Another Computer Language Comparison Blog

Michael Scott

MIRACL Labs

Abstract. In this article we describe our experience in implementing a high performance cryptographic library in multiple Computer Languages

1 Introduction

Most people have a favourite language they like to program in. Or they may have two or three on a horses-for-courses basis. Maybe they like one language for high level scripting, and another for low level stuff. Me I have always liked C and C++, and have many years experience using them. In the past I couldn't really comment on the competition, as I had no experience of them. Some I would have looked down on - Java was for people who can't get their head around pointers, Rust for high falutin academic types. In fact I would have held all sorts of absurd prejudices, but all based on no actual experience whatsoever.

As a result of an internal project I was tasked to implement a portable high-performance cryptographic language in as many languages as possible. All implementations in all languages should give exactly the same output for the same input. The library is quite interesting and useful in its own right, and is documented elsewhere on this site. Suffice it to say it implements symmetric encryption, hashing, random number generation, public key cryptography (using both RSA and elliptic curves), and pairing-based cryptography. This last is an out-growth of elliptic curve cryptography, which is of particular interest to my employers. The library does a lot of very complex all-integer calculations. Perhaps the most complex calculation is of the "pairing" itself. Indeed the viability of pairing-based cryptograhy depends on the ability to compute pairings quickly. So performance is a big concern.

2 First impressions

Its hard not to view language development over the last 40 years or so as anything other than an extended homage to Kernighan and Richie, and their invention of C. All of the languages considered here (C, Java, Swift, Javascript, Rust, C# and Go) are all very C like. In fact there is more difference between C and its immediate predecessor Fortran, than they is between C and any of the newer languages considered here.

3 First Hurdle

The type of integer arithmetic required by number theoretic cryptography is rather specialised. Since the numbers used in cryptography (maybe 256 bits) tend to be much bigger than register sizes (maybe 64 bits), we need to implement what is known as multi-precision arithmetic. So ideally one of these big cryptographic numbers would be represented as an array of computer words. Clearly the bigger the word-size of the processor the better, as a 256 bit number is more efficiently represented and manipulated as four 64-bit "digits" rather than as eight 32-bit digits.

Now when two 32-bit numbers are multiplied the result is 64-bits. This was clearly understood by Kernighan and Richie who had the foresight to specify two integer types, `int` and `long`. The product of two ints would fit into a long.

So on a 32-bit processor we need a 64-bit type. And on a 64-bit processor we need a 128-bit type. And this is where the trouble starts. Unbelievably none of our "modern" computer languages support a standard 128-bit integer type, even though 64-bit processors are now ubiquitous, now even pushing 32-bit processors our of their niche in the mobile market. Fortunately those responsible for the popular GCC compiler for C and C++ have seen the light, and a 128-bit type in now available. And GCC defines its own standard.

This means that in C by judicious use of `#define` and `typedef` we can switch in an instant from a 32-bit to a 64-bit version of the library. Alas this is not possible in any of the other newer languages. A good example would be Java where the `int` type is stuck for all time at 32-bits, and the `long` at 64-bits. Fine back in the day when 32-bit processors were all the rage, now its a ball-and-chain.

We can rather inefficiently program our way out of the problem by faking a pseudo 128-bit type. But its going to be slow. So it looks like my initial prejudice in favour of C will have some validity.

4 Taking the plunge

One thing I needed was a text editor that could cope with all of the languages and do nice stuff like keyword high-lighting. I didn't think there was one out there that would support them all, but there is. Lets hear it for Sublime Text! Highly recommended - get it here `https://www.sublimetext.com/`.

As I programmed in each language the Stockholm sydrome quickly kicked-in. I actually started to like them, and started to see their individual upsides. Some of the changes from C seemed quite arbitrary. Semi colons are in (Java, Javascript,C#), out (Swift and Go) and just when I got used to not having them, they are back in again (Rust). And what is wrong with the old `for(i=10;i>=0;i--)` statement? Its so expressive and you can do almost anything you like with it. Java, C#, Javascript and Go agree, but Swift and Rust do not. What is so much better about `for i in (0..11).rev()`? Also each language has its own preferred convention for naming variables, and can (like Rust) be quite tediously insistant that we stick to the house rules.

A big issue for me is memory management. Not that I have much memory to manage, relatively little actually. So I would like all memory to be allocated off the Stack, and not the Heap. With C you know exactly where it comes from. As a programmer I am a bit of a control freak, so I found it unsettling not to know exactly what was happening under the hood with the other languages. Its seems that Java and Go and Swift (to a lesser extent) like to use Heap memory, while Rust was more C-like in this regard. There is a "leave it to me attitude" about the newer languages that I am not entirely comfortable with.

I like clean syntax, and Swift, Javascript, Java and Go tend to be friendly in this regard. Rust is more C-like and its hard to avoid a surfeit of `&`s and `mut`s. Given the same result I would hope that each language is designed so that the cleanest syntax leads to the fastest code. In other words I didn't want to fight the language to get it to do something efficiently.

I had most trouble with Rust, but then I noticed something. With Rust my debugging tended to shift from run-time debugging to compile-time debugging. The Rust compiler actually made it hard to write incorrect code. Quite impressive actually.

Go is really clever. I love its minimalism. And I loved its "enhanced structures" which were a perfect fit for what I was trying to do. Rust has something similar. I didn't really need the full power of classes, which some of the other languages forced on me.

The sequence I followed was C to Java to C# to Javascript to Go to Swift to Rust. This was probably rather fortuitous, as differences with the immediate predecessor were reasonably small. And in fact the transition from Java to C# is so simple, I got an automatic tool to do it for me.

Maturity was an issue for Swift, as the specification kept changing even as I was writing it. Hopefully it will settle down with version 3.

4.1 How fast?

My pairing calculation program consisted of a loop which calculated 1000 pairings. For those that know about pairings, the actual pairing was calculated on a 455-bit BLS curve. Compiler optimization was maximised in all cases.

All timings are in seconds, mostly on an i3 Core Intel processor. The Swift timings are on an Apple Mac, scaled for comparison purposes.

1. C (32-bit) – 22
2. C (64-bit) – 9
3. Java (32-bit) – 41
4. Java (pseudo 64-bit) – 24
5. C# (pseudo 64-bit) – 21
6. Javascript (Google Chrome) – 390
7. Go (pseudo 64-bit) – 182
8. Swift (32-bit) – 187
9. Rust (32-bit) – 23

4.2 Discussion

What the hell is the problem with Go? It is so slow! OK it compiles very fast, but who cares. Its like an ISP that boasts unlimited downloads, without mentioning the very limited bandwidth. And Swift ain't so swift. But you have to admire the performance of the Java and C# just-in-time compilers. Clearly those man-years invested have paid off. Rust claims to be fast, and it is.

But C is still the winner in terms of performance. Which was my favourite? Well each one in turn as I used it, and my last effort was with Rust, so yes right now I like Rust a lot. But after a cooling off period I will probably come full circle back to C.