Hybrid Programming Languages — You Are Probably Using One

Daniel Santos Follow
May 7, 2017 · 3 min read ★

Personally, I am a fan of hybrid cars because I don't feel like going 100% electric yet. Similarly, I am a fan of hybrid programming languages, also known as multi-paradigm programming languages.

Although there are multiple programming paradigms, to keep it simple, I'm only going to talk about functional and objected-oriented. If we go back to the car ideology,

functional are electric cars and object-oriented are regular gas cars. We know that cars can run on water, natural gas, and who knows what else, but let's just take the mainstream ones. Gas and electric; functional and object-oriented.

Just to mention a few; Python, Java, C++11, and Swift are hybrid programming languages. Using these languages, one can design programs with a functional or object-oriented approach in mind. If you want to learn more about functional programming, check this post. Below is a C++ example of two common way of going through containers, a functional way and using a loop:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>


int main() {
    std::vector<int> v = {1, 2, 3, 4, 6, 7};

    std::for_each(v.rbegin(), v.rend(), [] (int& num) {
      num += 1;
    });

    for(auto num : v) {
      std::cout << num << std::endl;
    }

    return 0;
}
```

FunctionalC++.cpp hosted with ♡ by GitHub view raw

In the code, we declared a vector and initialized it with a series of numbers using an initializer list, since C++11. Later, we applied some operations on those numbers inside the vector. From the 'algorithm' library, we can use the '*std::for_each*' function that takes a section of a container to apply the lambda that we pass to it. We passed a **reference** to the beginning and end of the vector. Something that is easy to overlook is '*rbegin()*' and '*rend()*'. These methods return a **reference iterator** and what that means is that any mutation to the value that the iterator is holding will affect the actual value inside the vector, it is not a copy. This breaks one of the fundamental rules of functional languages,

that is, to avoid mutation. Remember, we are hybrid, we can break one or two rules. But for the curious, without mutation:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>


int main() {
    std::vector<int> v = {1, 2, 3, 4, 6, 7};

    std::vector<int> newVector;

    std::for_each(v.begin(), v.end(), [&newVector] (int num) {
      num += 1; // No change in 'v'
      newVector.push_back(num );
    });

    std::cout << "New old:" << std::endl;
    for(auto num : v) {
      std::cout << num << std::endl;
    }

    std::cout << "New vector:" << std::endl;
    for(auto num : newVector) {
      std::cout << num << std::endl;
    }

    return 0;
}

/* OUTPUT:
New old:
1
2
3
4
6
7
New vector:
2
3
4
5
```

```
42  7
43  8
44  * */
```

FunctionalC++NoMutation.cpp hosted with ♡ by GitHub view raw

Here we have to major changes; '*v.begin()*' and '*v.end()*', and '*&newVector*'. We don't need a reference to the values in the vector, we call the methods that return a copy iterator. Be careful with those because it might hurt your performance if you have a large amount of data in a container. '*&newVector*' is a 'lambda-capture', check syntax here, which is basically capturing a reference to 'newVector' inside the lambda, so we can push the numbers to the vector in the outer scope. The output we get is an untouched 'v' and a filled 'newVector'. The output shows the values in 'v' are the same although we mutated '*num*' inside the lambda, '*num += 1*'.

We just saw some high-order function concept in action, we passed a function to a function. Now can I return a function from a function? Check this example:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  inline std::function<void()> makeReporter(std::vector<int>& vector,
7                                std::function<void(const int)> printer) {
8    return [&] () {
9      std::for_each(vector.cbegin(), vector.cend(), printer);
10   };
11 }
12
13 int main() {
14     std::vector<int> vector {1, 2, 3, 4};
15     auto report = makeReporter(vector, [](const int num) {
16       std::cout << "************** "
17                 << "The number is: "
18                 << num
19                 << " **************"
20                 << std::endl;
21     });
22
23     report();
```

```
    return 0;
}
/* OUTPUT:
************** The number is: 1 **************
************** The number is: 2 **************
************** The number is: 3 **************
************** The number is: 4 **************
*/
```

HighOrderFunctionCpp.cpp hosted with ♡ by GitHub view raw

If you don't understand what this code those, please check my post on functional programming languages. In summary, '*makeReporter*' is a function that returns a function at runtime. I believe this is a clear example of a hybrid programming language like C++. Inside '*makeReporter*', we are calling a **method**, which is an objected-oriented concept, and we return a **function** from a function, which is a functional concept.

These examples show the power of combining these two powerful ideas. Trust me, my examples are very basic and trivial compared to real applications. I am just trying to transfer these ideas. Nowadays many robust and popular frameworks relied on the power of these two ideas, functional and object-oriented. Just to mention a few, iOS (UIKit, etc), React, Express, Vapor, Qt, and many others.

Have fun discovering the power of hybrid programming languages and happy coding!

Functional Programming Programming Hybrid Object Oriented Cpp