Chepurni multimethods for contemporary C++

Abstract

Multiple dispatch, or multimethods, is a feature of some programming languages in which a function or method can be dynamically dispatched based on the run-time (dynamic) type or, in the more general case, some other attribute of more than one of its arguments[1]. The need for such a feature appears, for instance, in software architectures where numerous classes of objects interacts with each other in a way, specific for each pair. The C++ language does not provide such feature on the language level. Suggestions[2] to include multimethods in the core language are not (yet?) listed in the C++23 Overall Plan[11].

This study does not advocate a solution that would fit all use cases. Instead, it focuses on practical solutions for different use cases. The author encourages readers to experiment with the proposed solutions and tune or rework them for their particular needs.

1 Introduction

1.1 Existing Solutions

Since a need for multiple dispatching is as old as the world of programming, there are quite a few solutions for C++ have been created (see [3], [4], [5], [6], [7], [8], [9], [10]). However, none of them has a chepurni look (chepurni, Ukrainian *чепурні* – neat, clean, deft). Undoubtedly, chepurness is a subjective category. In author's opinion a chepurni solution is a simple to implement, easy to use, modern, non-intrusive and well structured. In other words, the complexity of a chepurni solutions adequate to the problem's complexity, its use does not create extra dependencies, does not complicate the project maintenance, does not require changes to the existing designs, and every element of the solution addresses a single concern[12] or bears a single responsibility[13].

1.2 Problem Overview

A dynamic, run-time dispatching requires knowledge about the class of the object. C++ facilitates Run-Time Type Information (RTTI), which is the choice number one for the project which already deploy it or are allowed to. However, the solution would not be chepurni if it does not offer an alternative for the projects, where enabling RTTI would make them unviable.

Traditionally, multiple dispatching in C++ is implemented via a virtual method (the first dispatch) that performs next level dispatches with `if` or `switch` statement, or with another virtual call to the other object. From the data modelling point of view, in many cases these methods do not look as native parts of the classes implementing them, but rather as a workaround, caused by a missing language feature. They would look more organically when implemented as functions. However, in this study we would not restrict from using the methods.

1.3 Data Models

In this study we will exercise on hierarchies of shapes and calculus expressions with simple inheritance:

```
namespace shapes {
struct Shape {
	virtual ~Shape() {}
};
struct Rect : Shape { };
struct Circle : Shape {};
struct Square : Rect { };
}
namespace calculus {
struct Expression {
	virtual ~Expression() {}
};
struct Constant : Expression {};
struct Integer : Constant {};
struct Float : Constant {};
}
```

2 Functions Multidispatcher

Let's start with the `calculus` model and assume that it defines operations `add`, `sub`, implemented as template functions:

```
template<class A, class B>
Expression* add(const A&, const B&);
template<class A, class B>
Expression* sub(const A&, const B&);
```

This assumption greatly simplifies the start, and we will make it more complicated as we advance.

As it was mentioned earlier, our multimethod should discover actual argument types, select a proper function from the list of available according to the discovered types, and invoke the selected function. Here we stated three concerns. Let's address them.

2.1 Function List

To define a list of function to dispatch we will use a variadic template with **auto** parameters, so it can be used as the following:

```
multidispatcher<
  add<Integer, Integer>,
  add<Float, Integer>,
  add<Integer, Float>,
  add<Float, Float>>::dispatch(a,b);
```

For this kind of usage, the template may look as the following:

```
template<auto Entry, auto ... Entries>
struct multidispatcher {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    if (entry<Entry>::matches(arguments...)) {
      return entry<Entry>::call(arguments...);
    }
    if constexpr (sizeof...(Entries)>0) {
      return multidispatcher<Entries...>::dispatch(arguments...);
    }
  }
};
```

In the **dispatch** method we delegated type identification and invocation to another template `entry`, let's write it:

```
template<auto Method>
struct entry;

template<class Return, class ... Parameter, Return (*Function)(Parameter...)>
struct entry<Function> {
  template<class ... Argument>
  static constexpr bool matches(const Argument& ... argument) noexcept {
    return (expected<Parameter>::matches(argument) and ...);
  }
  template<class ... Argument>
  static Return call(Argument& ... argument) noexcept(noexcept(Function)) {
    return (*Function)((Parameter)(argument)...);
  }
};
```

Another template `expected`, determines if the actual argument's type and whether it matches the expected type. With this design decisions, the complete implementation is the following:

```
template<class Parameter>
struct expected {
```

```
  template<class Argument>
  static constexpr bool matches(const Argument& argument) noexcept {
    return typeid(Parameter) == typeid(argument);
  }
};
template<auto Method>
struct entry;

template<class Return, class ... Parameter, Return (*Function)(Parameter...)>
struct entry<Function> {
  template<class ... Argument>
  static constexpr bool matches(const Argument& ... argument) noexcept {
    return (expected<Parameter>::matches(argument) and ...);
  }
  template<class ... Argument>
  static Return call(Argument& ... argument) noexcept(noexcept(Function)) {
    return (*Function)((Parameter)(argument)...);
  }
};

template<auto Entry, auto ... Entries>
struct multidispatcher {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    if (entry<Entry>::matches(arguments...)) {
      return entry<Entry>::call(arguments...);
    }
    if constexpr (sizeof...(Entries)>0) {
      return multidispatcher<Entries...>::dispatch(arguments...);
    }
  }
};
```

It is available for experiments on this link: godbolt.org/z/oz585K

This implementation is very simple, although, comparing to the open methods, it has certain constraints:

1) the length of the function list is limited by the compiler's recursion depth
2) the return type covariance is not supported
3) the parameter covariance is not supported
4) a first good candidate is selected instead of the best one
5) virtual parameters not distinguished form regular, all treated as virtual

The second constraint restricts the users from using a function that returns a covariant result, for example one like this `Float* sub(const Float&, const Float&)`. Also, this implementation does not do anything special about multiple inheritance. Such cases are handled the same way as simple inheritance – the first matching function is selected.

Third, fourth and fifth constraints limits the use cases. We will address them in the chapters below. For now, we will focus on the first and second constraints.

3 Multimethods

3.1 Return Type Covariance

To support the return type covariance, we need to define the most generic return type for all used functions. To make it simple, we put this responsibility on the user and add this type as a parameter to our new template:

```
template<typename ReturnType, auto ... Entries>
struct multimethod;
```

As we are given the return type, we may use it for replacing the recursion with a folding expression:

```
template<typename ReturnType, auto ... Entries>
```

```
struct multimethod {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    ReturnType value;
    if (((entry<Entries>::matches(arguments...)
      and ((value = entry<Entries>::call(arguments...)),true)) or ...))
      return value;
    else
      throw std::logic_error("Missing dispatch entry");
  }
};
```

This modified example is available on this link: godbolt.org/z/rj5vvY

We have to admit, that this approach requires the return type to be default constructible and to support move semantic. For our examples it makes no difference. For the other use cases the users have to account that.

A curious reader perhaps noticed that the last example on godbolt.com is not using overloaded functions for **sub**:

```
Float* subf(const Float&, const Float&);
Integer* subi(const Integer&, const Integer&);
```

This is a workaround for a C++ feature for the overloaded functions. To get an address of an overloaded function, one has to specify its full type: `(Float* (*)(const Float&, const Float&))&sub`. This inconvenient syntax could be a bit sugared with a template `resolve`:

```
resolve<Float*, const Float&, const Float&>{}(&sub).
```

3.2 RTTI Substitute

To work around the dependency on RTTI contributed with **typeid**, the model has to implement a custom type identification and introspection facilities (CTII). For instance, it could be a manually or automatically generated ID, assigned to every class. We may follow a simple autogenerating approach with a hash of `__PRETTY_FUNCTION__`:

```
template<class Class>
constexpr auto class_hash() noexcept {
  return hash(std::string_view(__PRETTY_FUNCTION__));
}
```

Unfortunately, `std::hash` is not yet **constexpr**, thus we have to write out own hash function, (for example, one like given in [14]). Now we can easily assign unique IDs to our classes:

```
  static constexpr auto classid = class_hash<Rect>();
```

To access *classid* we define a template function **classinfo**:

```
using class_info = size_t;
template<class Class>
class_info classinfo() noexcept {
  return Class::classid;
}
```

For the dynamic type introspection, we define a virtual method:

```
  //Shape
  virtual bool instance_of(const class_info& expected) const noexcept {
    return classinfo<decltype(*this)>() == expected;
  }

  //Rect, Circle
  bool instance_of(const class_info& expected) const noexcept override {
    return classinfo<decltype(*this)>() == expected or Shape::instance_of(expected);
  }
```

To avoid a dependency from the custom class `class_info`, we hide it behind a façade:

```
  //Shape
  template<class Expected>
  bool instanceof() const noexcept {
    return instance_of(classinfo<Expected>());
  }
```

Note, these methods are not overloaded. This will simplify our feature detecting template **has_instanceof**.

While we were using RTTI, we could ignore differences between virtual and regular parameters – **typeid** works for all types, and optimizer removes extraneous type checking for static types from the generated binary code. With CTII, however, we need to decide how to distinguish virtual parameters from non-virtual and how to compare the types for the latter. We may set the following rule: lvalue reference parameters to polymorphic types are virtual, the others are not:

```
template<class Class>
struct is_virtual_parameter {
  static constexpr bool value =
       std::is_polymorphic_v<std::remove_reference_t<Class>> and
       std::is_reference_v<Class>;
};
```

For the type check of not-virtual parameters we may use, for instance, `is_same`, or `is_assignable`. With this design of CTII, the adjusted template `expected` may look as the following:

```
template<class Parameter>
struct expected {
  using parameter_type = std::remove_reference_t<std::remove_cv_t<Parameter>>;
  template<class Argument>
  static constexpr bool matches(Argument&& argument) noexcept {
    if constexpr(has_instanceof<Argument>(0)) {
      return argument.template instanceof<parameter_type>();
    } else {
#if __cpp_rtti >= 199711
      return typeid(Parameter) == typeid(argument);
#else
      static_assert(not is_virtual_parameter_v<Parameter>, "No class info available");
      return is_assignable_parameter_v<Parameter, Argument>;
#endif
    }
  }
};
```

3.3 Supporting the Methods

So far, our `multimethod` template supports functions only. Let's extend it to accept methods as well. First what we need is to separate an object from the remaining arguments. We may, for example, define a new method in `multimethod` that accepts the objects as its first argument:

```
// multimethod
template<class Target, class ... Arguments>
static auto call(Target& target,Arguments& ... arguments) {
  ReturnType value;
  if (((entry<Entries>::matches(target, arguments...)
    and ((value = entry<Entries>::call(target, arguments...)),true)) or ...))
    return value;
  else
    throw std::logic_error("Dispatcher failure");
}
```

And specialize template `entry` for methods:

```
template<class Target, class Return, class ... Parameter, Return
(Target::*Method)(Parameter...)>
```

```
struct entry<Method> {
  template<class Object, class ... Argument>
  static constexpr bool matches(Object& obj, Argument& ... argument) noexcept {
    return expected<Target>::matches(obj)
       and (expected<Parameter>::matches(argument) and ...);
  }
  template<class Object, class ... Argument>
  static Return call(Object& target, Argument& ... argument) {
    return ((Target&)(target).*Method)((Parameter&)(argument)...);
  }
};
```

Dealing with methods are somewhat more complicated than with functions – their signature may have specifier `const`. Thus, we will need two specializations of `entry`, and two methods `call`. With such implementation our `multimethod` accepts functions intermixed with methods. The sources for this design are available for experiments on this link: godbolt.org/z/ehEnnc

3.4 Parameter Covariance

The CTII design with `instance_of` calling the base class also enabled parameter covariance. However, to get it working properly, the list should be ordered in a specific way: functions with more specific parameters should precede ones with more generic. It does not seem feasible to sort the list at compile time. Instead, one could add an order check which ensures that there are no functions below the current, accepting classes derived from the current ones. Computational complexity of such checking is estimated as $O(kn^2)$, where k is number of parameters, and n – number of functions in the list. It worth to note that this checking may significantly slowdown the compilation.

3.5 Multiple Inheritance

Our CTII may also help with multiple inheritance – a class, inheriting multiple base classes should simply call `instance_of` of all its base classes. However, there might be cases, when the list ordering would not be sufficient for selecting the best match for certain combinations of parameter types.

3.6 Possible Improvements

Maintaining a long list of functions may become too difficult for long function lists. To address this issue, one may group the list by the first parameter, like in the following example:

```
groupdispatcher<
  group<Expression*,
    add<Integer, Float>,
    add<Integer, Integer>>,
  group<Expression*,
    add<Float, Integer>,
    add<Float, Float>>>::dispatch(a,b);
```

Each group then can be maintained in a separate header file. Also, one can use virtual methods for the first dispatch and `multimethod` - for the next dispatches.

3.7 Performance

This implementation of `multimethod` sequentially examines the functions till it finds a suitable one. Experiments shows 700 instructions in average for a list of 40 functions (please refer to perf.cpp on the github or to godbolt.org/z/1caGTs). With this implementation, a test run on x64 completes 10,000,000 dispatches in 6.4 sec. A table dispatching, expectingly, would show a better performance.

4 Table Dispatching

4.1 Assumptions

For a table dispatching, which is a matrix dispatch for two parameters, we need to fulfil some preconditions and solve some challenges:

1. An effective table dispatching requires sequential class IDs, preferably with no gaps.
   - A manual ID assignment is error prone and, for some projects, may be too difficult in maintaining.
2. A type-safe matrix may contain only functions of the same type, e.g. with identical signature.
   - A manual matrix filling is error prone, difficult even for small set of classes and practically impossible for large hierarchies.

For now, we just assume that sequential identification is made by the user, and address this concern in a chapter below. Also, as in previous designs, we assume that the functions are defined with the parameter types they actually operate on.

4.2 Matrix Design

We have outlined the challenges, let's find a design to solve them. As in earlier chapters, we start with a hierarchy of N classes that has K dispatchable functions defined on it. These functions we may list in our variadic template:

```
matrixdispatcher<
  add<Integer, Integer>,
  add<Float, Integer>,
  add<Integer, Float>,
  add<Float, Float>>::dispatch(a,b);
```

As we assumed, class identification is made by the user:

```
  static constexpr auto classid = 1;
  virtual size_t classID() const { return classid; }
```

If we require the same signature for all K functions, our template would not be able to distinguish them and pick the best one. Also, requiring the same signature we would transfer responsibility of down casting on the user. This seems to be too intrusive. So, we instead assume that the functions have different signatures with type of parameters they actually operate. To close the gap between different signatures on input and identical in the matrix we need some intermediate functions. With help of templates, we may delegate generating needed quantity of functions to the compiler and fill the matrix with pointers to them. They all should have the same signature, with the most common parameters of all dispatchable functions. Deriving such signature does not seem feasible, so we let the user to specify it as an input parameter `FunctionType` of our template. Also, we are not able to determine the actual number of classes, that we may get in our `dispatch` method. So, we will require this information to be a part of the input as well.

With this design, our matrix may look like in this example:

```
FunctionType matrix[sizeA][sizeB];
```

However, filling this matrix in a `constexpr` manner is not handy, so we make it a bit more complex:

```
std::array<std::array<FunctionType, sizeA>, sizeB> matrix;
```

Now, let's find a way to fill it with function pointer. We need a `constexpr` filling to ensure that it will happen during compilation. C++ provides some means for filling `constexpr` arrays, in this design we use `index_sequence`. For instance, we may fill one row in our matrix with this line of code:

```
std::array<FunctionType,N> { &Template<I> ... };
```

Where `Template` - is a template function with class ID as a parameter, and `I` - an index sequence. Elaborating this design to some degree of completeness we get:

```
template<template<size_t> class Template, size_t N>
class jumpvector : public std::array<typename Template<0>::value_type,N> {
public:
  using value_type = typename Template<0>::value_type;
```

```
  constexpr jumpvector() : jumpvector<Template,N>(std::make_index_sequence<N>()) {}
private:
  template<size_t ... I>
  constexpr jumpvector(std::index_sequence<I...>)
   : std::array<value_type,N> { &Template<I>::function ... } {}
};
```

Please note, in the code above we had to shift from a template function `Template`, to a template class `Template` with a static method `function`. This is because a template function would require signature, defined at this time. While with a static method we may defer the signature definition to a late stage.

Applying this design to all rows, we may fill entire matrix:

```
template<template<size_t, size_t> class Template, size_t N, size_t M>
class jumpmatrix : public std::array<std::array<typename Template<0,0>::value_type, M>,
N> {
public:
  using value_type = std::array<typename Template<0,0>::value_type, M>;
  constexpr jumpmatrix() : jumpmatrix<Template, N, M>(std::make_index_sequence<N>()) {}
private:
  template<size_t ... I>
  constexpr jumpmatrix(std::index_sequence<I...>)
    : std::array<value_type, N> {
        jumpvector<reduce<Template,I>::template type,M>{} ... } {}
};
```

This implementation sets up the requirements for the template class `Template`:

```
  template<size_t A, size_t B>
  struct func {
    static result_type function(parama_type& a, paramb_type& b);
  };
```

Where `result_type`, `parama_type`, and `paramb_type` are parts, deduced from the `FunctionType` signature.

4.3 Scoring and Selecting

For each specialization of our template `func`, we know exact class of each parameter - they are set by `A` and `B`. Thus, we can select the best candidate yet at compilation. To make this selection simple, we split it on two steps - scoring all functions with some criteria and selecting one with the highest score. The criteria should operate on actual and expected types. In C++17 we have standard templates `is_same` and `is_base_of`. For example, a class scoring template may be implemented as this:

```
template<class Parameter, class Argument>
constexpr ssize_t compute_score() noexcept {
  if( std::is_same_v< Argument, Parameter> ) return 2;
  if( std::is_base_of_v<Parameter, Argument> ) return 1;
  return 0;
}
```

The function score then can be computed as a product of its parameter scores. This would work for simple hierarchies without multiple inheritance. More complex hierarchies may require a more advanced scoring design, based on a custom genesis inspection.

Now, we fill an array with the scores for every function from the list:

```
template<auto Entry>
static constexpr function_score make_score(size_t index) noexcept {
  using entry = function_traits<decltype(Entry)>;
  using paramA = typename entry::template nth_arg_type<0>;
  using paramB = typename entry::template nth_arg_type<1>;
  return function_score {
    index, compute_score<paraьA,argA>() * compute_score<paramB,argB>() };
}
```

And select an element with the highest score:

```
template<class ClassA, class ClassB, auto ... Entries>
constexpr ssize_t find_best_match() noexcept {
  constexpr auto sc = function_scores<ClassA, ClassB, Entries...>{};
  return sc.highest();
}
```

In a snippet of code above, `function_traits` - is an auxiliary template for determining the function's characteristics:

```
template<typename Function>
struct function_traits;

template<class Return, class ... Parameter>
struct function_traits<Return (*)(Parameter...)> {
  using result_type = Return;
  static constexpr size_t parameter_count = sizeof...(Parameter);
  template<size_t N>
  using nth_arg_type = std::tuple_element_t<N, std::tuple<Parameter...>>;
};
```

Once we have a `constexpr` function index, we can get a function pointer from the list of input functions. To pass parameters to that function, we need a `down cast`, but since we already proved the proper relationship between the types, we may safely use `static cast`. Thereby, our function template becomes:

```
template<size_t A, size_t B>
struct func {
  static result_type function(parama_type& a, paramb_type& b) {
    constexpr auto best = find_best_match<type<A>, type<B>, Entries...>();
    if constexpr(best >= 0) {
      constexpr auto f = get_entry<best, Entries...>();
      return (*f)( static_cast<type<A>>(a), static_cast<type<B>>(b));
    } else {
       LOGIC_ERROR("Dispatcher failure");
      }
    }
  }
};
```

Here we used another template `type<B>`, returning a type by its ID, e.g., implementing the reverse class-ID mapping. Since we have custom class IDs, this mapping has to be custom as well. The mapping and supply of the maximal class ID are related responsibilities, so we may delegate them to a single concept, further named domain. We have two input parameters and they may belong to different domains, thus we need two custom domains, which we will get as the parameters of our `matrixdispatch`. When both parameters share the same domain, the same domain class will be used in place of both domain parameters. All these design decisions can be implemented with the following code:

```
template<typename FunctionType, class DomainA, class DomainB, auto ... Entries>
class matrixdispatch {
public:
  using entry = function_traits<FunctionType>;
  using result_type = typename entry::result_type;
  using parama_type = typename entry::template nth_arg_type<0>;
  using paramb_type = typename entry::template nth_arg_type<1>;
  constexpr matrixdispatch() noexcept {}
private:
  template<size_t A, size_t B>
  struct func {
    static result_type function(parama_type& a, paramb_type& b) {
      //See code snippet above
    }
  };
  static constexpr jumpmatrix<func, DomainA::size, DomainB::size> matrix {};
```

```
public:
  static result_type dispatch(parama_type arg1, paramb_type arg2) {
      return matrix[arg1.classID()][arg2.classID()](arg1, arg2);
  }
};
```

With some more work, we may improve this template to support methods, functions and combinations of them. Live example for this template is available on the following link: godbolt.org/z/Y89ThK

4.4 Performance

The matrix dispatch shows much better performance - 40 instructions (vs 700 in linear) for a list of 40 functions. However, it is also much more resource consuming for the compiler: $O(pnk^2)$ where p is a number of parameters, n - number of functions, and k – number of classes. Compilation of an example with 25 classes and 40 functions takes 20 sec more, when the `matrixdispatch` template is actually used. Also, the 15-times decrease of the instruction count per dispatch does not lead to the same decrease of the dispatch time. An experiment on x64 for 10,000,000 dispatches complete in 1.2 sec vs 6.4 sec for the linear dispatch.

5 Automatic Identifier Assignment

To make the class identifier assignment simple, we may craft a domain template:

```
template<class ... Classes>
struct domain {
  static constexpr auto size = sizeof...(Classes);
  template<size_t ID>
  using type = std::tuple_element_t<ID, std::tuple<Classes...>>;
  template<class Class>
  static constexpr size_t id_of() noexcept {
    constexpr auto value = index_of<Class, Classes...>();
    return value;
  }
};
```

This template accepts a list of classes, and implements forward (`id_of<MyClass>()`) and reverse (`type<ID>`) mapping. This template does not require the listed classes to be completely defined, just forward declarations of them is sufficient.

To reduce the compilation complexity, we may exclude abstract classes from the matrix – instances of such classes cannot be created, and thus, they will never participate in the dispatch. However, to make this exclusion efficient, we need to assign abstract classes IDs from a lower diapason [0..M]. To address this challenge in a simple way we may assume that all abstract classes listed prior the concrete classes and provide a validation facility to check, whether this assumption is true.

6 Final Notices

1. gcc compiler for some examples, published on godbolt.org, generates code with static dispatching instead of dynamic. To make it truly dynamic, the **`test`** functions should be compiled in different compilatons units, as in examples on github.
2. Examples from this study are also available on gist.github.com/hutorny:
   calculus multidispatcher
   calculus multifunction
   shapes multimethod
   matrix dispatch
3. Ready to use templates are available as include-only library:
   github.com/hutorny/multimethods

References

[1] Multiple dispatch, Wikipedia.
https://en.wikipedia.org/wiki/Multiple_dispatch

[2] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup,
Open Multi-Methods for C++,
https://www.stroustrup.com/multimethods.pdf

[3] Elsevier B.V.,
EVL: A framework for multi-methods in C++,
https://www.sciencedirect.com/science/article/pii/S0167642314003360

[4] D. Shopyrin,
MultiMethods in C++: Finding a complete solution,
https://www.codeproject.com/Articles/7360/MultiMethods-in-C-Finding-a-complete-solution

[5] Loki Library, Multiple dispatcher,
https://sourceforge.net/projects/loki-lib/ Reference/MutiMethod.h

[6] L. Bettini,
Doublecpp - double dispatch in C++,
http://doublecpp.sourceforge.net/

[7] J. Smith,
Draft proposal for adding Multimethods to C++,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html

[8] stfairy,
Multiple Dispatch and Double Dispatch,
https://www.codeproject.com/Articles/242749/Multiple-Dispatch-and-Double-Dispatch

[9] J. Smith,
Multimethods,
http://www.op59.net/accu-2003-multimethods.html

[10] D. Shopyrin,
Multimethods in C++ Using Recursive Deferred Dispatching
https://www.computer.org/csdl/magazine/so/2006/03/s3062/13rRUxBa5ve

[11] V. Voutilainen
To boldly suggest an overall plan for C++23
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html

[12] Wikipedia, Separation of Concerns
https://en.wikipedia.org/wiki/Separation_of_concerns

[13] Wikipedia, Single-responsibility principle
https://en.wikipedia.org/wiki/Single-responsibility_principle

[14] E. Hutorny,
FNV-1b hash function with extra rotation
https://gist.github.com/hutorny/249c2c67255f842fae08e542f00131b5