INTERFACE DESIGN IN C++ - PART II

- **How to Deal with Values That Are Both Input and Output**
- **Named Arguments in C++**
- **Generalizing Unordered Named Arguments**
- **Don’t Make Your Interfaces *Deceptively* Simple**

Fluent {C++}

How to Deal with Values That Are Both Input and Output

Passing inputs and getting outputs from a function is pretty straightforward and uncontroversial:

- inputs get in as function arguments by const reference (or by value for primitive types),
- outputs get out via the return type.

```
Output function(Input1 const& input1, int input2);
```

Now this is all well, until **input-output values** get in the picture. An input-output value is a value that the function modifies.

One use case for that is with a value that goes through several functions that build it incrementally:

```
void addThis(Value& x);
void addThat(Value& x);
void topItOffWithACherry(Value& x);
```

This construction is not packed into a constructor of the type `Value` because those functions can be called or not, in order to build the `Value` with various combinations of features.

In the above snippet, input-output values are represented as non-const references. And this is the guideline provided by the C++ Core Guidelines: F.17: For “in-out” parameters, pass by reference to non-const.

But is it ok? Not everyone thinks so. Here are the views of several conference speakers on the question, with their alternative solutions.

Thanks a lot to Kate Gregory for reviewing of this article.

Kate Gregory: Find an abstraction

When I attended Kate Gregory's talk at ACCU 2018 (which is a very good one btw), I was surprised by one of her guidelines. She recommends to avoid output parameters, which I totally understand (indeed outputs should come out via the return type). But she goes further than that, by suggesting that we should *also* avoid input-output parameters.

Parameters are fundamentally inputs of a function, they look like so at call sites, and it can be confusing to use a function parameter for output even if it's also an input. It makes sense, but then how do we pass a parameter for a function to modify? There are valid use cases for this, isn't there?

Along with the guideline of avoiding them, Kate gives a way out of input-output parameters.

In some cases, you can remove the input-output parameters altogether from a function, by transforming the function into a **class method**.

In our example, we could refactor the code so that it gets called this way:

```
x.addThis();
x.addThat();
x.topItOffWithACherry();
```

The implementation of the method goes and changes the value of the class data members, and we no longer have to deal with an (input-)output parameter.

What's interesting is that when you read it, this code suggests that it modifies `x`. On top of naming (that was already there) those methods now take `void` and return `void`. So apart from modifying the object they operate on, there is not much else they can do (apart from a global side-effect).

What if we can't change the interface?

We don't always have the possibility to modify the interface of `Value` though. What if it is `int` for example, or `std::vector<std::string>`, a class from a third-party library or just some code that we don't have ownership on?

In this case, Kate suggests to **look for an abstraction**. Let's take the example of `std::vector` to illustrate.

Say that we have a `std::vector<Item>`, to which we'd like to add certain elements:

```
void addComplimentaryItem(std::vector<Item>& items);
```

We can't modify the interface of `std::vector` to add a method to add a complimentary item for a customer (and that's probably a good thing that we can't!).

One way that sometimes work is to take a step back and look at the context where this `std::vector` is used. Maybe there is an abstraction it belongs to, for example an Order here.

When we find that abstraction, we can wrap our `std::vector` in an Order class, that may also contain other things:

```
class Order
{
public:
    addComplimentaryItem();
    // other things to do with an order...

private:
    int orderId_;
    std::vector<Item> items_;
};
```

And the input-output parameter is gone.

Don't force an abstraction

This sort of refactoring is an improvement to the code, that goes beyond the removal of input-output parameters. Indeed, such an abstraction allows to tidy up some bits of code and to hide them behind a meaningful interface.

This is why we should do this sort of refactoring only when it leads to meaningful interfaces. It makes no sense to create a `VectorWrapper` interface just for the sake of transforming the input-output parameters into class members.

Also, in the cases of function taking several input-output parameters, it can be harder to move the code towards one of them to create an abstraction.

Mathieu Ropert: carrying along the object's guts

On his very well written blog, Mathieu demonstrates an experimental technique to get rid of input-output parameters: breaking them into an input parameter and an output parameter, and use move semantics:

```
Value x;
x = addThis(std::move(x));
x = addThat(std::move(x));
x = topItOffWithACherry(std::move(x));
```

And the function would take the parameters by value:

```
Value addThis(Value x);
Value addThat(Value x);
Value topIfOffWithACherry(Value x);
```

An interesting advantage of using move semantics here is that it expresses that the input parameter plunges into the function and comes out of it via its return type.

And then there is std::swap

As a final note, consider the standard library function `std::swap`, that takes no less than two input-output parameters:

```
template< typename T >
void swap(T& a, T& b);
```

I don't see a reasonable `Swapper` abstraction that would get rid of the input-output parameters of `std::swap`. And moving in and out the parameters to swap would be very confusing also. So none of the above techniques seems to work with `std::swap`.

But on the other hand, `std::swap` is… OK the way it is! Indeed, when you look at it from a call site:

```
std::swap(x, y);
```

it is unambiguous that it swaps together the contents of `x` and `y`.

But why is it OK? Is it because `std::swap` does *just* that? Or is it because we're used to it? Does everyone in the world like swap the way it is? Are there other cases where input-output parameters make the interface clear, or is `std::swap` a singularity?

Named Arguments in C++

Ah, named arguments!

If the term “feature envy” wasn’t already used to designate a refactoring, we would have employed it to talk about named arguments: it’s a feature that languages that don’t have it envy to the languages that do.

Named arguments consist in specifying at call site the parameter names of the arguments passed. For instance, consider this function:

```
void displayCoolName(std::string const& firstName, std::string
const& lastName)
{
    std::cout << "My name is " << lastName << ", " << firstName << '
' << lastName << '.';
}
```

The call site of that function looks like this:

```
displayCoolName("James", "Bond");
```

(hey, want to try it out with your own name to see how cool your name sounds?)

With named arguments, the call site would rather look like that:

```
displayCoolName(firstName = "James", lastName = "Bond");
```

It has the advantage of being more explicit so that you don’t mix up the order of the parameters. Also, a reader of the call site doesn’t have to go check the function’s prototype to understand the meaning of the function’s parameters.

Some languages have this. Objective-C has it, Python has something not far, hell even C99 has something resembling it for initiating structures.

And us in C++? We stand here, envying the feature.

Envy no more, here is a technique to implement named arguments in C++.

I will present this in 3 steps:

- step 1: strong types to differentiate parameters,
- step 2: a trick to get the right syntax,
- step 3: going further: passing the parameters in any order.

I want to thank Reddit user /u/matthieum from which I got largely inspired when he commented on the Reddit thread of Strong types for strong interfaces.

Step 1: Strong types to differentiate parameters

If you are a regular reader of Fluent C++, you probably have already encountered **strong types**.

Strong types consist in replacing a type with another one that adds meaning through its name. In our above example we could create two strong types, `FirstName` and `LastName` that both wrap a `std::string` to pin a specific meaning (like representing a first name, or a last name) over it.

For this we will use the `NamedType` library, of which you can get a overview on its GitHub page or in Strong types for strong interfaces, if you're not familiar with it.

```
using FirstName = NamedType<std::string, struct FirstNameTag>;
using LastName = NamedType<std::string, struct LastNameTag>;
```

These are two different types. They both wrap a `std::string` that they expose through their `.get()` method.

Let's replace the naked `std::string`s in our `displayCoolName` function with those strong types:

```
void displayCoolName(FirstName const& firstName, LastName const&
lastName)
{
```

```
    std::cout << "My name is " << lastName.get() << ", " <<
firstName.get() << ' ' << lastName.get() << '.';
}
```

Now here is what a call site looks like:

```
displayCoolName(FirstName("James"), LastName("Bond"));
```

That can play the role of named arguments, and it would already be reasonable to stop here.

But let's wrap around the C++ syntax to get to those oh-so-enviable named arguments, with the equal sign and all.

Step 2: A trick to get the right syntax

We would like to be able to write a call site like this one:

```
displayCoolName(firstName = "James", lastName = "Bond");
```

Let's reason about this syntax: we need the first argument to be of type `FirstName`. So we need to define an object `firstName` that has an `operator=` that takes a `std::string` (or something convertible to it) and that returns a `FirstName`.

Let's implement the type of this object `firstName`. We call this type `argument`. Since `argument` must know `FirstName`, which is a template class, I think the most convenient is to implement `argument` as a nested class inside the class `FirstName`.

`FirstName` is an alias of `NamedType`, so let's add `argument` inside `NamedType`:

```
template< /* the template args of NamedType */ >
class NamedType
{
public:
    // ...

    struct argument
    {
        template<typename UnderlyingType>
        NamedType operator=(UnderlyingType&& value) const
        {
```

```
            return NamedType(std::forward<UnderlyingType>(value));
        }
    };
};
```

We can now create the `firstName` and `lastName` helpers to accompany our function:

```
static const FirstName::argument firstName;
static const LastName::argument lastName;

void displayCoolName(FirstName const& theFirstName, LastName const&
theLastName)
{
    std::cout << "My name is " << theLastName.get() << ", " <<
theFirstName.get() << ' ' << theLastName.get() << '.';
}
```

And now the call site of `displayCoolName` looks at last like this:

```
displayCoolName(firstName = "James", lastName = "Bond");
```

Yay, named arguments!

The NamedType library now has this feature available.

As a side note, since the `firstName` and `lastName` helpers are not supposed to be passed to a function, let's delete the default-generated move and copy methods:

```
struct argument
{
    template<typename UnderlyingType>
    NamedType operator=(UnderlyingType&& value) const
    {
        return NamedType(std::forward<UnderlyingType>(value));
    }
    argument() = default;
    argument(argument const&) = delete;
    argument(argument&&) = delete;
    argument& operator=(argument const&) = delete;
    argument& operator=(argument&&) = delete;
};
```

Step 3: Going further: passing the parameters in any order

Since we indicate which argument corresponds to what parameter, do we really need a fixed order of arguments?

Indeed, it would be nice if any given call site had the choice to write either this:

```
displayCoolName(firstName = "James", lastName = "Bond");
```

or that:

```
displayCoolName(lastName = "Bond", firstName = "James");
```

and that it would have the same effect.

We're going to see a way to implement this. However, I don't think it is production-ready because of some readability drawbacks that we will see.

So from his point on we're threading into the exploratory, and of course your feedback will be welcome.

Since we don't know the types of the first and second parameter (either one could be `FirstName` or `LastName`), we are going to turn our function into a template function:

```
template<typename Arg0, typename Arg1>
void displayCoolName(Arg0&& arg0, Arg1&& arg1)
{
    ...
```

Now we need to retrieve a `FirstName` and a `LastName` from those arguments.

Picking an object of a certain type amongst several objects of different types sounds familiar: we can use **`std::get`** on a `std::tuple` like when we used strong types to return multiple values.

But we don't have a `std::tuple`, we only have function arguments. Fortunately, there is nothing easier than packing function arguments into a `std::tuple`, thanks to the `std::make_tuple` function. The resulting code to pick a type looks like this:

```
template<typename TypeToPick, typename... Types>
TypeToPick pick(Types&&... args)
```

```
{
    return
std::get<TypeToPick>(std::make_tuple(std::forward<Types>(args)...));
}
```

Let's use this to retrieve our `FirstName` and `LastName` from the arguments:

```
template<typename Arg0, typename Arg1>
void displayCoolName(Arg0&& arg0, Arg1&& arg1)
{
    auto theFirstName = pick<FirstName>(arg0, arg1);
    auto theLastName = pick<LastName>(arg0, arg1);
    std::cout << "My name is " << theLastName.get() << ", " <<
theFirstName.get() << ' ' << theLastName.get() << '.' << '\n';
}
```

Now we can call either:

```
displayCoolName(firstName = "James", lastName = "Bond");
```

or:

```
displayCoolName(lastName = "Bond", firstName = "James");
```

And in both cases we get:

```
My name is Bond, James Bond.
```

One of the drawbacks that I see with this latest technique is that it converts our function into a template. So it needs to go to a header file (unless we do explicit instantiation of all the permutations of the arguments).

To mitigate this, we could extract a thin layer that picks the arguments and forwards them the the function as it was before:

```
// .hpp file

void displayCoolNameImpl(FirstName const& theFirstName, LastName
const& theLastName);

template<typename Arg0, typename Arg1>
void displayCoolName(Arg0&& arg0, Arg1&& arg1)
{
    displayCoolNameImpl(pick<FirstName>(arg0, arg1),
pick<LastName>(arg0, arg1));
}


// .cpp file

void displayCoolNameImpl(FirstName const& theFirstName, LastName
const& theLastName)
{
    std::cout << "My name is " << theLastName.get() << ", " <<
theFirstName.get() << ' ' << theLastName.get() << '.' << '\n';
}
```

Another drawback is that the names of the parameters in the prototype lose all their meaning ("Argo"...).

Generalizing Unordered Named Arguments

This is a guest post is written by Till Heinzel. Till is a physicist-turned-software developer at Luxion Aps in Denmark, who is very interested in expressive C++ and the growth of the language in a more expressive direction. Till can be found online on LinkedIn.

First off, I would like to thank Jonathan for creating FluentCpp and allowing me to contribute with this post.

One of the more subtle effects of using strong types for function-arguments is the fact that each argument is guaranteed to be of unique type. We can exploit that fact to create interfaces for functions that take the arguments in any order, and use some metaprogramming to put the arguments in their correct place, as Jonathan explores in his post on Named Arguments, upon which this post builds.

I was struck by Jonathan's post as I tried to implement something similar a few years back when I was implementing a physics-library that contained some optimization algorithms. The algorithms had a lot of places where we wanted users to be able to adjust the behaviour (e.g. outputs from the algorithm, specific linesearches, stopping conditions, etc.), preferably by letting them inject their own code (security was not an issue).

Often, the injected parts would be very simple, so we decided to use a kind of policy-pattern, where users could pass callables to the algorithm, which would then call them at specific points during its execution. See this file for an example, around line 145.This lead to a lot of arguments for this function.

Worse, there was no sensible order to the arguments, and often we wanted some of them to be defaulted. While we could have used a struct and set its fields, this would have made the API harder for physicists, to whom that approach would not be intuitive.

So I decided to build a rather complex mechanism with named arguments in any, and to allow for defaults as well. So in a way the following is a refinement both of Jonathan's approach and my own previous work.

Note: While I don't think Named Arguments and unordered interfaces should be used indiscriminately, there are some cases where they can make a complex part of an API less so, at the expense of a bit more complex machinery for the developers.

Recap

In Jonathan post on Named Arguments, he arrives at the following:

```
// displayCoolName.hpp

void displayCoolNameImpl(FirstName const& theFirstName, LastName
const& theLastName);
template<typename Arg0, typename Arg1>
void displayCoolName(Arg0&& arg0, Arg1&& arg1)
{
   displayCoolNameImpl(pick<FirstName>(arg0, arg1),
pick<LastName>(arg0, arg1));
}

// displayCoolName.cpp

void displayCoolNameImpl(FirstName const& theFirstName, LastName
const& theLastName)
{
   std::cout << "My name is " << theLastName.get() << ", " <<
theFirstName.get() << ' ' << theLastName.get() << '.' << '\n';
}
```

Note: This works also without the named argument-syntax that is the main topic of that post. This is pretty cool! `displayCoolName` can now be called in any order we want, just by labelling our arguments at call-site. While this is not useful in all contexts, there are corner-cases where this can really improve an API. Let's see if we can generalize the approach a little. What we would like is to create a generic component that allows us to easily reproduce this pattern with

- different names,
- different impl-functions,

- and different parameters to be picked.

... without making the use of the component or the call to the resulting function more complex. That is a pretty tall order, and will require some atypical approaches.

Some preparation

First, let's simplify things a little bit by assuming that NamedTypes are cheap to copy. As they typically wrap either a built-in type or a (const) reference to something more complex, I think this is reasonable. It removes the need to consider everything in terms of references and using forwarding references etc.

A first approach

Different impl-functions and parameters could be achieved by e.g. passing a functor and a typelist:

```
// displayCoolName.hpp

template<typename... Args>
void genericPicker(F f, TypeList<PickArgs>, Args... args)
{
   auto tup = std::make_tuple(args...);
   f(std::get<PickArgs>(tup)...);
}

template<typename... Args>
void displayCoolName(Args... args)
{
   auto coolNameFunctor = [](FirstName firstName, LastName lastName)
   {
       displayCoolNameImpl(firstName, lastName);
   }
   genericPicker(coolNameFunctor, TypeList<FirstName, LastName>(),
args...)
}
```

However, this is definitely harder to use. It also doesn't solve 1: we still have to define the template for each function we want to use the pattern with. Back to the drawing board.

Thinking outside the Box

The first requirement is really quite hard – how do you create a function that can have different names? My solution to this issue uses the fact that there is a second way we can create the syntax of a global function: a callable global variable. I saw that approach when looking at the code for boost::hana, where it is used to e.g. implement if. We can rewrite our earlier approach to

```
// UnorderedCallable.hpp

template<class Function, class... OrderedParameters>
class UnorderedCallable
{
public:
   constexpr UnorderedCallable(F f): f_(f) {}
   template<class... CallParameters>
   void operator() (CallParameters... Ts) const
   {
       auto tup = std::make_tuple(args...);
       f(std::get<PickArgs>(tup)...);
   }
private:
   Function f_;
};

// displayCoolName.hpp

struct DisplayCoolNameImpl
{
   void operator() (FirstName theFirstName, LastName theLastName);
};
constexpr UnorderedCallable<DisplayCoolNameImpl, FirstName,
LastName> displayCoolName;
```

Now we're talking! This is definitely a reusable piece of code. However, we are still declaring the interface of the impl-function twice: once when we declare operator(), and once when we pass the argument-types to the `UnorderedCallable` template. That is repeat work, and a potential source of errors. It can be solved by moving the declaration of the impl-function into `UnorderedCallable`, and explicitly specializing the method:

```
// UnorderedCallable.hpp

template<class FunctionID, class... OrderedParameters>
class UnorderedCallable
{
```

```
public:
   constexpr UnorderedCallable(F f): f_(f) {}
   void impl(OrderedParameters... params) const ;
   template<class... CallParameters>
   void operator() (CallParameters... Ts) const
   {
       auto callParamTup = std::make_tuple(Ts...);
       impl( std::get<OrderedParameters>(callParamTup )...);
   }
};

// displayCoolName.hpp

using DisplayCoolName = UnorderedCallable<struct DisplayCoolNameID,
FirstName, LastName>
constexpr DisplayCoolName displayCoolName;
```

Almost there! The header and source look very close to those of a normal function.

Finishing off

We can do two more cheap improvements:

- Allow return values
- remove the named types from the impl-function by calling `.get()` in the template

With this, the final version is:

```
// UnorderedCallable.hpp

template<class, class F> // we only use the partial specialization
where F has the form Ret(Params)
class UnorderedCallable{
   static_assert(std::integral_constant<F>(false), "second template
parameter must be of function type: Ret(Params)")
}
template<class FunctionID, class Retval, class... OrderedParameters>
class UnorderedCallable<FunctionID, Ret(OrderedParameters)>
{
public:
   constexpr UnorderedCallable(F f): f_(f) {}
   Ret impl(typename OrderedParameters::type... params) const ;
   template<class... CallParameters>
   auto operator() (CallParameters... Ts) const
   {
       auto callParamTup = std::make_tuple(Ts...);
       return impl( std::get<OrderedParameters>(callParamTup
).get()...);
```

```
    }
};

// displayCoolName.hpp

using FirstName = NamedType<std::string const&, struct FirstNameID>;
using LastName = NamedType<std::string const&, struct LastNameID>;
using DisplayCoolName = UnorderedCallable<struct DisplayCoolNameID,
void(FirstName, LastName)>
constexpr DisplayCoolName displayCoolName;
// displayCoolName.cpp
void DisplayCoolName::impl(std::string const& theFirstName,
std::string const& theLastName)
{
    std::cout << "My name is " << theLastName << ", " << theFirstName
<< ' ' << theLastName << '.' << '\n';
}
```

Using `NamedType` of references together with the function form for the template parameters to `UnorderedCallable` makes the declaration look like simple pseudocode. Using the underlying type directly in the implementation-function makes the implementations' bodies look exactly like normal function definitions, without losing the usefulness of `NamedTypes`. The only danger I see is that when you want to swap the order of the two arguments, the compiler would not help you. However, you don't ever need to do that because you can pass the arguments in in any order anyway, and it makes more complex function definitions way easier to read without all the `.get()` calls. Note that this may require some slight adjustments to `NamedType` itself. The complete version of this approach is in the repo, which also has support for default values.

Don’t Make Your Interfaces *Deceptively* Simple

Just because we *can* provide an interface doesn’t mean that we *should*.

At least this is one of the takeaways that I got from from Howard Hinnant’s opening keynote at Meeting C++ 2019.

In this impressive keynote, Howard made a presentation about `<chrono>` and the host of features it brings in C++20. But beyond showing us how to use `<chrono>`, Howard explained some of the design rationale of this library.

Those are precious lessons of design, especially coming from someone who had a substantial impact on the design of the standard library. I believe we can apply those practices to our own code when designing interfaces.

So, just because we can provide an interface doesn’t mean that we should. To illustrate what this means in practice, let’s go over two examples in the C++ standard library.

Thanks to Howard Hinnant for reviewing this article.

std::list doesn’t provide operator[]

Contrary to `std::vector`, C++ standard doubly linked list `std::list` doesn’t have an `operator[]`. Why not?

It’s not because it’s technically impossible. Indeed, here is one possible, even simple, implementation for an `operator[]` for `std::list`:

```
template<typename T>
typename std::list<T>::reference std::list<T>::operator[](size_t
index)
{
    return *std::next(begin(), index);
}
```

But the problem with this code is that providing access to an indexed element in the `std::list` would require iterating from `begin` all the way down the position of the element. Indeed, the iterators of `std::list` are only bidirectional, and not random-access.

`std::vector`, on the other hand, provides random-access iterators that can jump anywhere into the collection in constant time.

So even if the following code would look expressive:

```
auto const myList = getAList();
auto const fifthElement = myList[5];
```

We can argue that it's not: it does tell what the code really does. It looks simple, but it is *deceptively* simple, because it doesn't suggest that there we're paying for a lot of iterations under the cover.

If we'd like to get the fifth element of the list, the STL forces us to write this:

```
auto const myList = getAList();
auto fifthElement = *std::next(begin(myList), 5);
```

This is less concise, but it shows that it starts from the beginning of the list and iterates all the way to the fifth position.

It is interesting to note that both versions would have similar performance, and despite the first one is simpler, the second one it better. This is maybe not an intuitive thought at first, but when we think about it it makes perfect sense.

Another way to put it is that, even if expressive code relies on abstractions, too much abstraction can be harmful! A good interface has to be at the right level of abstraction.

year_month_day doesn't add days

Let's get to the example taken from the design of `<chrono>` and that led us to talk about this topic in the first place.

`<chrono>` has several ways to represent a date. The most natural one is perhaps the C++20 long awaited `year_month_day` class which, as its name suggests, is a data structure containing a year, a month and a day.

But if you look at the `operator+` of `year_month_day` you will see that it can add it years and months… but not days!

For example, consider the following date (note by the way the overload of `operator/` that is one of the possible ways to create a date):

```
using std::chrono;
using std::literals::chrono_literals;

auto const newYearsEve = 31d/December/2019;
```

Then we can't add a day to it:

```
auto const newYearStart = newYearsEve + days{1}; // doesn't compile
```

(Note that we use `days{1}` that represents the duration of one day, and not `1d` that represents the first day of a month)

Does this mean that we can't add days to a date? Is this an oversight in the library?

Absolutely not! Of course the library allows to add days to dates. But it forces you to make a detour for this, by converting your `year_month_date` to `sys_days`.

sys_days

`sys_days` is the most simple representation of a date: it is the number of days since a certain reference epoch. It is typically January 1st, 1970:

- …
- December 31st, 1969 is -1
- January 1st, 1970 is 0
- January 2nd, 1970 is 1,
- …

- December 31st, 2019 is 18261
- ...

`sys_days` just wraps this value. Implementing the sum of a `sys_days` and a number of days is then trivial.

Adding days to year_month_day

To add a day to a `year_month_day` and to get another `year_month_day` we need to convert it to `sys_days` and then back:

```
year_month_day const newYearStart = sys_days{newYearsEve} + days{1};
```

Adding days to a `year_month_day` could be easily implemented by wrapping this expression. But this would hide its complexity: adding days to a `year_month_day` could roll it into a new month and this requires executing complex calendar calculations to determine this.

On the other hand, it is easy to conceive that converting from `year_month_day` and back triggers some calendar-related calculations. The above line of code then makes it clear for the user of the interface where the calculations happen.

On the other hand, providing an `operator+` to add days to `year_month_day` would be simple, but *deceptively* simple.

Make your interfaces easy to use correctly and hard to use incorrectly. Make them simple, but not deceptively simple.