boosted_test_serial_pipe_word_count_4_directory( std::string_view dir_name,
std::string_view accept_ext, int num_topmost = 5 )
{
   if( ! fs::exists( dir_name ) || ! fs::is_directory( dir_name ) )
      return false;// exit if wrong path (not existing or not a dir)

   // Iterate through the directory
   std::vector< std::string > thePaths;

   for( const auto & file_obj : fs::directory_iterator( dir_name ) )
      if( fs::is_regular_file( file_obj ) )
         if( file_obj.path().extension().string().contains( accept_ext ) )
            thePaths.push_back( file_obj.path().string() );
```

Parallel pipelines – example

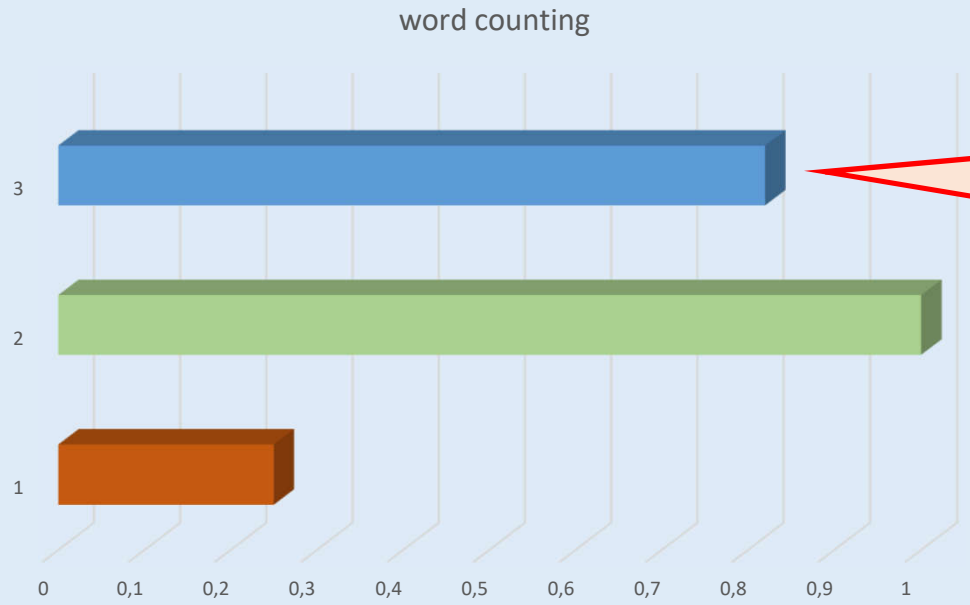
However, this problem can be easily parallelized taking advantage of separate data files, e.g.




```
    std::vector< std::future< bool > >my_thread_pool;

    for( decltype(thePaths)::size_type i {}; i < thePaths.size(); ++ i )
       my_thread_pool.push_back(
             std::async( std::launch::async, test_serial_pipe_word_count_4_single_file,
                                             thePaths[ i ], num_topmost ) );

    assert( thePaths.size() == my_thread_pool.size() );
    int sum {};
    for( decltype(my_thread_pool)::size_type i {}; i < my_thread_pool.size(); ++ i )
       sum += my_thread_pool[ i ].get() ? 1 : 0; // get() blocks until the async is done

    return sum == my_thread_pool.size() ? true : false;
}
```

Parallel pipelines – example

However, this problem can be easily parallelized taking advantage of separate data files, e.g.



But hey! Why is there such a big difference in performance? Here we compare: **function parallelism** ...

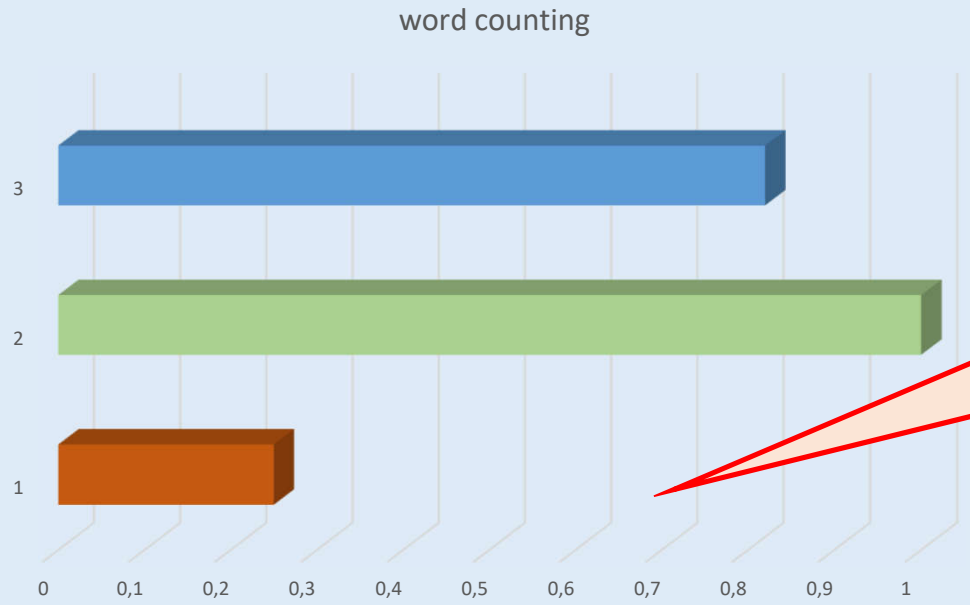
Parallel serial pipeline

In practice try out different variants and **measure!!**

Speed up 4 times!

Parallel pipelines – example

However, this problem can be easily parallelized taking advantage of separate data files, e.g.



But hey! Why is there such a big difference in performance? Here we compare: **function parallelism** vs. **data parallelization**

Parallel serial pipeline

In practice try out different variants and **measure!!**

Speed up 4 times!

CONCLUSIONS

A pipeline architecture in C++ - CONCLUSIONS

- A **pipeline architecture** can be very efficient in practical programming. This is a type of functional programming apart from the `std::ranges` library.
- With simple means we can define our own **`operator |`** and use it in a daily code.
- **`std::expected`** is a new and potentially a very useful object in STL.
- The **monadic** operations of `std::expected` constitute an alternative to the overloaded `operator |`

https://www.youtube.com/watch?v=6UYXSu_9dXI

A pipeline architecture in C++ - CONCLUSIONS

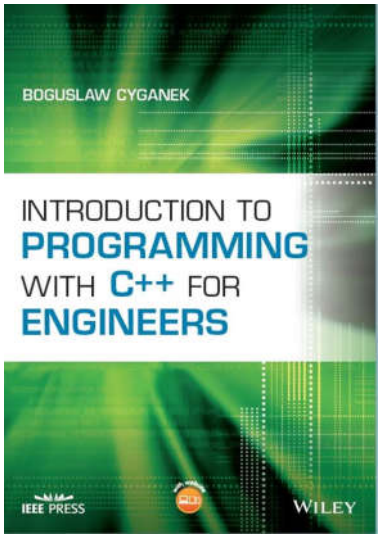
- To deal with a chain of error objects `std::variant` can be one of the options.
- The pipeline pattern facilitates the proper division of code into components-nodes, as well as their testing (especially if using ’pure’ functions).

A pipeline architecture in C++ - CONCLUSIONS

- The pipeline framework can be easily extended into the **parallel version**.
- We can benefit from the parallel version if we have a long pipeline with **many nodes** that are approximately **equally and heavily loaded** computationally.
- In many cases a serial (parallel?) pipeline but entirely run in a parallel fashion by itself.
- We have many variants of parallelism: **function parallelism** and **data parallelization**.
- In addition, each node can spawn its own threads/jobs.
- In practice – always **measure** performance of your solution. In some situations a parallel version can be *slower* than a serial one!

A pipeline architecture in C++ - CONCLUSIONS

https://home.agh.edu.pl/~cyganek/BookCpp.htm



Thank you!

https://github.com/BogCyg