Compare Functional Programming in Golang and C++

Florian Bayeff-Filloff

Introduction

This article gives a comparison between C++ and Go(lang), primarily focusing on the functional programming style in these two languages. It covers important aspects of both languages concerning functional programming:

- Typing
- Lambda Calculus
- Functional Composition
- High-Order Functions

But first it gives information about what the programming language C++ is and how it is different from Golang.

The Programming Language C++

C++ was developed by Bjarne Stroustrup in 1979 as the programming language "C with classes" shortly after his Ph.D. thesis. He based it on the programming language C and it was meant as a superset for C. His primary goal was to add object-oriented programming (oop) concepts, like classes, into the existing C language. His language included classes, basic inheritance, inlining, default function arguments and strong type checking in addition to all features C already offered at the time.

In 1983 the name of the language was changed to C++. Also, many new features were added to the language like function overloading, references with the '&' symbol, the const keyword and single-line comments with two forward slashes.

In 1985 C++ was first published under Stroustrup's reference 'The C++ Programming Language'.

In 1998 C++ was first standardized by the ISO and IEC as ISO/IEC 14882:1998, also known as C+98. The standard committee release new standards for C+ after that in a three-year cycle, adding and removing features from release to release.

In 2011 the C++11 standard introduced regular expressions support, a new time library, a standard threading library, the 'auto' keyword and a lot more features.

The last release was C++17 in December 2017.

The Programming Language Go(lang)

Go or Golang is a programming language which was created by a team at Google in 2009. The purpose of GO is to be a system programming language to address current unfulfilled requirements that came up and other languages don't support enough. It also is a combination of object-oriented, functional programming and some other concepts.

With Golang Google made a programming language which should handle the new challenges that

we have now, like networking, client/server focused approaches, usage of multi-core CPUs, without much trouble as it was designed for them.

Characteristics of C++

This section of the article shows some characteristics C++ has that differ or set it apart from Go(lang). Not all features of C++ are shown in this section because of their massive number fills multiple books easily.

Typing Concept and auto

Both C++ and Go use a static typing concept. You can't assign different types to an already known variable, without causing problems.

Go and C++ both allow multi assignments in on line. But Go uses the ':=' for assigning values rather than '=' like C++ and other programming languages. Also Go sets the typing behind the names of variables, classes or functions, C++ like other languages writes it before declaring the name.

```
// Go
a int             // not necessary in Go
a, b := 1, 2

// C++
int a, b = 1, 2;
auto a, b = 1, 2;
```

Another difference is that Go knows what type a and b are in this example without declaring the type explicitly. In C++ the type of a value is normally declared directly from the beginning. Although he same behavior can be achieved since C++11 with the 'auto' keyword. This keyword tells the compiler to figure out the needed type while compiling rather than before hand. This is later important for generic written functions and lambdas.

Importing libraries and header files

In C++ and Go static linking of libraries used. In addition C++ uses header files like C. These are linked by the keyword '#include' followed by the file name. If a header file is part of the standard library it's content is accessed by std::<name> otherwise the file's name is used. With this source code files can also be imported but isn't recommended to do.

```
// Go
import "fmt"
fmt.Println("Hallo World")

// C++
#include <iostream>
std::cout << "Hallo Welt" << std::endl;
```

Pointers / References / Call by Value

C++ offers Pointers and all functionalities which come with them like in C. But generally these are avoided by programmers because of there unsafe behaviors. Instead Smart Pointers should be used. These offer extra features like memory and ownership control.

By default, non-pointer arguments are passed by value. This means the argument's value is copied into the corresponding function parameter and can be used without changing the argument's value outside of the function.

Classes in C++

Classes in Go are detailed in other articles. In C++ classes have a typical appearance:

```
class Point {
public:
  type element/function1();
  type element/function2();
  // more
private:
  type attribute1;
  type attribute2;
  // more
};
```

Data and functions belonging together are combined inside the class. It has a name directly declared after the 'class' keyword, public and private class members. These are declared under the 'public:' or 'private:' keyword and can contain functions, constants or variables. Normally like other program structures classes are decelerated as prototypes in <file_name.h> header files and defined in <file_name.cpp> source files which include the corresponding header file. The '::' symbol is used to access class or struct public members.

Rule of zero, three, five

In C++ we have the rule of zero, three and five, respectably. These three rules declare what parts of a class have to be implemented by the programmer by hand. The compiler doesn't do these automatically if one part mentioned in one rule is already present.

If the class has a copy constructor implemented, a copy destructor and a copy assignment operator have to be declared by hand as well and vice versa. This is called the rule of three.
If also a move constructor is manually declared a move destructor has to be declared as well. This is called the rule of five.
If none of the before mentioned components is declared manually the compiler does these automatically. This is called the rule of zero.

Lambda function objects

Since C++11 lambda function objects allow unnamed, anonyms functions to be declared. These consist of multiple parts, starting with square brackets indicating how the arguments are being used inside the lambda function object. After that are the functions arguments in round and the instructional body in curly brackets.

```
auto f = [](){}
```

Inside the square brackets is defined how values of the function object surroundings are used inside the instructional body. Nothing means no surrounding values can be used. If surrounding values should be used inside the instructional body '=', '&' or 'this' must be used. They indicate that the values are used as constant copies, mutable references or references for all visible values inside a class. Also the arguments can be initialized in here, giving them another value inside the function object other to the outside.

Data files and streams

In C++ all transfers of data are called streams. On there lowest level these already just bytes pushed from one place to another and only get interpreted on higher levels. By being all the same all streams can be combined and chained together by the '<<' symbol for input and '>>' for output out of the left side to the right side of the symbol.

```
#include<iostream>
std::cout << "Hallo World" << std::endl;
```

In the example the byte stream for "Hallo World" is sent into std:cout followed by the byte stream for the end line symbol std::endl. Std::cout is the standard output stream object. The <fstream> header file allows streams to be redirected into or out of data files after opening them inside the program. making read and write operations easy.

Functional Programming in C++ and Golang

Functional programming in its core has some commonly accepted principles. These are expanded or reduced depending on who you ask what functional programming is.

According to the haskell.org wiki entry on functional programming functions are first-class citizens, meaning you can use them as values and pass them around like integers, strings or any other data

type.
Only pure functions should be used, these are functions where the result is only determined by its input values. They always returning the same result for the same input.
The use of mutable states should be avoided at any point, meaning no object is changed after its first initialization. This results in having no unnoticed side effects by changing the value of data around. This principle also means no flow control structures like loops are not allowed, because of their usage of an iterator variable to continue the loop.

Lambda Calculus

As functional programming is a style by which all functions are meant to be built like mathematical expressions the **Lambda Calculus** concept is a widely used pattern in functional programming. The example shows how the basic logical building blocks **True**, **False** and **Not** can be implemented in a functional programming style in Golang and C++ by using the Lambda Calculus.

In Golang a Lambda Calculus expression can be done by defining a new recursive function type that can take other functions as arguments of the same type. The logical building blocks are than defined by functions of the newly defined type, returning anonyms functions themselves. After that they can be used like a Lambda Calculus, chaining the expressions together one after another.

```
// new functions type fnf
type fnf func(fnf) fnf

// example functions to be called by the conditional control structure
ID := 3
f:= func(x fnf) fnf {...}
g:= func(y fnf) fnf {...}

// Lambda Calculus Boolean expressions
True := func(x fnf) fnf { return func(y fnf) fnf { return x } }
False := func(x fnf) fnf { return func(y fnf) fnf { return y } }
Not := func(b fnf) fnf { return b(False)(True) }

// select and call first function f(ID)
False(False)(True)(f)(g)(ID)     // --> f(3)
Not(False)(f)(g)(ID)             // --> f(3)
```

In C++ the same logical building blocks can be implemented by using lambda function objects that return new lambda function objects. For this to work correctly the keyword 'auto' must be used. We don’t know the data type of the functions passed into our logical building block yet.

```
int ID = 3;
auto f(x){...};
auto g(x){...};

auto True = [](auto x) { return [=](auto y) { return x; }; };
auto False = [](auto x) { return [=](auto y) { return y; }; };
auto Not = [=](auto x) {return x(False)(True); };

False(False)(True)(f)(g)(ID);    // --> f(3)
Not(False)(f)(g)(ID);            // --> f(3)
```

Functional Composition

To perform more complicated tasks in a functional programming style, functions have to be composable to new functions. This can be done by taking the first functions return value as an argument for the second function and enclose this expression inside an new function. The new function returns the combined return value of both functions used. Most programming languages support this pattern by default.

In Go this can be easily done by declaring one function with the same return type as the argument of another function. It is than used as an argument of the second function. This combination of functions is contained as a return value of a third function.

```
// functions to be composed together
f := func(x int) int { return x * x }
g := func(x int) int { return x + 1 }

// two functions composed to one function
gf := func(x int) int { return g(f(x)) }

fmt.Printf("%v\n", gf(2)) // --> 5
```

Exactly the same can be done in C++ with normal functions in the same pattern as described.

```
// functions to be composed together
int f(int x){ return x*x; }
int g(int x){ return x+1; }

// two functions composed to one function
int gf(int x){ return g(f(x)); }

std::cout << gf(2) << "\n"; // --> 5
```

Another type of functional composition is using a function as a direct argument for another function returning the combined functions as a value afterwards. In Go this is done by declaring two new types 'function' and 'any' to be used as the functions argument types allowing functions to

be combined with the same interface type.

```
type any interface{}
type function func(any) any

compose := func(g, f function) function {
  return func(a any) any {
    return g(f(x))
  }
}

square := func(x any) any { return x.(int) * x.(int) }

fmt.Printf("%v\n", compose(square, square)(2))                    // --> 16
fmt.Printf("%v\n", compose(compose(square, square), square)(2))   // --> 256
```

In C++ the same can be done by using a lambda function object that takes two arguments and returns a function object combining the two functions into one function.

```
auto square(int x) { return x * x; }

// lambda function object for composing two functions f and g
auto compose = [](auto f, auto g) {
    return [=](auto x) { return f(g(x)); };
};

std::cout << compose(square, square)(2) << "\n";                  // --> 16
std::cout << compose(compose(square, square), square)(2) << "\n"; // --> 256
```

These compositions can also be chained together like seen in the second output line.

High-order functions

In functional programming high-order functions are functions that can take other functions as their arguments and can return functions as return values. Since C++ 11 lambda function objects allow this pattern to be used more easily in C++.

The two classic high-order function families according to the haskell.org wiki entry for functional programming are map and fold. Each programming language supporting a functional programming style should offer these two functions or a way to implement them manually.

Map functions

The map function family in functional programming takes a function and a sequence of data as its arguments, applying the function to all elements of the sequence and returns a new sequence as its result value.

In Golang this is implemented in the Go Standard Library's function strings.Map for strings. It

returns a copy of the argument string with all its characters modified according to the argument function mapping.

```
func Map(mapping func(rune) rune, s string) strings

s := "Hello, world!"
s = strings.Map(func(r rune) rune {
  return r + 1
}, s)
fmt.Println(s) // --> Ifmnp-!xpsme"
```

For other data types the Map function has to be manually implemented as no generic function exists inside the standard library.

In C++ this pattern is equal to the standard library function std::transform, found in the <algorithm> header. It takes the start and end point of a sequence of data, the start point for the result sequence and a function object as arguments, applying the function object on all elements of the sequence, returning the transformed sequence inside the result argument. No return value is used by this function.
Noting here the transformed sequence must contain the same data type as the original sequence, as vectors and lists in C++ are type specific.

```
std::string v = "Hallo, world!";
std::transform(v.begin(), v.end(), v.begin(),
    [](int x) { return x + 1; }
);
std::cout << v << "\n"; // --> Ifmnp-!xpsme"
```

Fold / Reduce functions

Fold or Reduce functions in functional programming are a family of high-order functions that process a data structure e.g. a list in some order and build a return value out of its content. Like the map functions, these take a function and a sequence of data as arguments but return a single value as a result.

In Golang these functions are not directly implemented but can be easily done by defining them manually.
The defined function takes a function of the self defined accumulate type as an argument and applies it to a *SliceStream object which represents the data sequence. Inside the function the argument's function is applied to every element of the sequence. Resulting in a single value that is returned at the end.

```
func (s *SliceStream) Reduce(accumulate Accumulator) Any {
    var result interface{}
    for i, e := range s.data {
        if i == 0 {
            result = e
        } else {
            result = accumulate(result, s.data[i])
        }
    }
    return result
}
```

In C++ this is already implemented in the accumulate() function of the <numeric> header. It takes the start and end points of a data sequence and a starting value for the result as arguments. It then reduces the sequence by adding up all values inside the sequence by default using the '+' operator. Alternatively it can also be given a lambda function object as an optional argument telling it how to accumulate the sequence's elements together.

```
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::vector<std::string> s {"a", "b", "asdf", "Hallo", "World"};

  //sum ist 55
  auto sum = std::accumulate(v.begin(), v.end(), 0);

  // s2 = "abasdfHalloWelt"
  auto s2 = std::accumulate(s.begin(), s.end(), std::string(""));

  //s3 = "12345678910"
  auto s3 = std::accumulate(z.begin(), z.end(), std::string(""),
      [](std::string ret, int i) { return ret + std::to_string(i); }
  );
```

In the example s1 result in the sum of all numbers contained inside the vector starting with 0. S2 und s3 are given an empty string object as a starting point causing both to combine the vector values into one long string. For s3 to work like that we have to tell it to convert the integer values into strings before accumulating them together.

Conclusion

Golang and C++ are both no pure functional programming languages, but both offer programmers possibilities to implement code in a functional programming style.

For C++ it is a lot easier since 2011 because lambda function objects were standardized and allow most basic functional programming patterns to be implemented easily. Also, a lot of functional algorithms are already implemented inside the standard library like transform, reduce and accumulate.

Golang isn't as far as C++ what already existing functional programming patterns concerns but it allows functional patterns to be implemented by defining new function types and using them like normal data types. Other patterns like map or fold have to be implemented manually. Go lets use do that in a straight forward way without being to complicated.

Bibliography

Books

- [TPC] Prof. Dr. Breymann Ulrich. 'Der C++ Programmierer - Aktuell zu C\++17'. Carl Hanser Verlag 2018. ISBN 978-3-446-44884-1.

Web

- [HoC] 'History of C\++'. http://www.cplusplus.com/info/history/. Last visited: 27.12.2018
- [FP11] Grimm Rainer. 'Functional Programming in C++ 11'. science + computing ag. https://www.grimm-jaud.de/images/stories/pdfs/FunctionalProgrammingInC++11.pdf. Last visited: 31.12.2018
- [FPGN] Gigante Nicola. 'Functional Programming in C++ - Nicola Gigante - Meeting C++ 2015'. https://www.youtube.com/watch?v=SCC23W3CQc8. Last visited: 31.12.2018
- [CoPL] Weigend Johannes. 'Master Course: Concepts of Programming Languages - University of Applied Sciences Rosenheim (TH Rosenheim)'. https://github.com/jweigend/concepts-of-programming-languages. Last visited: 27.12.2018
- [HFP] 'Functional programming'. https://wiki.haskell.org/Functional_programming. Last visited 03.01.2019
- [Go] 'The Go Programming Language'. https://golang.org. last visited 03.01.2019