FUNDAMENTAL ASPECTS OF THE STL

- **STL Function objects: Stateless is Stressless**
- **Custom comparison, equality and equivalence with the STL**
- **The design of the STL**

Fluent {C++}

STL Function objects: Stateless is Stressless

The need for function objects arises almost as soon as you start using the STL. This post shows how to design them so that they contribute in making your code using the STL more expressive and more robust.

Function objects

Here is a brief recap on function objects before getting to the meat. If you're already familiar with them you can skip to the next section.

A **function object** is an object that can be used in a function call syntax:

```
myFunctionObject(x);
```

even though it is declared with a class (or a struct). This syntax is allowed by the declaration of an `operator()`:

```
class MyFunctionObject
{
public:
    void operator()(int x)
    {
        ....
    }
}
```

The advantage of function objects over simple functions is that function objects can **embark data**:

```
class MyFunctionObject
{
public:
    explicit MyFunctionObject(Data data) : data_(data) {}
    void operator()(int x)
    {
        ....usage of data_....
    }
private:
    Data data_;
}
```

And at call site:

```
MyFunctionObject myFunctionObject(data);

myFunctionObject(42);
```

This way the function call will use both `42` and `data` to execute. This type of object is called a **functor**.

In C++11, **lambdas** fill the same need with a lighter syntax:

```
Data data;
auto myFunctionObject = [data](int x){....usage of data....};

myFunctionObject(42);
```

Since lambdas arrived in the language in C++11, functors are much less used, although there remains some cases where you need to use them as will be shown in a dedicated post (scheduled Feb 07th).

Functions, functors and lambdas can be used with the same function-call syntax. For this reason they are all **callables**.

Callables are used profusely with the STL because algorithms have generic behaviours that are customized by callables. Take the example of `for_each`. `for_each` iterates over the elements of a collection and does *something* with each of them. This something is described by a callable. The following examples bump up every number of a collection by adding 2 to them, and show how to achieve this with a function, a functor and a lambda:

- with a **function** the value 2 has to be hardcoded:

```
void bump2(double& number)
{
    number += 2;
}

std::vector<double> numbers = {1, 2, 3, 4, 5};

std::for_each(numbers.begin(), numbers.end(), bump2);
```

with a **functor**, the bump value can be a passed as a parameter, which allows greater flexibility but with a heavier syntax:

```
class Bump
{
public:
    explicit Bump(double bumpValue) : bumpValue_(bumpValue) {}
    void operator()(double& number) const
    {
        number += bumpValue_;
    }
```

```
private:
    double bumpValue_;
};

std::vector<double> numbers = {1, 2, 3, 4, 5};

std::for_each(numbers.begin(), numbers.end(), Bump(2));
```

- and the **lambda** allows the same flexibility, but with a lighter syntax:

```
std::vector<double> numbers = {1, 2, 3, 4, 5};

double bumpValue = 2;
std::for_each(numbers.begin(), numbers.end(),
[bumpValue](double& number){number += bumpValue;});
```

These examples show the syntax to manipulate function objects with the STL. Now here is the guideline to use them effectively: **keep state away from them**.

Avoid keeping a state in function objects

It may be tempting, especially when you start out using the STL, to use variables in the data embarked in your function objects. Like for storing current results updated during the traversal of the collection, or for storing sentinels for example.

Even though lambdas supersede functors in standard cases, many codebases are still catching up to C++11 (as exposed in this article) and don't have lambdas available yet. Moreover as mentioned above, there remains cases that can only be solved by a functor. For these reasons I want to cover functors as well as lambdas in this post and in particular see how this guideline of avoiding state applies to both of them.

Functors

Let's consider the following code that aims at counting the number of occurrences of the value 7 in the collection `numbers`.

```
class Count7
{
public:
    Count7() : counter_(0) {}
    void operator()(int number)
    {
        if (number == 7) ++counter_;
    }
    int getCounter() const {return counter_;}
private:
    int counter_;
};
```

At call site, this functor can be used this way:

```
std::vector<int> numbers = {1, 7, 4, 7, 7, 2, 3, 4};

int count = std::for_each(numbers.begin(), numbers.end(),
Count7()).getCounter();
```

Here we instantiate a functor of type Count7 and pass it to for_each (the searched number could be parametrized in the functor to be able to write Count(7), but this is not the point here. Rather, I want to focus on the state maintained in the functor). `for_each` applies the passed functor to every element in the collection and then *returns* it. This way we get to call the `getCounter()` method on the unnamed functor returned by `for_each`.

The convoluted nature of this code hints that something is wrong in its design.

The problem here is that the functor has a state: its member `counter_`, and functors don't play well with state. To illustrate this, you may have wondered: why use this relatively unknown feature of the return value of `for_each` ? Why not simply write the following code:

```
std::vector<int> numbers = {1, 7, 4, 7, 7, 2, 3, 4};

Count7 count7;
std::for_each(numbers.begin(), numbers.end(), count7);

int count = count7.getCounter();
```

This code creates a counting functor, passes it to `for_each` and retrieves the counter result. The problem with this code is that **it simply doesn't work**. If you try to compile it you will see that the value in `count` is 0. Can you see why?

The reason is that, surprising at it sounds, count7 has never reached the inside of `for_each`. Indeed `for_each` takes its callable by value, so it is a *copy* of count7 that was used by `for_each` and that had its state modified.

This is the first reason why you should avoid states in functors: **states get lost**.

This is visible in the above example, but it goes further that this: `for_each` has the specificity of keeping the same instance of functor all along the traversal of the collection, but it is not the case of all algorithms. Other algorithms do not guarantee they will use the same instance of callable along the traversal of the collection. Instances of callables may then be copied, assigned or destructed within the execution of an algorithm, making the maintaining of a state impossible. To find out exactly which algorithm provides the guarantee, you can look it up in the standard but some very common ones (like `std::transform`) do not.

Now there is another reason why you should avoid states within function objects: it makes code more **complex**. Most of the time there is a better, cleaner and more expressive way. This also applies to lambdas, so read on to find out what it is.

Lambdas

Let's consider the following code using a lambda that aims at counting the number of occurrences of the number 7 in `numbers`:

```
std::vector<int> numbers = {1, 7, 4, 7, 7, 2, 3, 4};

int count = 0;
std::for_each(numbers.begin(), numbers.end(),
              [&count](int number){ if (number == 7) ++count;});
std::cout << count << std::endl;
```

This code calls a `for_each` to traverse the whole collection and increments the variable counter (passed by reference to the lambda) each time a 7 is encountered.

This code is not good because it is **too complex** for what it is trying to do. It shows the technical way of counting elements by exposing its state, whereas it should simply tell that it is counting 7s in the collection, and any implementation state should be abstracted away.

This really ties up with the principle of Respecting levels of abstraction, which I deem to be the most important principle for programming.

What to do then?

Pick the right high-level construct(s)

There is one easy way to re-write the particular example above, that would be compatible with all versions of C++ for that matter. It consists of taking `for_each` out of the way and replacing it with `count` which is cut out for the job:

```
std::vector<int> numbers = {1, 7, 4, 7, 7, 2, 3, 4};

int count = std::count(numbers.begin(), numbers.end(), 7);
```

Of course this does not mean that you never need functors or lambdas – you do need them. But the message I'm trying to convey is that if you find yourself in need for a state in a functor or a lambda, then you should think twice about the higher-level construct you are using. There is probably one that better fits the problem you are trying to solve.

Let's look at another classical example of state within a callable: **sentinels**.

A sentinel value is a variable that is used for the anticipated termination of an algorithm. For instance, `goOn` is the sentinel in the following code :

```
std::vector<int> numbers = {8, 4, 3, 2, 10, 4, 2, 7, 3};

bool goOn = true;
for (size_t n = 0; n < numbers.size() && goOn; ++n)
{
    if (numbers[n] < 10)
    {
        std::cout << numbers[n] << '\n';
    }
    else
    {
        goOn = false;
    }
}
```

The intention of this code is to print out numbers of the collection while they are smaller than 10, and stop if a 10 is encountered during the traversal.

When refactoring this code in order to benefit from the expressiveness of the STL, one may be tempted to keep the sentinel value as a state in a functor/lambda.

The functor could look like:

```
class PrintUntilTenOrMore
{
public:
    PrintUntilTenOrMore() : goOn_(true) {}

    void operator()(int number)
    {
        if (number < 10 && goOn_)
        {
            std::cout << number << '\n';
        }
        else
        {
            goOn_ = false;
        }
    }

private:
    bool goOn_;
};
```

And at call site:

```
std::vector<int> numbers = {8, 4, 3, 2, 10, 4, 2, 7, 3};
std::for_each(numbers.begin(), numbers.end(),
PrintUntilTenOrMore());
```

The analogous code with a lambda would be:

Click To Expand Code

C++

```
std::vector<int> numbers = {8, 4, 3, 2, 10, 4, 2, 7, 3};

bool goOn = true;
std::for_each(numbers.begin(), numbers.end(), [&goOn](int number)
{
```

```
    if (number < 10 && goOn)
    {
        std::cout << number << '\n';
    }
    else
    {
        goOn = false;
    }
});
```

But these pieces of code have several issues:

- the state `goOn` makes them complex: a reader needs time to mentally work out what is going on with it
- the call site is contradictory: it says that it does something "for each" element, and it also says that it won't go after ten.

There are several ways to fix this. One is to take the test out of the `for_each` by using a `find_if`:

```
auto first10 = std::find_if(numbers.begin(), numbers.end(), [](int number){return number >= 10;});
std::for_each(numbers.begin(), first10, [](int number){std::cout << number << std::endl;} );
```

This works well in this case, but what if we needed to filter based on the result of a transformation, like the application of a function f to a number? That is to say if the initial code was:

```
std::vector<int> numbers = {8, 4, 3, 2, 10, 4, 2, 7, 3};

bool goOn = true;
for (size_t n = 0; n < numbers.size() && goOn; ++n)
{
    int result = f(numbers[n]);
    if (result < 10)
    {
        std::cout << result << '\n';
    }
    else
    {
        goOn = false;
    }
}
```

Then you would want to use `std::transform` instead of `std::for_each`. But in this case the `find_if` would also need to call f on each element, which doesn't make sense because you would apply f twice on each element, once in the `find_if` and once in the `transform`.

A solution here would be to use ranges. The code would then look like:

```
for_each(numbers | transform(f) | take_while(lessThan10),
         [](int number){std::cout << number << std::endl;});
```

Custom comparison, equality and equivalence with the STL

Let's start with the following code excerpt:

```
std::vector< std::pair<int, std::string> > v1 = ... // v1 is filled
with data
std::vector< std::pair<int, std::string> > v2 = ... // v2 is filled
with data
std::vector< std::pair<int, std::string> > results;

std::sort(v1.begin(), v1.end());
std::sort(v2.begin(), v2.end());

std::set_difference(v1.begin(), v1.end(),
                    v2.begin(), v2.end(),
                    std::back_inserter(result),
                    compareFirst);
```

There are 2 sets of data represented by 2 sorted vectors v1 and v2, on which we apply a `std::set_difference` (see Algorithms on sets). This `std::set_difference` writes its output in `results`, with `std::back_inserter` ensuring that all outputs are push_back'd into results.

One particularity though: a custom comparison operator is provided to `std::set_difference`: `compareFirst`.

By default, `std::set_difference` compares the elements with the default comparison on `std::pair` (which compares both the first and second element of the pair), and here with `compareFirst` we want to compare pairs on their first element only. `compareFirst` is not in the STL, so we will try to implement it ourselves.

Before jumping on to the implementation, we already have an interesting take away here. Even if `std::set_difference` expect its input to be sorted, it is possible to use it (or any algorithm on sorted elements) based on a comparator (let's call it C) different from the comparator used for sort, provided that **the elements are also sorted by this comparator** C. In our case for instance, we use a `std::set_difference` that compares pairs by their first elements, although these pairs have been sorted by both their first and

second elements. But since this implies that they are *a fortiori* sorted by first, it is completely OK to do this.

Now let's implement `compareFirst`. A natural, naïve, first-try code would look like:

```
bool compareFirst(const std::pair<int, std::string>& p1, const
std::pair<int, std::string>& p2)
{
    return p1.first == p2.first; // not final code, bug lurking
here!
}
```

Actually, this implementation won't give the expected results at all. But why?? After all, set_difference should check whether a given element is equal to another one in the other collection, right?

The least we can say is that this seems completely unnatural, and the remainder of this post will consist in understanding how we came to this, and why this is in fact completely normal.

To understand this, we must view the STL as roughly divided into 2 parts: the part that operates on SORTED elements, and the part that operates on elements that are NOT SORTED.

The SORTED part of the STL

In this part are associative containters (`std::map`, `std::multimap`, `std::set`, `std::multiset`), because their elements are sorted.

Some algorithms also fall in this category, because they assume the elements they operate on are sorted: `std::set_difference`, `std::includes` or `std::binary_search` for example.

The UNSORTED part of the STL

In this part there are sequence containers (`std::vector`, `std::list`, `std::deque` and `std::string`), because their elements are not necessarily sorted.

And the algorithms that fall into this category are those that don't need their elements to be sorted, like `std::equal`, `std::count` or `std::find` for example.

Comparing elements

There are two ways to express *"a is the same as b"* in C++:

- the natural way: `a == b`. This is called **equality**. Equality is based on **operator==**.
- the other way: a is not smaller than b and b is not smaller than a, so `!(a<b) && !(b<a)`. This is called **equivalence**. Equivalence is based on **operator<**.

Then two questions naturally arise about equivalence.

How is it different from equality?

For simple types like `int`, and actually for most types in practice, equivalence is indeed the same thing as equality. But as pointed by Scott Meyers in Effective STL Item 19, there are some not-too-exotic types where the two are not the same, like case-insensitive strings for example.

Why such a far-fetched way to express a simple thing?

When an algorithm compares elements in a collection, it is easy to understand that there must only be **one way** of comparing them (having several comparators is cumbersome and creates a risk of inconsistency). So a choice needs to be made between comparing based on `operator==` or on `operator<`.

In the SORTED part of the STL, the choice is already made: by definition of sorting, the elements must be comparable with operator< (or a customized (operator<)-like function). The UNSORTED part on the other side does not have this constraint and can use the natural operator==.

Implementing the comparator

The **UNSORTED** part of the STL uses **operator==** to perform comparisons, while the **SORTED** part uses **operator<**. And custom comparison operators must follow this logic.

Now we understand how to implement our custom operator `compareFirst` for `std::set_difference`, which operates on sorted elements:

```
bool compareFirst(const std::pair<int, std::string>& p1, const
std::pair<int, std::string>& p2)
{
    return p1.first < p2.first; // correct, STL-compatible code.
}
```

All of this is crucial to understand in order to use the STL efficiently.

The design of the STL

As a logical part of the STL learning resource, here is how the STL has been designed, and how you can design your components to make them benefit from the power of the STL.

The design of the STL has been driven by the intention of **separating algorithms from data structures**.

Algorithms include:

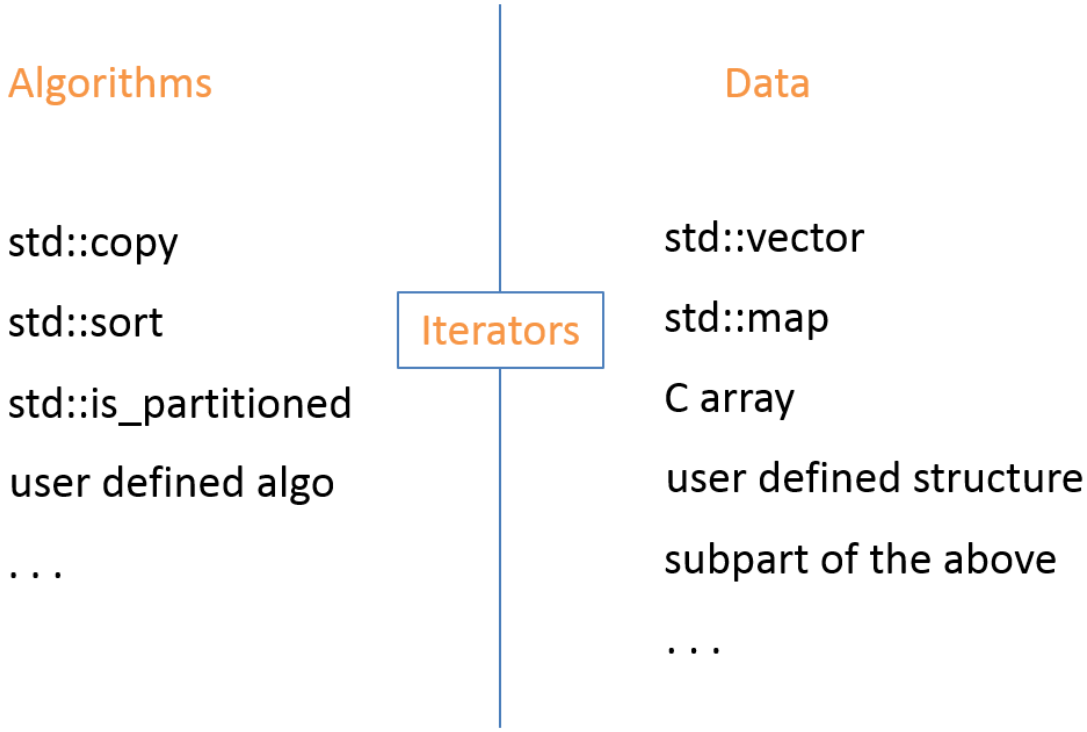
- those in the header <algorithm>,
- those we write when our need cannot be solved by a standard algorithm.

Data include:

- standard STL containers such as std::map and std::vector,
- C arrays,
- user-defined collections,
- any subpart of the above.

Data can even be drawn from a stream as we see in How to split a string in C++.

The intention to **separate algorithms from data structures** has been accomplished with an interface: **iterators**.



In order to benefit from all the advantages offered by the large variety of algorithms, data has to present an **Iterator** interface.

Here we show how to do this for different kinds of data.

STL containers

Iterators can be obtained via:

- begin(), end()
- rbegin(), rend() for reverse order iterators,
- cbegin(), cend() (or simply begin() and end() on const containers) for const iterators,
- crbegin(), crend()(or simply rbegin() and rend() on const containers) for const reverse order iterators.

C arrays

For C arrays, **pointers** play the role of iterators.

```
int myInts[100];

std::for_each(myInts, myInts + 100, doSomething);
```

Even if strictly speaking, `myInts` is not a pointer but an array, it still gives access to the first element of the array, while `myInts + 100` points to the "one-after-the-end" address, which follows the begin-end semantics.

So C arrays can be used with algorithms, which can be of great help in legacy code.

Note that a new uniform syntax has been introduced since C++11, with `std::begin` (and `std::end`) **free functions** (and not class methods). They can be used uniformly on any type showing a `begin` (resp. `end`) method that can be called with no argument, and they can also be used on C arrays.
The following code gives an example of this uniformity:

```
int myInts[100];
std::vector<int> vec(100, 0); // vector of size 100 initialized with
zeros

std::for_each(std::begin(vec), std::end(vec), doSomething);
std::for_each(std::begin(myInts), std::end(myInts), doSomething);
```

This makes C arrays easier to use, and is quite convenient for generic code.

Note that the `std` namespace has to be written explicitly for the C array, because it cannot use ADL, but could be omitted on the vector. More on ADL on in a later post.

User-defined collections

Sometimes we write our own collection that reflect domain needs. Let's take the example of the user-defined FlowCollection class, that represents a collection of financial flows.
Given what we saw above, it needs to publish iterators in order to benefit for algorithms.
How do we do this?

Typedef a standard collection

Every time you want to write a collection, ask yourself if a standard one won't do. This would be **as much code that you don't write**. In quite a few cases a standard collection suffices, and you can put a domain name on it with a typedef. For exemple for our collection of flows:

```
using FlowCollection = std::vector<Flow>;
```

This way you get all the iterators plus all the functionalities of std::vector for free, while having a type with a domain name.

Recycle the standard iterators

If a domain functionality is really necessary for the collection, or if you only want a subpart of what a standard container offers, you may have to define a class that wraps a standard container. In this case, iterators can be implemented with the standard container's iterators:

```
// INTERFACE

class FlowCollection
{
public:
    // ...domain interface...

    // iterators to allow data access
    using const_iterator = std::vector<Flow>::const_iterator;
    const_iterator begin() const;
    const_iterator end() const;

    // iterators to allow data modification
    using iterator = std::vector<Flow>::iterator;
    iterator begin();
    iterator end();

    // other iterators...

private:
    std::vector<Flow> m_flows;
    // ...domain data...
};


// IMPLEMENTATION

FlowCollection::iterator FlowCollection::begin()
{
```

```
    return m_flows.begin();
}
```

Implement your own iterators

If your collection has such a degree of complexity that the two previous techniques won't do, you may have to implement your own iterators. This is more complex to do and outside the scope of this post, and the occasions for such a need should be very rare.

This is where the STL stands in the C++ of today (<= C++17). To have a glimpse of how the STL is shaping for the future (and to see how you can start using it right now), hop over to ranges.