CURRIED OBJECTS IN C++

Fluent {C++}

Curried Objects in C++

Curried objects are like facilitators. They consist in intermediary objects between a caller and a callee, and helps them talk to each other in a smooth way. This ability makes the code simpler and easier to read.

While having seen and used the pattern at various places, the first time I encountered the actual term “Curried object” was in an article from James Noble, which clarified the bigger picture about those friendly little creatures.

An typical example of usage for curried objects is when outputting a line of strings separated by commas. If you’ve ever tried it, you probably encountered the obnoxious problem of the last word that should not be followed by a comma, and that forces us to write annoying bookkeeping code to check whether or not to print the bloody comma.

As we’ll see, curried object can relieve your code from those concerns. But this involves mutable curried objects, which we tackle in Part 2 of the series.

There are other uses for curried objects too, and for now we focus on **constant curried objects**.

Indeed, this series on curried objects contains:

- Curried objects – Part 1: Constant curried objects
- Curried objects – Part 2: Mutable curried objects
- Curried objects – Part 3: Curried objects and the STL

We’ll begin with a simple example and gradually build more elaborate ones. Let’s get more into the details of those little beings that want to make our lives easier.

Constant curried objects

Curried objects are closely related to **functions**. In fact, the word "currying" essentially means **partial application** of a function.

What does that mean in practice?

Imagine that we have a function that takes several (or even too many) parameters, and that you need to call that function multiple times by making only a limited number parameters vary every time.

For instance, consider this function that draws a point at coordinates `x` and `y`, and `z`:

```
void drawAt(float x, float y, float z)
{
    std::cout << x << ',' << y << ',' << z << '\n';
}
```

For the sake of the example, this function only prints out the points coordinates. To simplify the graphics generation in the examples that follow, I will feed the program outputs into MS Excel and generate the associated chart.

Factorizing a common parameter

Let's try out this function to draw each of the four cardinal points in the plane at z=0. We could write:

```
drawAt(1, 0, 0);
drawAt(0, 1, 0);
drawAt(-1, 0, 0);
drawAt(0, -1, 0);
```

But the last parameter doesn't bring any information when reading code here. Indeed, we only work in a plane at z=0, so we think in terms of `x` and `y` only.

We can therefore **partially apply** `drawPoint` by fixing the last argument at 0, which would result into a function that only takes `x` and `y` as parameters. This is called currying, but in practice we can implement it with a familiar lambda:

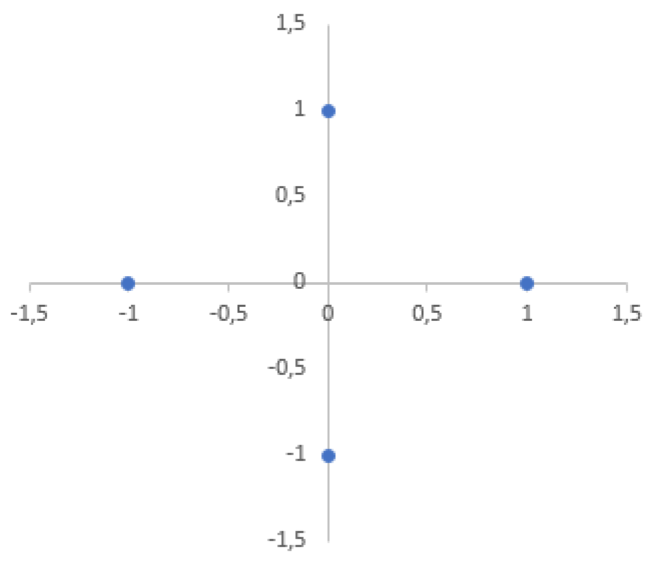
```
auto drawInPlaneAt = [](float x, float y){ drawAt(x, y, 0); };

drawInPlaneAt(1, 0);
drawInPlaneAt(0, 1);
drawInPlaneAt(-1, 0);
drawInPlaneAt(0, -1);
```

No more third coordinate to read about here.

Here is the code outputs:

```
1,0,0
0,1,0
-1,0,0
0,-1,0
```

And the corresponding chart:



Adapting parameters

Not convinced it's worth it? Let's see a slightly more complex example that does not only make a partial application, but also makes an adaptation of parameters (so strictly speaking, this is not only "currying" then).

We now want to draw a line of points identified by a slope and an y-intercept. We can refine our curried object to take a slope and a y-intercept and draw a point on this line, given an abscissa `x`:

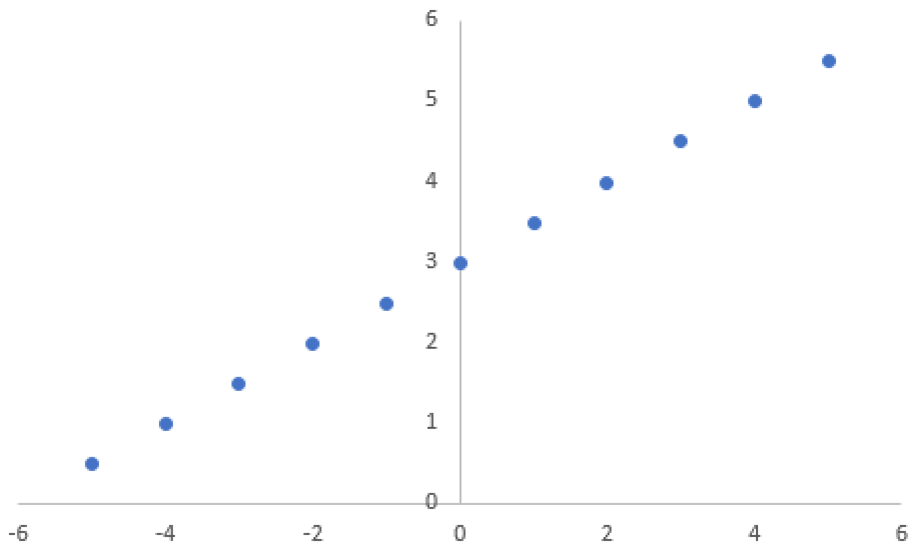
```
#include <iostream>

void drawAt(float x, float y, float z)
{
    std::cout << x << ',' << y << ',' << z << '\n';
}

auto drawOnLine(float slope, float yIntercept)
{
    return [slope, yIntercept](float x) { drawAt(x, slope * x +
yIntercept, 0); };
}

int main()
{
    auto drawOnMyLine = drawOnLine(0.5, 3);
    for (float x = -5; x < 6; x += 1)
    {
        drawOnMyLine(x);
    }
}
```

Note that this code uses C++14's `auto` return type in order to write expressive code with lambdas, but the lambda could be written in C++11 without the intermediary function `drawOnLine`. Or even with a functor in C++98. Those are various ways of writing our curried objects, but the idea remains the same: it is an object that facilitates the dialogue between the caller (here, `main()`) and the callee (here `drawAt`).

Here is the generated output:

```
-5,0.5,0
-4,1,0
-3,1.5,0
-2,2,0
-1,2.5,0
0,3,0
1,3.5,0
2,4,0
3,4.5,0
4,5,0
5,5.5,0
```

And the corresponding graphic:



Let’s now take a more elaborate example: let’s draw a circle!

We now have a `drawInPlane` method that takes an abscissa `x` and an ordinate `y`, and draws a point at that position. But those cartesian coordinates are just one way to identify a position in a plane.

Another representation of the plane is via **polar coordinates**: a distance `r` from an origin and an angle `theta` with the horizontal axis. To draw a circle for example, it is way easier to use polar coordinates than cartesian coordinates.

The curried object that we will create will adapt polar coordinates to cartesian coordinates with the following mathematical formulae:

$$x = r * sin(theta)$$

$$y = r * cos(theta)$$

Let’s now create our curried object that will take a succession of angles and draw a point on the circle for each of those angles:

```
auto drawOnCircle(float xCenter, float yCenter, float radius)
{
    return [xCenter, yCenter, radius](float angle)
    {
        const float xFromCenter = radius * std::sin(angle);
        const float yFromCenter = radius * std::cos(angle);
```

```
            drawInPlaneAt(xCenter + xFromCenter, yCenter + yFromCenter);
        };
}
```

Let’s now use the curried object to generate some points on the circle:

```
auto drawOnMyCircle = drawOnCircle(2, 1, 3);
for (float angle = -3.14; angle < 3.14; angle += 0.2)
{
    drawOnMyCircle(angle);
}
```

As a side note, you may have noticed this particular example is in dry need of strong typing, to be able to write something like that:

```
auto drawOnMyCircle = drawOnCircle(XCenter(2), YCenter(1),
Radius(3));
```

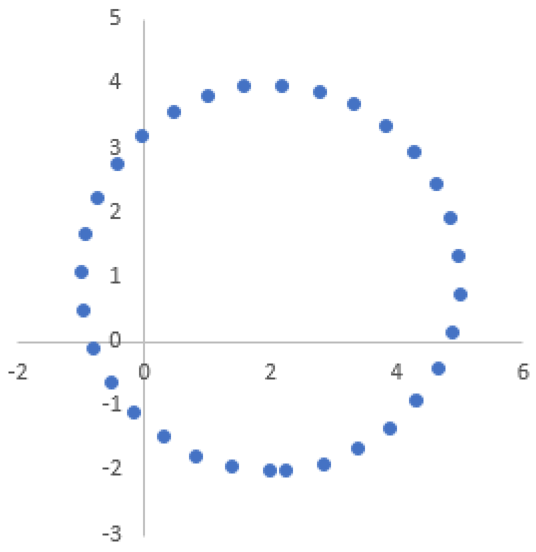
But end of side note, let’s keep the focus on curried objects.

Here is the output of the program:

```
1.99522,-2,0
1.39931,-1.93925,0
0.827346,-1.76132,0
0.302131,-1.47331,0
-0.155395,-1.08669,0
-0.526992,-0.616884,0
-0.797845,-0.0826181,0
-0.957158,0.494808,0
-0.998578,1.09238,0
-0.920453,1.68626,0
-0.7259,2.25278,0
-0.422674,2.76936,0
-0.0228629,3.21541,0
0.457593,3.57313,0
0.99954,3.82826,0
1.58137,3.97065,0
2.17989,3.9946,0
2.77124,3.89917,0
3.33185,3.68816,0
3.83935,3.36998,0
4.27353,2.95731,0
4.61707,2.46662,0
4.85627,1.91745,0
4.98161,1.33171,0
4.98807,0.732742,0
4.87541,0.144431,0
4.64812,-0.40977,0
4.31526,-0.90777,0
```

```
3.89009,-1.32971,0
3.38957,-1.65878,0
2.83366,-1.88184,0
2.2445,-1.99002,0
```

And here is the corresponding graphic:



Isn’t it too much indirection?

Let’s have a look at the code to generate those points, all put together:

```
#include <iostream>
#include <cmath>

void drawAt(float x, float y, float z)
{
    std::cout << x << ',' << y << ',' << z << '\n';
}

void drawInPlaneAt(float x, float y)
{
    drawAt(x, y, 0);
}

auto drawOnCircle(float xCenter, float yCenter, float radius)
{
    return [xCenter, yCenter, radius](float angle)
    {
        const float xFromCenter = radius * std::sin(angle);
        const float yFromCenter = radius * std::cos(angle);
        drawInPlaneAt(xCenter + xFromCenter, yCenter + yFromCenter);
    };
}
```

```
int main()
{
    auto drawOnMyCircle = drawOnCircle(2, 1, 3);
    for (float angle = -3.14; angle < 3.14; angle += 0.2)
    {
        drawOnMyCircle(angle);
    }
}
```

Now let's compare it with an equivalent code, but that doesn't use any curried object:

```
#include <iostream>
#include <cmath>

void drawAt(float x, float y, float z)
{
    std::cout << x << ',' << y << ',' << z << '\n';
}

int main()
{
    for (float angle = -3.14; angle < 3.14; angle += 0.2)
    {
        const float xFromCenter = 3 * std::sin(angle);
        const float yFromCenter = 3 * std::cos(angle);
        drawAt(2 + xFromCenter, 1 + yFromCenter, 0);
    }
}
```

The version with curried objects has more lines of code, and more indirections. Is it a good thing or a bad thing?

By itself, having more lines of code is not a good thing. But to decide if curried objects are worth this investment, let's consider what they brought us:

- **more labels**: if you had first seen the second version of the code above, the one without curried objects, would you have guessed it was drawing a circle? You probably would have, but after how much time? The version with curried objects has more code, but the extra lines **carry information** about the intent of the code. For this reason, I think they are useful.
- **more reuse**: if we want to draw another circle, the function `drawOnCircle` is there to be reused. And if we have several circles to draw, the version with curried objects will end up having *less* lines of code. More importantly, this version removes some

code duplication that the one without curried objects will have if we multiply the circles.

Now I'd be interested to hear you opinion of this. Are curried objects worth it in your opinion?

What is constant in Constant curried objects

You will notice that all those curried objects, that we have implemented as lambdas, have an `operator()` that is `const` (this is the default behaviour of lambdas). They all contain data, but this data is not modified by the application of the curried object.

What happens when the state of the curried object is modifiable? Does it bring any benefit?

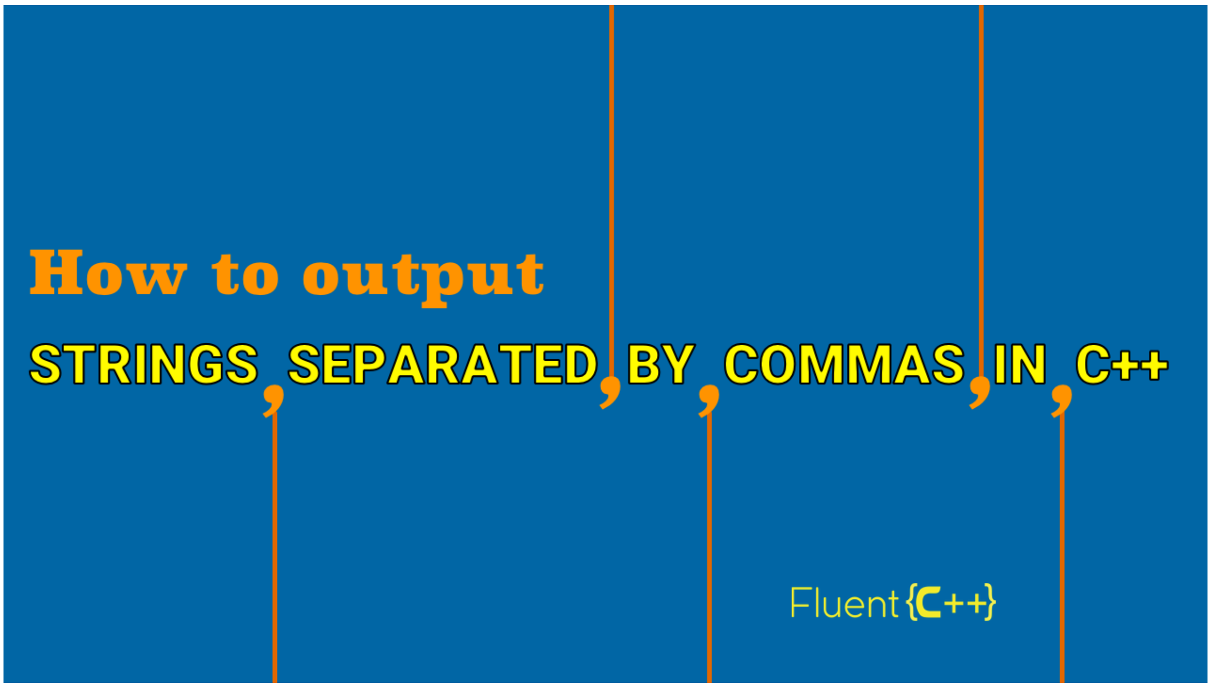
It turns out that it does, and this is what we explore in Part 2 of the series on curried objects in C++.

How to Output Strings Separated by Commas in C++

Every once in a while we all face that problem: how to **output strings separated by commas** (or by any other character), and not have a comma appear after the last one?

Or rather: how to avoid writing a comma after the last string AND **keep the code clean** of the annoying bookkeeping that this little operation needs?

This article will show you how to output several strings separated by commas with little burden on your calling code. Now if you have a **whole collection** (like an STL container for example) of strings to intersperse with comma, the article you want to lok at is Integrating Mutable Curried objects with the STL.



Here we'll also use curried objects. We've seen constant curried objects already, that facilitate the dialog between two interface by storing data and translating an interface. And in the case where we want to output strings separated by commas, there is a dialogue between the main application code, that has the strings, and the component that can output those strings (a `std::ostream` for instance).

But the dialogue is tense. The application code ends up burdened with bookkeeping code to follow what the stream has received already, and whether or not to push the infamous comma.

We can use curried objects, which are facilitators, to simplify the code. But here we'll need more than a constant curried object. We're going to use a **mutable curried object**.

The series on curried object contains:

- Curried objects – Part 1: Constant curried objects
- Curried objects – Part 2: How to Output Strings Separated by Commas in C++ (Mutable curried objects)
- Curried objects – Part 3: Integrating Mutable Curried objects with the STL

Motivating example: tick, tack

Let's create a function that prints a certain number of times "tick" and "tack", interspersed with commas, into an output stream. This output stream could be linked to the console (`std::cout`), a file (`std::ofstream`) or even just a `std::string` (`std::ostringstream`).

A quick and dirty trial could look like this:

```
void printTickTack(std::ostream& output, int numberOfTimes)
{
    for (int i = 0; i < numberOfTimes; ++i)
    {
        output << "tick,tack,";
    }
}
```

It’s quick because it’s short and simple, but it’s dirty because calling the function with `printTickTack(std::cout, 3);` outputs this:

```
tick,tack,tick,tack,tick,tack,
```

Note the trailing comma at the end.

Here is a way to change the code so that it no longer outputs the trailing comma:

```
void printTickTack2(std::ostream& output, int numberOfTimes)
{
    if (numberOfTimes > 0)
    {
        output << "tick,tack";
    }
    for (int i = 0; i < numberOfTimes - 1; ++i)
    {
        output << ",tick,tack";
    }
}
```

Which outputs (with the same calling code):

```
tick,tack,tick,tack,tick,tack
```

The result is correct, but now it’s the code that has become dirty. The spec is very simple yet the application code is burdened with

- an additional if statement,
- two lines of code instead of one that send data to the output,
- a non-trivial breaking clause for the for loop,
- an odd string, `",tick, tack"`, different from the other one `"tick,tack"`, even though the spec does not mention anything about two different strings.

This technical trick makes as much superfluous code in the **main application logic** for a reader to parse. But on the other hand, the **stream** cannot take on this complexity because it is a generic component.

Let's introduce an intermediary object that will help the two talk to each other.

A mutable curried object

Let's change the above code to introduce a parameter: `isFirst`, that is `true` at the first iteration of the loop, and becomes `false` afterwards. With it, the loop knows whether to output a comma before the `"tick, tack"`:

```
void printTickTack(std::ostream& output, int numberOfTimes)
{
    bool isFirst = true;
    for (int i = 0; i < numberOfTimes; ++i)
    {
        if (isFirst)
        {
            isFirst = false;
        }
        else
        {
            output << ',';
        }
        output << "tick,tack";
    }
}
```

Let's try out the code with `printTickTack(std::cout, 3);`:

```
tick,tack,tick,tack,tick,tack
```

The result is still correct but, if anything, the code has become worse than before. Now there is an if statement inside the loop and a boolean variable to keep in mind while reading the application code.

However, we can extract a function out of this code, parametrised with `isFirst` and the string to output:

```
void printSeparatedByComma(std::string const& value, std::ostream&
output, bool& isFirst)
{
    if (isFirst)
    {
        isFirst = false;
    }
    else
    {
        output << ',';
    }
    output << value;
}

void printTickTack(std::ostream& output, int numberOfTimes)
{
    bool isFirst = true;
    for (int i = 0; i < numberOfTimes; ++i)
    {
        printSeparatedByComma("tick,tack", output, isFirst);
    }
}
```

It’s not ideal since `printSeparatedByComma` operates on `isFirst` which is outside of its scope, but on the other hand most of the complexity has gone to that new function.

An interesting consequence is that we can totally remove the comma delimiter from the calling code. Indeed, the following code outputs the same result:

```
void printTickTack(std::ostream& output, int numberOfTimes)
{
    bool isFirst = true;
    for (int i = 0; i < numberOfTimes; ++i)
    {
        printSeparatedByComma("tick", output, isFirst);
        printSeparatedByComma("tack", output, isFirst);
    }
}
```

The calling code looks better, however there are at least two issues left with it:

- it still shows the technical variable `isFirst`,
- the function `printSeparatedByComma` is called several times with the same argument.

To facilitate the dialogue between `printTickTack` and `printSeparatedByComma`, let’s introduce a curried object, that will take care of the two fixed parameters `output` and `isFirst`:

```
class CSVPrinter
{
public:
    explicit CSVPrinter(std::ostream& output) : output_(output),
isFirst_(true) {}

    friend CSVPrinter& operator<<(CSVPrinter& csvPrinter,
std::string const& value)
    {
        if (csvPrinter.isFirst_)
        {
            csvPrinter.isFirst_ = false;
        }
        else
        {
            csvPrinter.output_ << ',';
        }

        csvPrinter.output_ << value;
        return csvPrinter;
    }
private:
    std::ostream& output_;
    bool isFirst_;
};
```

We implement an `operator<<` to give it a stream-like interface.

Now the calling code becomes much simpler:

```
void printTickTack(std::ostream& output, int numberOfTimes)
{
    CSVPrinter csvPrinter{output};
    for (int i = 0; i < numberOfTimes; ++i)
    {
        csvPrinter << "tick";
        csvPrinter << "tack";
    }
}
```

No more bookkeeping in the application code, an not even a trace of a comma any more. We could easily parametrize the `CSVPrinter` to accept another delimiter than a comma.

Discussion

The effect of introducing the curried object has made the calling code nearly as simple as its specification, which is a good thing. This curried object is mutable in the sense that some of its members (here, `isFirst`) are not const and are designed to change in the course of its life.

Now is mutable state a good thing? Indeed, mutable state is at the origin of some bugs when it is not in the state we expect it to be (which is why the functional programming paradigm forbids mutable state). In our case though, the operation itself has some complexity, and it is better off in an encapsulated object with a clear interface rather than as a wart on the main application logic.

Another issue with mutable state is multithreading. Indeed, a shared mutable state is not easy to handle when several threads have access to it. In our case, even if the above component could be modified to be thread-safe (likely at the expense of performance), the above version helps simplifying a local piece of code that needs to build a string separated by commas.

Finding an elegant name

In his paper Arguments and Results, James Noble introduces a mutable curried object with the interface of a word processor, to which a client code can ask to write a piece of text at a given position and with a given font.

A call to the interface (which is in SmallTalk) looks like this:

```
view drawString: 'This is an example' at: origin font: font.
```

The initial problem with this interface is that

- if we want to write several pieces of text with the same font, which is a common case, we have to pass the font every time,
- each time we want to write a piece of text we have to work out the position to write at, and it depends on the words we have written before.

The article proposes to introduce a curried object in much the same vein as our `CSVPrinter`, that takes the font once, and computes every incremental position so that its client code **only has to send it the next piece of text**.

But the beautiful thing about the curried object in James's article is its name: `Pen`.

In three letters, the interface explains its usage in an intuitive manner, by referring to a concept that we already know. To write a word, we pick up a pen, write the word, and put the pen down. Then to write another word, we pick it up again and write the new word. And so on.

Compared to “`Pen`“, the name of our `CSVPrinter` seems pretty crappy now. Isn’t there a concept that our curried object models, and that could provide a better inspiration for its name?

Perhaps one possibility would be to name it `CSVTypewriter`. Indeed, the CSV writer doesn’t work the same way as the word processor. In the word processor, the pen goes to the next line whenever there is more than enough text to fill a line. A CSV line however, can be arbitrarily long: it is only a specific action on the interface that can break it off. Just like a typewriter, where the writer needs to pull a lever to slide the carriage back to the left.

But this could be over the top, and maybe there is a more adapted analogy. As usual, your opinions are welcome.

Anyway, this idea of a typewriter made me realize that, whichever the name of our helper, it would make sense to add to it a method to go to the next line:

```
#include <iostream>

class CSVPrinter
{
public:
    void nextLine()
    {
        output_ << '\n';
        isFirst_ = true;
    }

    // ...
};
```

Here is a full code example that uses this methods along with the others:

```
Click To Expand Code
```

```
C++
```

```
#include <iostream>

class CSVPrinter
```

```
{
public:
    explicit CSVPrinter(std::ostream& output) : output_(output),
isFirst_(true) {}
    void nextLine()
    {
        output_ << '\n';
        isFirst_ = true;
    }

    friend CSVPrinter& operator<<(CSVPrinter& csvPrinter,
std::string const& value)
    {
        if (csvPrinter.isFirst_)
        {
            csvPrinter.isFirst_ = false;
        }
        else
        {
            csvPrinter.output_ << ',';
        }

        csvPrinter.output_ << value;
        return csvPrinter;
    }
private:
    std::ostream& output_;
    bool isFirst_;
};

void printTickTack(CSVPrinter& csvPrinter, int numberOfTimes)
{
    for (int i = 0; i < numberOfTimes; ++i)
    {
        csvPrinter << "tick";
        csvPrinter << "tack";
    }
}

int main()
{
    CSVPrinter csvPrinter{std::cout};

    printTickTack(csvPrinter, 3);
    csvPrinter.nextLine();
    printTickTack(csvPrinter, 4);
}
```

And this code outputs:

```
tick,tack,tick,tack,tick,tack

tick,tack,tick,tack,tick,tack,tick,tack
```

Can a STL algorithm send data to a curried object?

The loop we have used here to demonstrate the concept of a mutable curried object was very simple.

What if we had more complex loops over collections, such as those in the STL algorithms? How do we integrate curried objects with them?

Stay tuned, as this is the topic of the 3rd episode in our series on Curried objects coming up!

How to Send an STL Collection to a Curried Object

After seeing how to send individual objects to a curried object, let's see how we can haul a whole collection into one of those curried creatures. One use case for this is to **intersperse a collection of strings with commas**.

If you're jumping in the topic of curried objects just now, a curried object is an intermediary object that facilitates the dialogue between a caller and a callee by fixing some parameters and potentially adapting other parameters.

We've seen that those objects can carry some logic that ends up simplifying the application code. If you want to catch up on the previous episodes, the series on curried objects contains:

- Part 1: Constant curried objects
- Part 2: How to Output Strings Separated by Commas in C++ (Mutable curried objects)
- Part 3: Sending an STL Collection to a Curried Object

Motivating example

Let's pick up our curried object `CSVPrinter`. It accepts successive strings and sends them to a stream by alternating them with commas, and makes sure not to write a trailing comma at the end of the stream.

Here is the implementation of `CSVPrinter`:

```
#include <iostream>

class CSVPrinter
{
public:
    explicit CSVPrinter(std::ostream& output) : output_(output),
isFirst_(true) {}

    friend CSVPrinter& operator<<(CSVPrinter& csvPrinter,
std::string const& value)
    {
        if (csvPrinter.isFirst_)
        {
            csvPrinter.isFirst_ = false;
        }
        else
        {
            csvPrinter.output_ << ',';
        }

        csvPrinter.output_ << value;
        return csvPrinter;
    }
private:
    std::ostream& output_;
    bool isFirst_;
};
```

Note that this is just one particular case of curried object. Curried object don't have to have an `operator<<`.

Here is some calling code to exerce it:

```
CSVPrinter csvPrinter{std::cout};

csvPrinter << "tick";
csvPrinter << "tack";
csvPrinter << "toe";
```

This code outputs:

```
tick,tack,toe
```

Now let's take a collection of strings:

```
static std::vector<std::string> getSentence()
{
    return {"And", "then", "there", "were", "none"};
}
```

And let's send all the objects of this collection to our curried object. In our case, it will print them by interspersing them with commas:

```
CSVPrinter csvPrinter{std::cout};

auto sentence = getSentence();

for (auto const& word : sentence)
{
    csvPrinter << word;
}
```

Indeed, the following code outputs:

```
And,then,there,were,none
```

Now we have a very simple operation, that has a very simple code and that's all well.

So what the point of going further?

It is to **integrate curried objects with STL algorithms**, to let the algorithms send their outputs into a curried object.

Not that it would be useful in this particular case, because the code is so simple here. But working on such a basic case will let us focus on the integration of the curried object with STL algorithms in general (to easily intersperse their outputs with commas, for one example).

So let's get into this.

First (bad) attempt: using a function object

To turn this piece of code into an algorithm call:

```
for (auto const& word : sentence)
{
    csvPrinter << word;
}
```

A intuitive option could be to use `std::for_each`:

```
auto sendToCsvPrinter = [&csvPrinter](std::string const& word)
{csvPrinter << word;};

std::for_each(begin(sentence), end(sentence), sendToCsvPrinter);
```

Granted, this may not be an improvement to the code because it was so simple, but we're just studying how to connect an algorithm with a curried object in a simple case.

Let's run the code:

```
And,then,there,were,none
```

The result is correct. But is this the right way to integrate the curried object with the STL? Can we generalize it to other algorithms than `for_each`?

The answer is **No**, for at least two reasons. One is that all algorithms don't take a function object, to begin with. Take `set_difference`, or `partial_sum`, or `rotate_copy` for example.

The other reason is that even for the algorithms that do take a function object, such as `std::transform` for instance, some don't guarantee that they will traverse the input range **in order**. So the algorithm may call the function object in any order and send the result to our curried object in an order different from the input, that could lead to, for example:

```
then,none,there,were,And
```

`std::for_each` guarantees to traverse the input collection in order tough.

Note that in general, carrying a mutable state inside of a function object can lead to incorrect results because most algorithms are allowed to make internal copies of the function object (`std::for_each` guarantees that it won't, though). This leads to the mutable state being located in different object, that could lose consistency with each other (this is why in the STL function objects, stateless is stressless). However, here we don't have this problem since the function object only has a **reference** to the state, and not the state itself.

Anyway, for the above two reasons using a function object to connect an STL algorithm to a curried object is not a good idea.

So what to do then?

A better solution: using the output iterator

Going back to our initial code:

```
for (auto const& word : sentence)
{
    csvPrinter << word;
}
```

Another way to see the situation is that we're sending data to the `CSVPrinter`, or said differently, that we're **copying** data from the `sentence` container over to the `CSVPrinter`. So instead of `std::for_each`, we could use `std::copy`.

But then, we need something to make `std::copy` send the data to the curried object. `std::copy` uses an output iterator to emit its output data. So we need an custom output iterator that we could customize and plug to `CSVPrinter`.

A custom inserter? Let's use `custom_inserter`!

As a reminder, the definition of `custom_inserter` looked like this:

```
template<typename OutputInsertFunction>
class custom_insert_iterator
{
public:
    using iterator_category = std::output_iterator_tag;
    using value_type = void;
    using difference_type = void;
    using pointer = void;
    using reference = void;

    explicit custom_insert_iterator(OutputInsertFunction
insertFunction) : insertFunction_(insertFunction) {}
    custom_insert_iterator& operator++(){ return *this; }
    custom_insert_iterator& operator*(){ return *this; }
    template<typename T>
    custom_insert_iterator& operator=(T const& value)
    {
        insertFunction_(value);
        return *this;
    }
private:
    OutputInsertFunction insertFunction_;
};

template <typename OutputInsertFunction>
custom_insert_iterator<OutputInsertFunction>
custom_inserter(OutputInsertFunction insertFunction)
{
    return
custom_insert_iterator<OutputInsertFunction>(insertFunction);
}
```

The most important part in `custom_inserter` is this:

```
custom_insert_iterator& operator=(T const& value)
    {
        insertFunction_(value);
        return *this;
    }
```

It is an iterator that, when an algorithm sends data to it, passes on this data to a custom function (`insertFunction_` in the above code).

Here is how `custom_inserter` can help us connect `std::copy` to our curried object `CSVPrinter`:

```
auto sendToCsvPrinter = custom_inserter([&csvPrinter](std::string
const& word) {csvPrinter << word;});
std::copy(begin(sentence), end(sentence), sendToCsvPrinter);
```

which outputs:

```
And,then,there,were,none
```

We had encountered `custom_inserter` when making STL algorithms output to legacy collections, but we see here another usage: **outputting to a curried object**.

In a more elaborate operation on a collection, such as `std::set_difference` for example, we can use `custom_inserter` to send the output of the algorithm to the curried object in a similar way:

```
std::set_difference(begin(set1), end(set1), begin(set2), end (set2),
sendToCsvPrinter);
```

Using the channel of the output iterators doesn't suffer from the two issues that we raised when attempting to use the function object. Indeed:

- all the algorithms that output a range do have one (or more) output iterators,
- even if some algorithms don't treat the input in order, they all send data to the output in order.

It's not just about interspersing with commas

All the way through this example, we've used a curried object to intersperse the words of a sentence with commas. Note that this curried object wrapped into an output iterators is in the technical specification for the future standard library under the name of `std::ostream_joiner`. But this is just one specific case of curried objects, and there are other ways than curried objects to fill this need.

As a side note, the most elegant way I know of to intersperse a sentence with commas is by using the range-v3 library:

```
#include <iostream>
#include <string>
#include <vector>
#include <range/v3/to_container.hpp>
#include <range/v3/view/intersperse.hpp>

int main()
{
    std::vector<std::string> numbers = {"And", "then", "there",
"were", "none"};
    std::vector<std::string> results = numbers |
ranges::view::intersperse(",") | ranges::to_vector;

    for (auto const& result : results) std::cout << result;
}
```

Which outputs:

```
And,then,there,were,none
```

Isn't it beautiful? However if you don't have range-v3 available, a curried object is a nice way to do the job, in my opinion.

Conversely, curried objects can be used for so much more. They make application code (and therefore, life) easier to read and write and, as an icing on the cake they can be integrated with the STL by using smart output iterators.

That's it for our series on curried objects. Your reactions are, as usual, welcome.