INTERFACE DESIGN IN C++ - PART I

- **Make your functions functional**
- **How to Be Clear About What Your Functions Return**
- **How to Use Tag Dispatching In Your Code Effectively**
- **When to Use Enums and When to Use Tag Dispatching in C++**

Fluent {C++}

Make your functions functional

Introduction : global variables

Global variables are a Bad Thing. Everyone knows this, right ?

But do you know exactly why ? I have asked this question around, and many of us can't exactly explain *why* global variables should be avoided.

It is not a question of scope. Indeed, global constants have the same scope as global variables, but global constants are generally seen as a Good Thing, because they let you put a label over what would otherwise be "magic values".

Some people answer that global variables should be avoided because they cause multithreading issues. They do cause multithreading issues, because a global variable can be accessed from any function, and could be written and read simultaneously from several threads, but I don't think this is the main issue. Because, like everyone knows, global variables should be avoided even when there is only a single thread in a program.

I think that global variables are a problem because they **break functions**.

Functions are useful to decompose a program (or another function) into simpler elements, and for this reason, they reduce complexity, and are a tool to improve expressiveness of the code. But to do this, functions must respect certain rules. One of the rules to respect stems from the very definition of a function:

A function takes inputs, and provides outputs.

It sounds simple, because it is. And to keep it simple, the important thing to understand is that a function must clearly **show what its inputs and outputs are**. This is where global

variables break functions. As soon as there is a global variable, every function in its scope can potentially have this global variable as input and/or output. And this is **hidden** from the function declaration. So the function has inputs and outputs, but does not tell exactly what they are. Such functions are… dysfunctional.

Note how global constants don't have this issue. They are not an input of a function, because they cannot vary (as input does by definition), and they are certainly not an output either, because the function cannot write in them.

As a result, a function must clearly express its input and outputs. This idea happens to be at the basis of functional programming, so we could formulate the guideline this way:

Make your functions functional !

The rest of this post shows how to do this in an idiomatic way in C++.

Expressing the inputs of a function

Quite simply, inputs come in to a function through its parameters. Generally, inputs are expressed by passing a reference-to-const parameter (const T&). So when you read or write a function prototype, bear in mind that **reference-to-const** means **input**. For some types, input can also come in by **value** (like primitive types for instance).

Expressing the input-output parameters

C++ allows to modify inputs of a function. Such parameters are both input and output. The typical way to represent this is by **reference-to-not-const** (T&).

Expressing the outputs of a function

The rule here is:

Outputs should come out by the return type.

```
Output f(const Input& input);
```

This does sound natural, but there are many cases where we are reluctant to do this, and instead a more clumsy way is often seen: passing the output in parameter as a reference-to-not-const (T&), like so:

```
void f(const Input& input, Output& output);
```

Then the function would be in charge of filling this output parameter.

There are several drawbacks with using this technique:

- It is not natural. Outputs should come out by the return type. With the above code, you end up with an awkward syntax at call site:

```
Output output;
f(input, output);
```

As opposed to the simpler syntax:

```
Output output = f(input);
```

And this gets even more awkward when there are several functions called in a row.

- You have no guarantee that the function is actually going to fill the output,
- Maybe it does not make sense to default-construct the Output class. In this case you would force it to be, for a questionable reason.

If producing outputs through the return type is better, why doesn't everyone do it all the time ?

There are 3 types of reasons that prevent us from doing it. And all of them can be worked around, most of the time very easily. They are : performance, error handling and multiple return type.

Performance

In C, returning by value sounded like folly, because it incurred a copy of objects, instead of copying pointers. But in C++ there are several language mechanisms that elide the copy when returning by value. For instance Return Value Optimisation (RVO) or move semantics do this. For example, returning any STL container by value would move it instead of copying it. And moving an STL container takes about as much time as copying a pointer.

In fact you don't even have to master RVO or move semantics to return objects by value. **Just do it !** In many cases the compiler will do its best to elide the copy, and for the cases it doesn't, you have over 80% probability that this code is not in the critical section for performance anyway.

Only when your profiler showed that a copy made during a return by value of a specific function is your bottleneck for performance, you could think of degrading your code by passing the output parameter by reference. And even then, you could still have other options (like facilitating RVO or implementing move semantics for the returned type).

Error handling

Sometimes a function may not be able to compute its output in certain cases. For example the function may fail with certain inputs. Then what can be returned if there is no output ?

In this case some code falls back to the pattern of passing output by reference, because the function *doesn't have to* fill it. Then to indicate whether the output was filled or not, the function returns a boolean or an error code like:

```
bool f(const Input& input, Output& output);
```

This make for a clumsy and brittle code at call site:

```
Output output;
bool success = f(input, output);
if (success)
{
   // use output ...
}
```

The cleanest solution for the call site is for the function to throw an exception when it fails, and return an output when it succeeds. However, the surrounding code has to be exception safe, and many teams don't use exceptions in their codeline anyway.

Even then, there is still a solution to make output come out by the return type: use **optional**.

You can see all about optional in a dedicated post, but in short, optional<T> represent an object that can be any value of type T, or empty. So when the function succeeds, you can return an optional containing the actual output, and when it fails, you can just return an empty optional:

```
boost::optional<Output> f(const Input& input);
```

Note that optional is in the process of standardization and will be natively available in C++17.

And at the calling site:

```
auto output = f(input); // in C++11 simply write auto output =
f(input);
if (output)
{
   // use *output...
}
```

Multiple return types

In C++, only one type can be returned from a function. So when a function must return several outputs, the following pattern is sometimes seen:

```
void f(const Input& intput, Output1& output1, Output2& output2);
```

Or worse, asymmetrically:

```
Output1 f(const Input& input, Output2& output2);
```

Still falling back to the dreaded pattern of passing outputs by reference.

The cleanest solution to fix this and produce several outputs by return type, as the language stands today (< C++17), is defining a new structure grouping the outputs:

```
struct Outputs
{
   Output1 output1;
   Output2 output2;
};
```

Which leads to the more expressive declaration:

```
Outputs f(const Input& input);
```

If the two outputs are often together, it might even make sense to group them in an actual object (with private data and public methods), although this is not always the case.

In C++11, a quicker but less clean solution is to use tuples:

```
std::tuple<Output1, Output2> f(const Input& input);
```

And at call site:

```
Output1 output1;
Output2 output2;
std::tie(output1, output2) = f(inputs);
```

This has the drawback of forcing the outputs to be default constructible. (If you're not familiar with tuples yet, don't worry, we get into the details of how the above works when we explore tuples in a dedicated post).

As a final note, here is a syntax that will probably be integrated in C++17 to natively return multiple values:

```
auto [output1, output2] = f(const Input& input);
```

This would be the best of both worlds. It is called Structured Bindings. f would return an std::tuple here.

Conclusion

In conclusion, strive to have outputs coming out of your functions by their return type. When this is impractical, use another solution, but bear in mind that it is detrimental for the clarity and expressiveness of your code.

How to Be Clear About What Your Functions Return

What's in a function's interface?

In most languages, a function's interface has 3 main parts:

- the function's name: it indicates what the function does,
- the function's parameters: they show what the function takes as input to do its job,
- the function's return type: it indicates the output of the function.

```
ReturnType functionName(ParameterType1 parameterName1,
ParameterType2 parameterName2);
```

So far, so good.

But when looking at this prototype, we can notice that something isn't symmetric: the function's parameters have both a type and a name, while the returned value only has a type. Indeed, **the return value doesn't have a name**.

In a function declaration, one could choose to also omit the names of the parameters. But still, the return type doesn't have a choice. It can only be… a type.

Why is that? My take is that it's because we expect the the function's name to be clear enough to express what it returns, plus the returned value has a visible type. So a name for the returned value itself would be superfluous.

But is this the case 100% of the time?

A use case that should not exist, but that does

No. In theory it works fine but, realistically, it's not always the case that a function's name informs you exactly of what to expect as a return value.

Let's take the example of a function that performs a side effect, like saving a piece of information in a database:

```
void save(PieceOfData const& preciousData);
```

And say that this operation could potentially fail. How does the function lets it caller know whether or not the operation succeeded?

One way to go about that is to make the `save` function throw an exception. It works, but not everyone uses exceptions (exceptions need exception-safe code surrounding them, they may impact performance, some teams ban them from their coding conventions...). There have been hot debates and suggested alternatives about this.

We already come across a clear way to indicate that a function could potentially fail to return its result: using optionals. That is to say, return an `optional<T>`, conveying the message that we expect to return a `T`, but this could potentially fail, and the function caller is supposed to check whether that returned `optional` is full or empty.

But here we're talking about a function that returns **nothing**. It merely saves piece of data in a database. Should it return an `optional<void>` then? This would read that it is supposed to return `void` but it may return something that isn't really a `void`, but an empty box instead. An empty void. Weird. And `std::optional<void>` doesn't compile anyway!

Another possibility is to return a boolean indicating whether or not the function succeeded:

```
bool save(PieceOfData const& preciousData);
```

But this is less than ideal. First, the returned value could be ignored at call site. Though this could be prevented by adding the `[[nodiscard]]` attribute in C++17:

```
[[nodiscard]] bool save(PieceOfData const& preciousData);
```

Second, just by looking at the function's prototype, we don't know if that `bool` means success or failure. Or something else totally unrelated, for that matter. We could look it up in the documentation of the function, but it takes more time and introduces a risk of getting it wrong anyway.

Since the function is only called "`save`", its name doesn't say what the return type represents. We could call it something like `saveAndReturnsIfSuceeded` but... we don't really want to see that sort of name in code, do we?

Meta information

It is interesting to realize that this is a more general use case that just failure or success. Indeed, sometimes the only way to retrieve a piece of information about a certain operation is to actually perform it.

For instance, say we have a function that takes an `Input` and uses it to add and to remove entries from a existing `Entries` collection:

```
void updateEntries(Input const& input, Entries& entries);
```

And we'd like to retrieve some data about this operation. Say an `int` that represents the number of entries removed, for example. We could make the function output that `int` via its return type:

```
int updateEntries(Input const& input, Entries& entries);
```

But the return type doesn't tell what it represents here, only that it's implemented as an `int`. We've lost information here.

In this particular case, we could have added an `int& entriesRemoved` function parameter, but I don't like this pattern because it forces the caller to initialize a variable before calling the functions, which doesn't work for all types, and a non-const reference means input-output and not output, so it's not exactly the message we'd like to convey here.

What to do then?

Named return types: strong return types?

So in summary, we have return types that lack a meaningful name. This sounds like a job for strong types: indeed, strong types help put meaningful names over types!

Spoiler alert: strong types won't be the option that we'll retain for most cases of return types in the end. Read on to see why and what to use instead.

Let's use `NamedType` as an implementation of strong types, and create return types with a name that make sense in each of our functions' contexts.

So our `save` function returns a `bool` that is `true` if the operation was a success. Let's stick a name over that `bool`:

```
using HasSucceeded = NamedType<bool, struct HasSucceededTag>;
```

The second parameter of `NamedType` is a "phantom type", that is to say that it's there only for differentiating `HasSucceeded` from another `NamedType` over a `bool`.

Let's use `HasSucceeded` in our function's interface:

```
HasSucceeded save(PieceOfData const& preciousData);
```

The function now expresses that it returns the information about whether the operation succeeded or not.

The implementation of the function would build a `HasSucceeded` and return it:

```
HasSucceeded save(PieceOfData const& preciousData)
{
    // attempt to save...
    // if it failed
    return HasSucceeded(false);
    // else, if all goes well
    return HasSucceeded(true);
}
```

And at call site:

```
HasSucceeded hasSucceeded = save(myData); // or auto hasSucceeded =
...

if(!hasSucceeded.get())
{
    // deal with failure...
```

Note that we can choose to get rid of the call to `.get()` by making `HasSucceeded` use the `FunctionCallable` skill.

For the sake of the example, let's apply the same technique to our `updateEntries` function:

```
using NumberOfEntriesRemoved = NamedType<int, struct
NumberOfEntriesRemovedTag>;

NumberOfEntriesRemoved updateEntries(Input const& input, Entries&
entries);
```

By looking at the interface, we now know that it outputs the number of entries removed via the return type.

Just a weak type will do here

The above works, but it's needlessly sophisticated. In this case, the only thing we need is a name for other human beings to understand the interface. We don't need to create a specific type used only in the context of the return type to also let the compiler know what we mean by it.

Why is that? Contrast our example with the case of input parameters of a function:

```
void setPosition(int row, int column);

// Call site
setPosition(36, 42);
```

Since there are several parameters that could be mixed up (and the program would still compile), introducing strong types such as `Row` and `Column` are useful to make sure we pass the parameters in the right order:

```
void setPosition(Row row, Column column);

// Call site:
setPosition(Row(36), Column(42));
```

But in the return type, what is there to mix up? There is only one value returned anyway!

So a simple alias does the job just well:

```
using HasSucceeded = bool;
HasSucceeded save(PieceOfData const& preciousData);
```

This is the **most adapted solution** in this case, in my opinion.

The case where strong types *are* useful in return types

However there are at least two specific cases where strong types are helpful to clarify a returned value.

One is to use strong types to return multiple values.

The other is when you already have a strong type that represents the return value, and that you **already use** at other places in the codeline. For instance, if you have a strong type `SerialNumber` that strengthen a `std::string`, and you use it at various places, it makes perfect sense to return it from a function.

The point I want to make is not to create a strong type for the sole purpose of returning it from a function and immediately retrieving the value inside it afterwards. Indeed, in this case **a classical alias will do**.

What's in an expressive function's interface?

This technique helps us be more explicit about what it is that a function is returning.

This is part of a more general objective, which is to leverage on every element of the function to express useful information:

- a clear function name: by using good naming,
- well-designed function parameters (a 3-post series coming soon),
- an explicit output: either by returning the output directly (thus making functions functional), or by using an optional or, if it comes to that, returning something else, like we saw today. But always, by being the clearest possible about it.

How to Use Tag Dispatching In Your Code Effectively

Constructors lack something that the rest of the functions and methods have in C++: **a name**.

Indeed, look at the following code:

```
class MyClass
{
public:
    MyClass();
    void doThis();
    void doThat();
};

void doSomethingElse(MyClass const& x);
```

Every routine has a name that says what it does, except for the constructor, which only bears the name of its class.

There is some logic in this though: its a constructor, so its job is to... construct the class. And if it had a name it would be something like `constructMyClass`, so what's the point, let's just call it `MyClass` and give it a constructor syntax. Fine.

Except this becomes a problem when we need **several ways** to construct the class: `constructMyClassThisWay` and `constructMyClassThatWay`. To remedy to that, constructors can be overloaded:

```
class MyClass
{
public:
    MyClass();
    MyClass(int i);
    MyClass(std::string s);
};
```

Which is good, but sometimes not enough. Indeed, sometimes we need **several ways** to construct a class with the **same types of parameters**. The simplest example of that is

default construction, that is a constructor taking no parameters, to which we want to affect different behaviours.

The thing I want you to see here is that different overloads allow several constructors taking **different types of data**. But there is no native way to have several constructors taking the same types of data, but with **different behaviours**.

One way to go about this and to keep code expressive is to use **tag dispatching**. This is the topic of today: how to use tag dispatching in your code and, just as importantly, **when to use it and when to avoid it.** In the opinion of yours truly, that is.

How tag dispatching works

If you are already familiar with tag dispatching you can safely skip over to the next section.

The "tag" in tag dispatching refers to a type that has no behaviour and no data:

```
struct MyTag {};
```

The point of this is that, by creating several tags (so several types), we can use them to route the execution through various overloads of a function.

The STL uses this technique quite intensively in algorithms that have different behaviours based of the capabilities of the iterator type of the ranges they are passed. For instance, consider the function `std::advance`, which takes an iterator and moves it forward by a given number of steps:

```
std::vector<int> v = { 1, 2, 3, 4, 5 };
auto it = v.begin(); // it points to the 1st element of v
std::advance(it, 3); // it now points to the 4th element of v
```

If the underlying iterator of the collection is a forward iterator then `std::advance` applies `++` on it 3 times, whereas if it is a random-access iterator (like it is the case for `std::vector`), it calls `+= 3` on it. Even if you're not familiar with this, the bottom line is that `std::advance` can behave differently depending on a propriety of its iterator.

To implement that, the STL typically uses tag dispatching: the iterator provides a tag (how it provides it is outside of the scope of this article): `forward_iterator_tag` for forward iterators, and `random_access_iterator_tag` for random access iterators. The implementation of `std::advance` could then use something like:

```
template <typename Iterator, typename Distance>
void advance_impl(Iterator& it, Distance n, forward_iterator_tag)
{
    while (--n >= 0)
        ++it;
}

template <typename Iterator, typename Distance>
void advance_impl(Iterator& it, Distance n, random_iterator_tag)
{
    it += n;
}
```

and call `advance_impl` by instantiating the correct tag depending on the capabilities of the iterator. Function overloading the routes the execution to the right implementation.

How to use tag dispatching in your code

Even if it is not as technical as the implementation of the STL, you can still benefit from tag dispatching **in your own code**.

Let's take the typical example of a class that has a default constructor (that is, taking no parameter) and where you want this constructor to behave in different ways depending on the **context** you're calling it from.

In that case you can define your own tags. You can put them in the scope of the class itself to avoid polluting the global namespace:

```
class MyClass
{
public:
    struct constructThisWay{};
    struct constructThatWay{};
```

```
    // ...
```

And then you have the associated constructors:

```
class MyClass
{
public:
    struct constructThisWay{};
    struct constructThatWay{};

    explicit MyClass(constructThisWay);
    explicit MyClass(constructThatWay);

    // ...
};
```

These are no longer "default" constructors, because they are more than one. They are constructors that take no data, but that can behave in different ways. I used the keyword `explicit` because this is the default (no pun intended!) way to write constructor accepting one parameter, in order to prevent implicit conversions. When you're not 100% sure that you want implicit conversion and that you know what you're doing, better block them.

The call site then looks like this:

```
MyClass x((MyClass::constructThisWay()));
```

Note the abundance of parentheses. This feeling of Lisp is a way to work around C++'s **most vexing parse**, as Scott Meyers calls it in Effective STL, Item 6. Indeed if you don't double-parenthesize, the following code is parsed as a function declaration:

```
MyClass x(MyClass::constructThisWay());
```

(Note that we wouldn't face the most vexing parse here if there were another parameter passed to the constructor and that wasn't instantiated directly at call site like the tag is).

One way out of this is to use uniform initialization, with braces {}:

```
MyClass x(MyClass::constructThisWay{});
```

But there is another way to have less parentheses or braces: declaring **tag objects** along with tag types. But this makes for a less concise class definition:

```
class MyClass
{
public:
    static struct ConstructThisWay{} constructThisWay;
    static struct ConstructThatWay{} constructThatWay;

    explicit MyClass(ConstructThisWay);
    explicit MyClass(ConstructThatWay);
};
```

While the call site looks a little prettier:

```
MyClass x(MyClass::constructThatWay);
```

No more most vexing parse nor braces, since the argument is no longer a type. But this leads to more code in the class definition. It's a trade-off. You choose.

Finally, whichever way you decide to go with, nothing prevents you from having a real default constructor that takes no parameters, on the top of all that:

```
class MyClass
{
public:
    static struct ConstructThisWay{} constructThisWay;
    static struct ConstructThatWay{} constructThatWay;

    MyClass();
    explicit MyClass(ConstructThisWay);
    explicit MyClass(ConstructThatWay);
};
```

Why not use enums instead?

A natural reaction when you first see this technique of tags in business code is to wonder: wouldn't using an enum be a less convoluted way to get the same results?

In fact there are notable differences between using enums and using tags, and since there are quite a few things to say about that I've dedicated an entire post to when to use tag dispatching and when to use enums, coming next up in this series.

So back to tag dispatching.

When to use tag dispatching in your code

Use tag dispatching to provide additional information on behaviour, and strong types to provide additional information on the data.

My take on tag dispatching is that is should be used to **customize behaviour, and not to customize data**. Said differently, tag dispatching should be used to supplement the data passed to a constructor, with additional information on behaviour.

To illustrate, I'm going to show you a **bad example** of usage of tag dispatching. This is a class that represents a circle, that can be constructed either with a radius or with a diameter. Both a radius and a diameter are numeric values of the same type, expressed say with `double`.

So a **wrong usage** of tag dispatching is this:

```
class Circle
{
public:
    struct buildWithRadius{};
    struct buildWithDiameter{};

    explicit Circle(double radius, buildWithRadius);
    explicit Circle(double diameter, buildWithDiameter);
};
```

What is wrong in this code is that the information about the data is **spread over several arguments**. To fix this we can use **strong types** rather than tag dispatching to add information to the data:

```
class Circle
{
public:
    explicit Circle(Radius radius);
    explicit Circle(Diameter diameter);
};
```

Curious about strong types? Check out this series of posts on strong types!

So use tag dispatching to provide additional information on behaviour, and strong types to provide additional information on the data.

If you find this guideline reasonable, you may wonder why the STL doesn't follow it. Indeed, as seen above, the dispatch tags on the iterator categories are passed along with the iterator itself.

Not being a designer of the STL I could be wrong on that, but I can think of this: since the algorithm gets the iterator category from the iterator in a generic way, it would need a template template parameter to represent the strong type. Like `ForwardIterator` to be used like this: `ForwardIterator<iterator>`. And from the implementation of the iterator it may be less simple that specifying a tag. Or maybe it's more code to define strong types. Or maybe it's related to performance. Or maybe they just didn't think about it this way. Frankly I don't know, and would be glad to have your opinion on that.

Anyway in your own business code, when there is no generic code creating intricate design issues, I recommend you to **use tag dispatching to provide additional information on behaviour, and strong types to provide additional information on the data**. It will make your interface that much clearer.

When to Use Enums and When to Use Tag Dispatching in C++

Enums and tag dispatching are two ways to introduce several behaviours in the same interface in C++. With them, we can pass arguments that determine a facet of how we want a function to behave.

Even if enums and tag dispatching have that in common, they achieve it in **a quite different way**. Realizing what these differences are will give you tools to decide which one to use in any given situation.

To differentiate behaviour we could also use templates and runtime polymorphism based on inheritance, but I'm leaving these out of this discussion in order to focus on the differences between enums and tag dispatching specifically.

I think these differences boil down to three things: the structure they give to code, their moments of resolution, and how explicit their call site can be.

Separation of code

With tag dispatching, the code for each behaviour is localized in a separate function:

```
struct BehaveThisWay{};
struct BehaveThatWay{};

void f(int argument, BehaveThisWay);
{
    // use argument this way
}

void f(int argument, BehaveThatWay);
{
    // use argument that way
}
```

Whereas enums group the code for all the behaviours into the same function:

```
enum class BehaviourType
```

```
{
    thisWay,
    thatWay
};

void f(int argument, BehaviourType behaviourType);
{
    // check the value of behaviourType and behave accordingly
}
```

This can be either good or bad. If the various behaviours use really different code, as in `std::advance` for instance, then the separation of code brought by tag dispatching leads to a **separation of concerns**, which is a good thing.

However, if the implementation of the function is roughly the same for all behaviours, and changes only in local points in the function, then you're better off grouping everything in the same function and testing the enum at the few places it is needed.

Also, if you have n tags argument that can take m values each, the number of overloads grows exponentially to m^n. This is sustainable for a little number of arguments only (but you don't want your functions to accept too many arguments in general anyway).

Moments of resolution

Essentially, tags get dispatched at **compile-time** while enums values can be read at **runtime**.

Indeed, tag dispatching relies on function overloading. The calling site that passes a `BehaveThisWay` or a `BehaveThatWay` (or an object it receives from further up the call stack and that can be of one of those types) is compiled into binary code that calls either one function. So the behaviour of `f` for a particular call site is **hardwired** during compilation.

On the contrary, enums can be read at runtime, which allows deferring the value the enum takes at a particular call site until runtime, if necessary. This value could typically depend on a value coming into the system, provided by the user for example.

If the interface uses tag dispatching but the call site needs to wait until runtime to know which behaviour to choose, then its client is forced to jump through loops to use it:

```
if (myBehaviour == BehaviourType::thisWay)
{
    f(value, BehaveThisWay());
}
else if (myBehaviour == BehaviourType::thatWay)
{
    f(value, BehaveThatWay());
}
```

So if you know that your interface will be used with runtime info when you design it, you may want to consider enums over tag dispatching for that reason.

Explicit mention of the type

Finally, there is another difference between using enums and using tag dispatching: the enum forces you to write its type at call site:

```
f(value, BehaviourType::thisWay);
```

That is, if you're using an `enum class` and not a C `enum`. But that's what you want to use anyway, right?

You may find this extra `BehaviourType` more explicit, or needlessly verbose. I think it depends on taste, but I find it nice to write the type of an enum when it represents a question, to which the value of the enum is an answer.

For example, let's consider this function that write to a file, shamelessly inspired from its Lisp counterpart:

```
enum class IfExists
{
    supersede,
    doNothing
};

void writeToFile(std::string const& fileName, std::string const&
data, IfExists whatIfExists);
```

Then the call site would look like this:

```
writeToFile("myFile.txt", "contents", IfExists::supersede);
```

I find this looks lovely, doesn't it? It's because the enums answers a question: "what to do if it (the file) already exists?" Well, "supersede" it!

Note that you could achieve the same result with tag dispatching, if you need it for one of the reasons we saw, like separating the concerns in your code:

```
struct IfExists
{
    static struct Supersede {} supersede;
    static struct DoNothing {} doNothing;
};

void writeToFile(std::string const& fileName, std::string const&
data, IfExists::Supersede)
{
    // supersede if file exists
}

void writeToFile(std::string const& fileName, std::string const&
data, IfExists::DoNothing);
{
    // do nothing if file exists
}
```

It's like a tag inside a tag, if you want. And the call site still looks like that:

```
writeToFile("myFile.txt", "contents", IfExists::supersede);
```

Varying behaviours

Now there is much more than tag dispatching and enums to determine which behaviour to execute. For example, there are virtual functions for choosing behaviour at runtime, or policy-based design (see Modern C++ Design to dive into this – I recommend you to do so) for compile-time polymorphism.

But for a local, simple choice between several behaviours, enums and tag dispatching are concise ways to do the job. And knowing the differences between the two will help you pick the right one with reasoned arguments.