Go Research Paper

Alissa Ronca, Yousuf Abdullah, Eileen Bohen

Abstract

This research will provide a thorough overview of the Go language, with emphasis on the design features, comparison to the C language, inheritance capabilities, and use in game development.

Contents

I. Introduction

A. History of Go

The Go language was developed at Google by Robert Griesemer, Rob Pike and Ken Thompson, who began sketching the goals of the language in September of 2007. The creators felt frustrated with existing languages, specifically for the work being done at Google. In November of 2009, it was made public and open source (FAQ).

The main goal of Go was to address the tradeoff programmers were making between efficient compilation, efficient execution, and writability. Often, programmers would choose ease over safety and efficiency, choosing dynamically typed languages like Javascript and Python. This report will look into some of the interesting design decisions made in order to avoid this tradeoff (FAQ).

B. Design of Go

In Rob Pike's publication, he explains that Go was designed to improve the lives of software engineers and their coworkers. This meant that the language had to work at scale, for large programs with large numbers of dependencies and many programmers working on them. They also set out to create a language that was similar to C, as many Google programmers are familiar with C-like languages. Lastly, the creators wanted to make a modern language, with built-in support for concurrency (Pike).

1. Syntax

As part of the effort to design a language with readability in mind, the creators of Go made a series of decisions in implementing clear syntax. The language has only 25 keywords, much fewer than most modern languages (Pike).

2. Naming

Instead of using keywords like “public” and “private” to determine the visibility of an identifier, Go variables that start with a capital letter are public, whereas lowercase letters denote a variable only visible in that package. This rule applies to variables, types, functions, and everything else (Pike).

3. Semantics

Though this will be explored in depth later on, Go semantics are generally C-like, as the language was designed to be easy for C programmers to learn. There are a number of important differences, however, such as the lack of pointer arithmetic and implicit numeric conversions (Pike).

4. Concurrency

The work done at Google is highly reliant on concurrency, but programmers found that Java and C lacked proper support for it (Pike).

5. Garbage Collection

Go has no memory freeing operation, making garbage collection necessary (Pike).

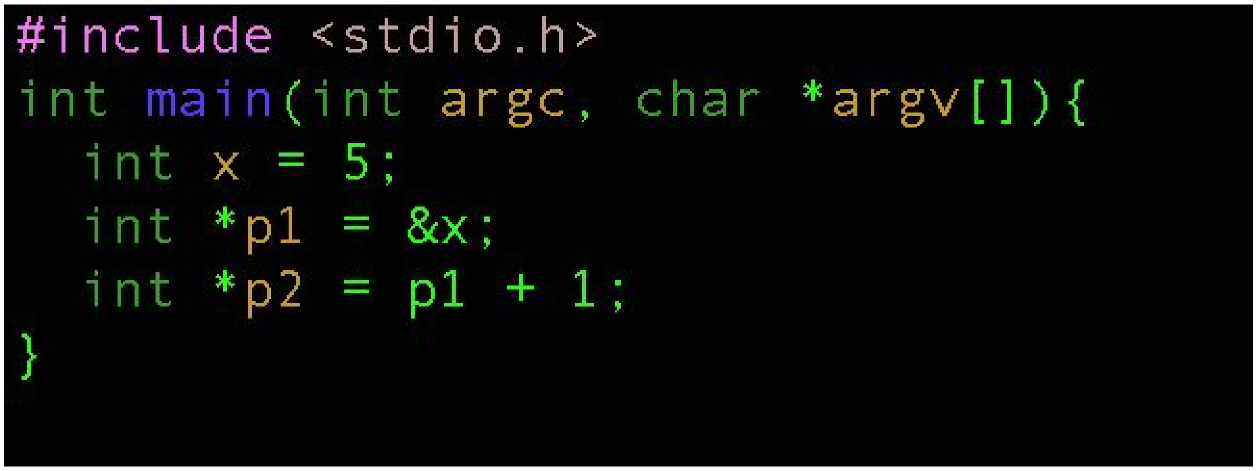
II. Body

A. How does Go compare to C?

Because Go was written in C, and written for C programmers, I'd like to compare the two. I'll start by exploring the items on this list and will write small programs to demonstrate these differences.

1. No Pointer Arithmetic

Pointers are familiar in Go in that they use the same "*" and "&" symbols as C. However, there is no support for pointer arithmetic, and attempts to increment, decrement, or use relational operators on a pointer will result in a compile time error (Pike). For example, this is not allowed:

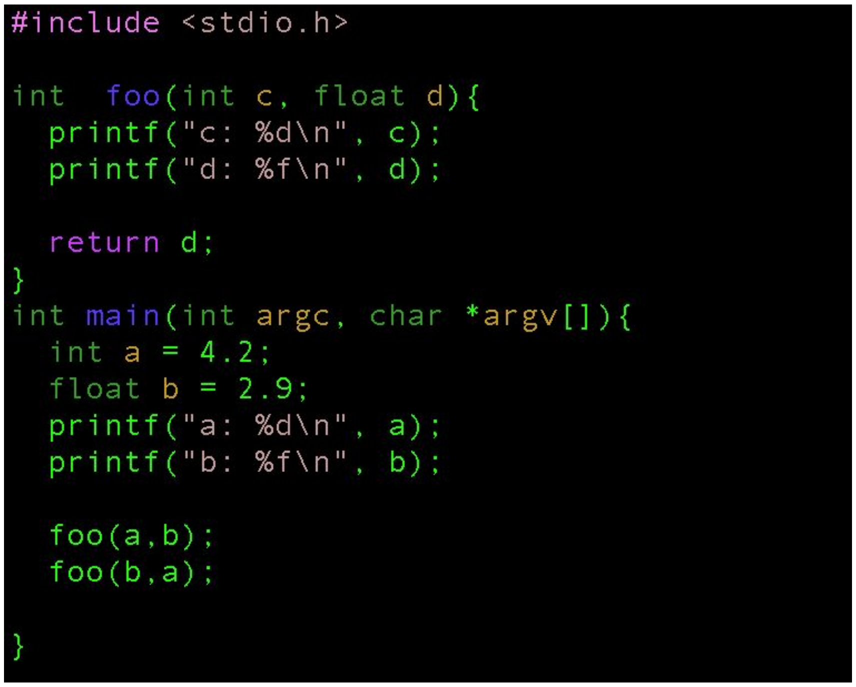


```
#include <stdio.h>
int main(int argc, char *argv[]){
  int x = 5;
  int *p1 = &x;
  int *p2 = p1 + 1;
}
```

You can, however, compare two pointers using the == operator. By disallowing pointer arithmetic, the Go language is more memory safe than C.

2. No Implicit Numeric Conversions

The creators of Go felt that more confusion than good arose from C's automatic conversion between numeric types (FAQ). In C's assignments and initializations, the value of the right operand is always converted to the type of the left operand; in function calls, the arguments are converted to the types of the corresponding parameters; in return statements, the value of the return expression is converted to the functions return type. These rules are not necessarily easy to work with, so Go decided to forgo them altogether (Crawford). Below is an example of legal C code:

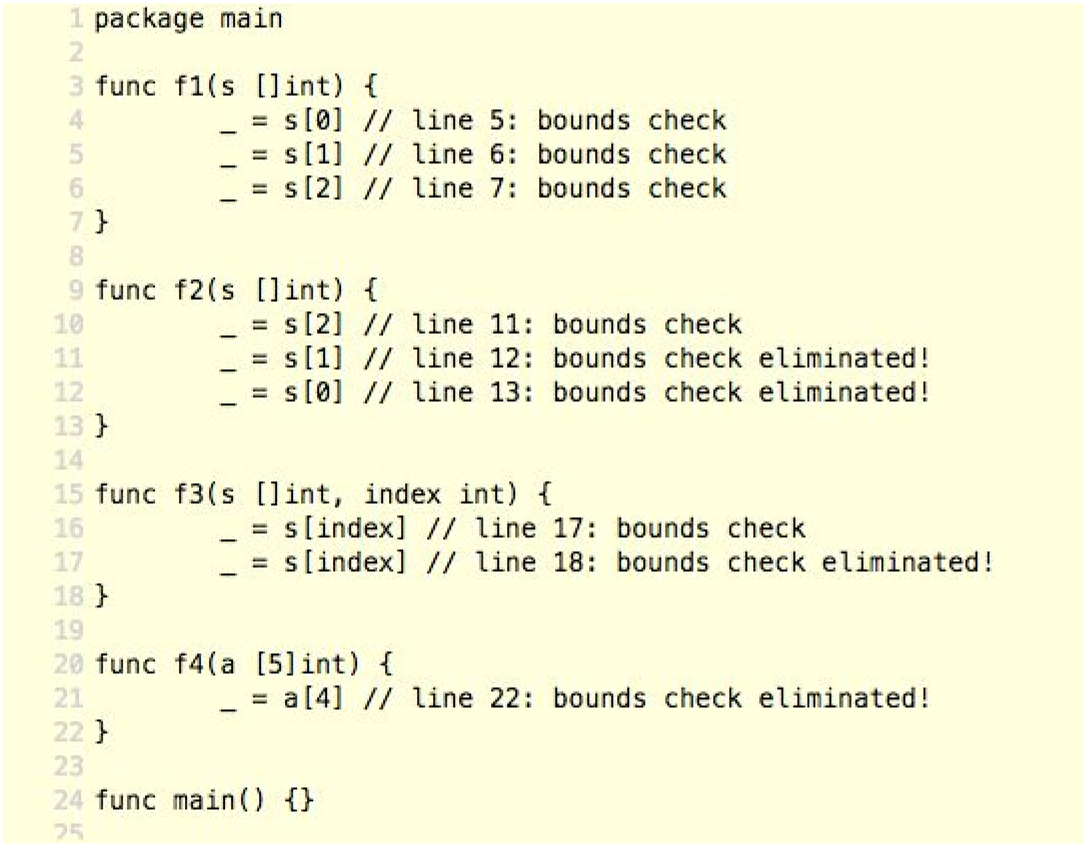


```c
#include <stdio.h>

int  foo(int c, float d){
  printf("c: %d\n", c);
  printf("d: %f\n", d);

  return d;
}
int main(int argc, char *argv[]){
  int a = 4.2;
  float b = 2.9;
  printf("a: %d\n", a);
  printf("b: %f\n", b);

  foo(a,b);
  foo(b,a);

}
```

3. Array Bounds Always Checked

As C programmers know, it is completely legal to take element at an array index greater than the length of the array. However, this memory is likely not what the programmer truly wants to access. Go differs from C in that it is memory safe - meaning before an array index can be accessed, the compiler checks if the index is within the bounds of the array.

Because the compiler must check the bounds of an array to prevent memory corruption, there is a necessary efficiency tradeoff. However, the compiler was implemented thoughtfully, and will avoid checking bounds if it is not necessary (“Bounds Check Elimination.”).

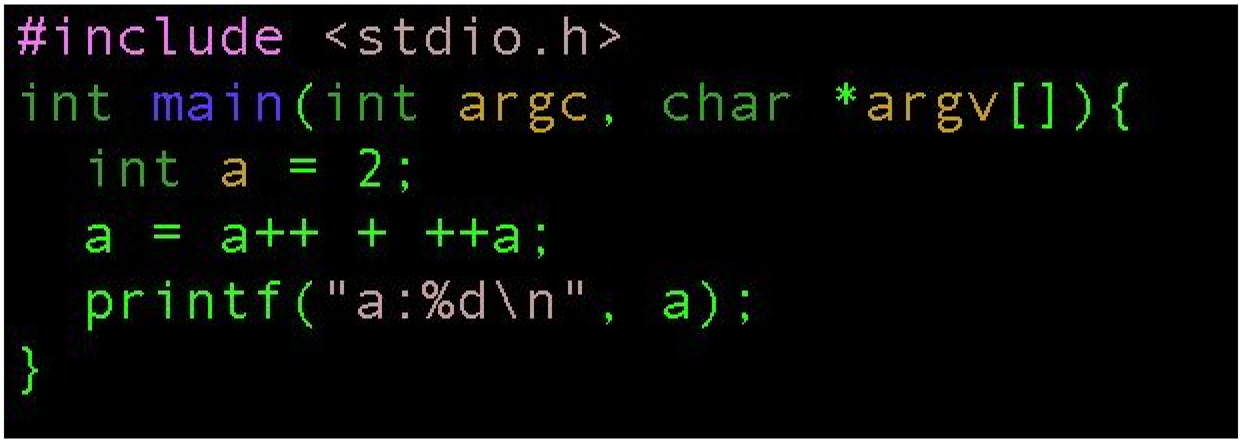


```go
package main

func f1(s []int) {
        _ = s[0] // line 5: bounds check
        _ = s[1] // line 6: bounds check
        _ = s[2] // line 7: bounds check
}

func f2(s []int) {
        _ = s[2] // line 11: bounds check
        _ = s[1] // line 12: bounds check eliminated!
        _ = s[0] // line 13: bounds check eliminated!
}

func f3(s []int, index int) {
        _ = s[index] // line 17: bounds check
        _ = s[index] // line 18: bounds check eliminated!
}

func f4(a [5]int) {
        _ = a[4] // line 22: bounds check eliminated!
}

func main() {}
```

Code based on “Bounds Check Elimination.”

4. ++ and -- are statements, not expressions

In C, ++ and -- operators can come before or after the variable they are modifying. This often leads to confusion. Because they are expressions, code like this is legal in C:



```c
#include <stdio.h>
int main(int argc, char *argv[]){
  int a = 2;
  a = a++ + ++a;
  printf("a:%d\n", a);
}
```

In the spirit of Go's simplicity, this is not allowed. The ++ statement does not return anything, but simply increments the operand by the constant 1 (Pike).

B. Inheritance in Go

Go takes a different approach to inheritance than object-oriented languages such as Java. Go allows for making an interface to represent abstraction, and structs or structures to represent data. In Go, "a type automatically satisfies any interface that specifies a subset of its methods" (FAQ). This is different from the typical object-oriented approach, because a struct does not explicitly declare which interface it implements, but it is determined at compile time whether it defines the methods of the interface. This allows for multiple inheritance in a way, because all a structure must do to implement multiple interfaces would be to define each of their methods. However, this is not true inheritance as the structure does not explicitly declare which interfaces it will inherit from.

In addition, we can also "inherit" from a struct, and this is done by either embedding the struct by value, or embedding a pointer to the struct (Westrup). This use of embedding rather than inheritance has several limitations in comparison with object-oriented languages such as Java. First, subtyping does not exist in Go for structs, only for interfaces (Westrup). A simple example would be in Java, making a penny class which is a subtype of coin, which is a subtype of money. In Go, we can only create a subtype of an interface, so we cannot have a line of inheritance like this. We would need to choose a superclass to be the interface, either coin or money, and have penny as the subclass by making it a struct. Another limitation is in the ability to override a method, which may not be done as expected because of how methods are bound to the struct at compile time.

Go emphasizes the use of embedding types within a struct or interface to gain the type's functionality, rather than inheriting such as in an object oriented paradigm. We can embed multiple interfaces within another interface, but their methods must not overlap in names. We can also embed multiple structs within a struct, which can be done by listing the types without giving them field names (“Effective Go”). When a type is embedded, its methods then become methods of the outer type, but when the method is invoked it is done as the inner class, not the outer one. This means the method is actually invoked on the field within the outer class, which is important to note as a programmer.

C. Game Development

To learn a new programming language it is quite important to answer a few fundamental questions which could help us to stay committed to the learning process of the new language. The article “Using the Go programming language” basically is helping us to find reasons of why learning Go could be a good investment of our time. In the article the author says that the main reason to shift to a new programming language is basically to ease the software implementation process. The author also highlights the problems that Google was facing which needed to be addressed by finding a simple and permanent solution. The problems were: long build times,code readability and concurrency; Go was the perfect solution for those (Westrup and Pettersson 8).

The purpose of introducing the Go language was to address some issues like long build time, Garbage Collection and difficulty in implementing concurrency in the program. So Go has absolutely solved those issues by reducing the build time, applying runtime garbage Collection and an easier way to implement concurrency in the program (Westrup and Pettersson 9). By analyzing these and other issues, we can analyze the advantages and disadvantages of using Go to solve specific problems such as game development.

The author also discusses and compares the Golang build tools with CMake and Java’s build tool Maven. In C and Java , developers do have to create that CMake file in C and Maven file in Java in order to configure them. The configuration of CMake is difficult to understand and modify which could lead to errors and could slow down the development process. Same goes with the Java’s Maven which build files on the XML.

So according to the author, this could be a time intensive task and error-prone . So Go provides a simple solution to this problem. It is so simple that we just need to create a directory in the Go workspace where the Go build tool automatically finds out the required dependencies that need to be compiled in order for the program to run (Westrup and Pettersson 40).

The author also discusses how the operation of concurrency was made easier compared to the other mainstream languages like c++ and Java. In Golang the whole concurrency is done by using goroutine and channels. This is the only way in Golang to work with concurrency. The author compares Golang with Java, where the concurrency mechanism is done by threading. Java provides numerous ways to implement the concurrency which could confuse different users. It is much simpler to implement concurrency in Go (Westrup and Pettersson 39). Because threading can be very important to gaming and many other applications, this is a definite advantage for Go.

The author also emphasizes some missing features that are not provided by Golang. Firstly, Golang does not support the object oriented programming approach but structures provide some of the features which are common in object oriented programming. It does not provide while loop and the keyword range has replaced the foreach loop which basically helps to iterate through the arrays in Golang. Surprisingly it does not support the exception and Generics (Westrup and Pettersson 23). So these missing features makes it compatible with the c programming language and Go can be integrated in the C programming language. However it would be really difficult because Go does support garbage collection but C does not, which might create confusion and inconsistency in the whole program for the user (Westrup and Pettersson 28). These missing features could be definite disadvantages for game development in Go.

The author also talks about the future of Golang. The mission of the Golang was to keep the code simple and clean, to avoid any complexity in the process of development. But to keep it simple, Golang has to sacrifice a lot of features like exceptions, Generics and object oriented programming. So we just have to wait and see if these compromises would help Golang to grow its popularity. The author also compares the popularity of Rust with Go where Rust provides those missing features in Go. So this could make the Golang team think into making those features available in the future(Westrup and Pettersson 46).

III. Conclusion

This paper laid the groundwork for further exploration of the Go programming language and its comparison to other popular programming languages. Though Go was created for the work done at Google, there are certainly many unexplored applications that would benefit from the language. For example, because of the clear syntax, Go may be a good choice for introductory computer science education.

Work Cited

"Bounds Check Elimination." *Bounds Check Elimination - Go 101 (an Online Book for Go Programming Language + Golang Knowledge Base)*, https://go101.org/article/bounds-check-elimination.html.

Crawford, Tony, and Peter Prinz. "C In a Nutshell." *O'Reilly* | *Safari*, O'Reilly Media, Inc., www.oreilly.com/library/view/c-in-a/0596006977/ch04.html.

"Effective Go". *golang.org*. https://golang.org/doc/effective_go.html.

"Frequently Asked Questions (FAQ)". *golang.org*. https://golang.org/doc/faq.

Henderson, Tim. "Object Oriented Inheritance in Go". *Hackthology*, Fri 20 May 2016, https://hackthology.com/object-oriented-inheritance-in-go.html.

Pike, Rob. "Go at Google: Language Design in the Service of Software Engineering". *talks.golang.org*. https://talks.golang.org/2012/splash.article

Westrup, Erik, and Fredrik Pettersson. "Using the Go Programming Language in Practice." May 23, 2014. https://www.researchgate.net/publication/312490994_Using_the_Go_Programming_Language_in_Practice.