JANCY: The Return of the Pointers

Contents

[Introduction 1](#_Toc369799250)

[C 1](#_Toc369799251)

[C++ 1](#_Toc369799252)

[Java/C# 2](#_Toc369799253)

[D 3](#_Toc369799254)

[Jancy 4](#_Toc369799255)

[Motivation 4](#_Toc369799256)

[Language design 5](#_Toc369799257)

[Compiler architecture 8](#_Toc369799258)

# Introduction

### C

A long time ago, in a galaxy far, far away… and to be more precise, in the end of sixties (starting in 1969) at AT&T Bell Labs Dennis Ritchie has created the most important programming language of all times: the C language. It’s difficult to overestimate its influence on development of computer science. Even now, over 40 years since its birth, good old C is still among the most used programming languages – regardless of which exact rating of programming language popularity one would choose. Moreover, the top of the list would always be shared with its direct descendant C++ and other sibling languages whose syntax was derived from or heavily influenced by C: Java, JavaScript, C#, Objective C. And if the rating would consider only the usage in systems programming, then the gap between C & C++ and the nearest competitor becomes simply embarrassing.

I love C. Syntax of C is simple, elegant, expressive and universal. And what’s very important in systems programming, the layer of code generated by compiler is thin and predictable: experienced programmer could take a look at the C snippet and immediately estimate how the same snippet would look in assembly language after compilation.

### C++

Direct descendant of C which emerged more than a decade later in 1983, C++, unfortunately cannot say the same about itself. C++ has passionate supporters and no-less passionate and irreconcilable opponents. Holy wars on C++ go on all the time, on any programming forum, any weather, any season. Argument usually goes around the following statements.

* C++ is indeed extremely expressive and powerful
* High performance of C++ programs

Actually, the second in no small measure is the consequence of the first. Thanks to expressive power of C++ programmer is able to create high-performance code.

* Difficulty of C++

Language is so difficult that it earned place in programmers jokes – and rightfully so. Together with excellent expressive power come numerous and diverse capabilities to “shoot your own foot”. It’s so much easier to write bad C++ than it is to write bad C!

The layer of compiler-generated code has significantly grown in size and complexity and became much less predictable even by very experienced programmers.

It’s quite possible that largely because of the above even after good C++ compilers came into mainstream, good old C remained the main language for systems programming. Windows, Linux, Doom and Quake are all written in plain C while C++ was around for many, many years.

* C++ lacks many capabilities which are expected to be built-in into the modern general-purpose language, such as automatic memory management or reflection.

As a bottom line, it’s totally possible to efficiently use C++ as a programming language for building software of any level of complexity. However, very high qualification of developer is an absolute must. Moreover, even a qualified specialist from time to time shoots to his own foot or feet of his colleagues.

### Java/C#

Managed languages came to finally protect the feet of developers. Yet another decade past creation of C++, Java emerged in 1995. Microsoft-created managed platform .NET with its flagman language C# initially looked like a clone of Java, thoughtlessly copying both good and bad but – to give the credit where it’s due – was later quite ingeniously extended. In the following paragraphs though, I will write about Java only. But most to certain extent applies to C# also.

Java compiler generates not the machine code for the specific processor, but rather Java byte-code, which is interpreted or JIT-compiled by Java Virtual Machine at run-time. This allows reusing of the same binary module on multiple architectures without re-compilation.

Java Virtual Machine also provides built-in facility for automatic memory management via garbage collection. Java developers do not need to worry about explicit freeing of memory blocks allocated for various objects: garbage collector will take care of that. This, of course, does not mean that memory leaks in Java are non-existent. Besides, there are situations where it’s highly desired to manage memory manually instead of relying on convenient but not-so-deterministic facility of garbage collection. But generally speaking, built-in accurate garbage collection is one big fat plus, which is admitted by most (adequate) C/C++ proponents:

* Large part of any C/C++ program is dedicated to freeing memory. If memory management was automatic, for example, most (not all, of course – but absolute most!) C++ destructors would simply become irrelevant cause all they do is free memory
* The alternatives of automatic memory management available to C++ developers (namely, variations of reference counting or conservative garbage collection) are in many respects inferior to built-in accurate GC.
* Passing objects between C++ modules is largely handicapped by the lack of standardized facility of automatic memory management
* The whole class of bugs related to premature freeing or the opposite, non-freeing of memory blocks – simply goes away

As a consequence of the above, built-in garbage collection simplifies and, the therefore, speeds up and reduces the cost of development cycle.

Java syntax resembles heavily simplified C++. As a consequence of this simplification and also as a good evidence of it Java grammar could be expressed with context-free LR (1) – unlike context-sensitive C++ (and in many rules so!). This is of course a big advantage for everybody, from regular Java developers and to developers of code-assisting IDEs. Simple is good!

Besides grammar simplification, the following language facilities were also given the boot.

* Namespaces
* Type aliases (typedef)
* Global functions and variables
* Pointers and pointer arithmetic
* Structs and unions
* Explicit stack allocation, RAII and manual memory management
* Operator overloading
* Multiple inheritance
* Const-correctness
* Default argument values

While I have no intention of starting yet another holy war, I will risk saying that most of these terminations were unjust. Of course there was reasoning for Java designers’ part. Like simplification of syntax, compiler, virtual machine; or like improving language safety via not giving a developer potentially dangerous tools; and so forth and so on. But it’s so easy to give examples where each of the above could be very helpful to make the code sometimes more elegant, sometimes more logical, and often times more efficient!

All in all built-in accurate garbage collector, familiar and simple C-like syntax, safety and ability to run the same binary on any platform given the fact it has Java Virtual Machine led to unquestionably huge success of Java. If we take a look at the ratings of programming language popularity (which I start this article with) we will see that not only Java consistently takes positions in the top but it also slowly but steadily improves that year by year.

### D

Although D programming language often doesn’t even make it into top ten or top twenty of the above ratings, it would be unjust not to say anything about this interesting language. D programming language was created by Walter Bright as a re-thinking of C++ at 1999 – about the same time Microsoft released the first version of C#. The main motto in design of D is: combining the performance of C/C++ world with convenience and safety of managed world.

D has full ABI (application binary interface) compatibility with C/C++ programs, supports pointers to data and functions, manual memory management, allows mixing in portions in assembly language and at the same time provides facilities from the managed world such as garbage collection, nested functions and closures, strong meta-programming facilities and more.

D is an awesome language and gave me tons of inspiration. Years back I was seriously considering and was running experiments on using D as a scripting language in our products, but youth of the D at that time, and most importantly the lack (again, at that time) of available for commercial usage back-end put that on hold.

Besides, D still has some rather questionable design decisions, while unsafe pointers and possibility of mixing in assembly snippets largely denies the usage of D as of a safe scripting language.

# Jancy

### Motivation

#### Motivation №1, bicycle-inventive

One of the remarkable qualities of Russian education was (and I hope still is) nurturing of critical view on the things. Starting with senior high and certainly in the university we were taught not to blindly accept what is written in the textbooks or what is told by a lecturer at the blackboard. I remember at the very first lecture on physics our teacher without breaking a straight face has given us a proof that 2 equals 5 (these are the designations of marks in Russian education system btw), and right after that followed the proof that acute angle equals the obtuse one. Then he put his idea into words: try not to thoughtlessly accept and memorize something told or shown by undoubtedly smart people (I suspect he meant himself) but rather comprehend and understand why it is the way it is, why it works like this, and not like that. Perhaps, you will find error in their reasoning. Perhaps, you will find another way to do the same which would be somehow better than the original one. And perhaps, you will realize that it’s better to apply your time and efforts to something else instead of trying to fix something that works already.

On one hand, I am extremely grateful for being taught and trained to use this over-analytical way of thinking. On the other hand, I to certain extent have become an irredeemable bicycle inventor – and that is of course not the title to be proud of. To my defense I can say though that another rule taught in university – namely, the rule to study existing types of bicycles before starting to build your own one – without a doubt reduced the number of bicycles I have created throughout my life by a magnitude. Maybe even two.

I was programming for the last 20 years, 15 years professionally. In every language I happened to use at work or in my leisure programming, my internal bicycle inventor was consistently looking for and finding things to improve. It so happened that my professional area of expertise is systems programming, particularly compilers, debuggers and other development tools. It is therefore not surprising that for many years I have had little toy languages as proving grounds for experiments. Taking into account my confession in love to C language, it’s not surprising that these languages always had C-like syntax.

Until recently though, these languages had no name – despite the fact that previous versions (or should I say incarnations? Cause most of the time the new version did not have a single line from the previous one) were used in commercial products as DSLs (domain-specific scripting languages). At one point I decided to put a stop to this anonymity and after a rather brief (lucky!) linguistic odyssey I found the name: Jancy.

It’s easy to see Jancy is an acronym: [in between] JAva aNd C. One could tell that I took the Java as starting point and brought in the features and facilities I love in C/C++. Or the other way around: that it’s simplified C/C++ which is taught the force of managed world. In that respect Jancy targets the similar design goals as D. Plus, of course, I implemented the features which are not found anywhere else (ok, maybe they could be found, but they certainly have not made their way to the mainstream programming languages. yet) – the features that I always dreamed of and which passed the test of my toy languages proving grounds.

There, I confess. One of the main motivations of creating Jancy was feeding my internal bicycle inventor.

#### Motivation №2, practical

But there was also a practical motivation for turning leisure development into a full-scale project. A couple of years ago our company released a product named I/O Ninja, one of the modules of which had to analyze binary data blocks and write results of analysis into log. To avoid unjust accusation in advertisement, I will purposefully not going to dive into details or even explanations what this product was all about and why it was needed. It was necessary to analyze and generate binary data blocks – that’s it.

Since we wanted to make analyzes as flexible as possible, we had to move the analyses logic into plug-ins or script. And since in the long run we planned to give our users possibility to add new analyzers or modify existing ones – script seemed like a better approach (plug-ins would mean the necessity of publishing the SDK; our users would also need to possess both C/C++ development tools and the required qualification to undertake C/C++ development).

So, script it is. But herein lies the problem: it’s difficult to perform analyzing/generating binary data blocks without data pointers and pointer arithmetic. Without these facilities it’s all about taking the bytes at fixed indexes and clueing them into words, and this generates unreadable and unsupportable code. Working with binary data is best done with pointer arithmetic. Period. There are no well-known/recognized scripting languages with data pointer supports suitable for embedding into C/C++ host application – not back then, not now. And even if there are some experimental languages Google knows nothing about, they are at no advantage over in-house-made bicycle.

The first version of I/O Ninja was actually built on plug-in architecture; the second one – on scripts written in Jancy prototype (unsafe pointers, automatic memory management based on reference-counting, rudimentary class support and hand-made virtual machine). The third version is under active development right now and is based on scripts in full Jancy language.

This product is a practical motivator for creating Jancy and will serve as an excellent real-life proving ground.

#### Motivation №3, philosophical

Let me philosophize just a bit more before getting to Jancy design goals and feature list. I in no way want to say that creation of Jancy was “forced”. That there was no way to solve the problem using different languages, different tools and different approaches. There was. There always is another way.

But that’s the story of progress: most innovations simply make some process a little bit better. A little bit faster, or easier, or more efficient. It was totally possible to write everything in good old C. And even directly in assembly language! Just like it’s still totally possible to move from point A to point B on horses or by walk. Or use candles to illuminate the house. But any process can always be improved a little bit. And then improved a bit more. These little tiny bit improvements, each of which is by no means revolutionary (hey, it was totally possible to achieve the same result before!) – add up to bigger ones, and even bigger ones. And finally they add up to the extent that they allow doing something which was simply impossible earlier.

These little non-revolutionary improvements (no sarcasm intended!) is what we are trying to introduce.

### Language design

The main model of usage considered while designing Jancy was the usage a scripting engine embedded into the host C/C++ application. In the long run, of course, it would be flattering to see Jancy used as a general-purpose language tailored for systems programming.

So, the main design goals of Jancy were:

* Usage as a scripting language from the host C/C++ application
* Familiar C-like imperative syntax
* Safe pointers and pointer arithmetic
* Automatic memory management via accurate garbage collection

Besides that, Jancy has a long list of innovations/improvements/features of different level of importance, summary of which is the following:

* Full ABI-compatibility with C/C++
  + After proper declaration of data types in Jancy script and in host C/C++ application it’s possible to directly pass data through arguments and return values, without the need to explicitly pushing/popping the stack of virtual machine, packing/unpacking them into variant-like containers etc. All the following types are supported:
    - All primitive C/C++ types (also integer types with inversed byte order)
    - Structs (with arbitrary pack factor)
    - Unions
    - Arrays
    - C/C++ pointers to data or to functions
* Classes
  + Classes are special types of data with ancillary header
  + All the standard features of classes are supported:
    - Constructors / destructors (both non-static and static)
    - In-place field initialization
    - Operator overloading
    - Abstract and virtual functions
  + Simple multiple inheritance (no dreadful C++ virtual inheritance)
  + Implementation of methods could be in-class, or out-of-class (like in C++).
  + Support of “preconstructor”s – special methods that should be called before any of the overloaded constructors: analogue of Java initializer blocks.
* RAII and full control over data storage
  + Support of explicit stack allocation and RAII paradigm (like in C++)
  + Support of allocating memory for multiple objects at once (by declaring class field members like in C++), without the need of explicitly calling “new” for every object
  + Built-in support of TLS variables (“thread” storage specifier)
  + Manual memory management of unmanaged heap with “heapu new” / “delete”
  + Storage specifiers: “static”, “heap”, “heapu”, “stack”, “thread“ control which memory should be used for particular data entity (field, variable, new operator)
* Pointers to data
  + Const-correctness
  + Safe pointer arithmetic
  + Support of weak class pointers
* Properties
  + Without false modesty – the most comprehensive implementation of properties to date
  + Natural form of simple property declarations
  + Full form for declaring properties of arbitrary complexity (with overloaded setters, member fields, ancillary methods etc)
  + Indexed properties (properties that expose semantics of arrays)
  + Autoget-properties eliminate the routine of writing trivial getters
  + Bindable-properties notify subscribers the moment they change
  + Bindable-data automatically generate bindable properties with trivial getters and setters
* Pointers to functions and properties
  + Function and property pointers can capture the value of arguments
  + Automatic creation of thunk functions when argument signatures do not match but could be converted
  + Support of weak pointers to functions and properties
  + Schedule operator on a function pointer creates an ancillary function pointer which ensures execution of the original one in desired context (in the proper thread, with mutex held, etc)
* Multicast
  + Multicasts allow accumulation of function pointers and then calling them all at once
  + Event pointers to multicasts only allow subscribing/unsubscribing but not calling
  + Two kinds of conversions from multicast to function pointer: live и snapshot
  + Support of weak multicasts (do not need explicit unsubscribe)
* Reactive programming
  + Syntax constructs for declaring expressions which need to be automatically re-evaluated upon the change of bindable r-values
  + Syntax constructs for assigning blocks of code as on-change handlers without the need of explicit subscribe-unsubscribe
* Exception handling
  + Exceptions in Jancy is merely a syntactic sugar over old C-style error code model
  + As a result, it is compatible with any external language
  + catch and finally can be declared at any scope
  + When calling the same function, developer can use either error code check, or exception semantics – depending on what’s more appropriate or convenient in particular case
* Dual access control model
  + Jancy only has 2 access specifiers: public and protected
  + Each namespace splits the rest of the namespaces into 2 teams: “friends” and “aliens”. Friends can have access to anything, aliens – to “public” items only
  + Dual modifier “constd” for declaring type of data which looks like “const” for aliens, and “non-const” for “friends”. Designated to eliminate the routine of creating trivial properties whose only purpose is providing read-only access
  + Dual modifier “eventd” for declaring events or event pointers. Aliens can only subscribe/unsubscribe (“event” semantics) while friends must be also able to fire event (“multicast” semantics)
* Special literals
  + Hex-literals for convenient declaration of constant binary data blocks
  + Formatting literals dynamically generate strings substituting the values of expressions referenced from within the literal (perl-style formatting)
* Miscellaneous
  + once/thread once
  + Comprehensive implementation of curly-intitializers
  + break-n/continue-n: controlling outer loops
  + basetype-n: convenient referencing of base types
  + enumf: flag enumerations
  + extend: extending types with methods after declaration
  + alias: expression aliases
  + Scopes-in-switch
  + Constructors/destructors of compilation units

### Compiler architecture

Jancy compiler uses lexical analyzer generated by Ragel universal finite state machine compiler. It perfectly suits for building lexer thanks to convenience and expressiveness of its input language and even more importantly, to the unbeatable performance of the output code.

As a backend Jancy uses LLVM. It allows making use of a reliable optimizer and native code generator for a wide variety of platforms at once. It also significantly simplifies ensuring of ABI compatibility with C/C++. All in all, LLVM usage as opposed to hand-written code generator or virtual machine was a decision that was never under discussion since I first laid my hands on LLVM.

As for syntax analyzer though I couldn’t resist it and written my own ~~bicycle~~ generator of table-driven top-down LL (k) parsers. What’s wrong with ANTLR, Coco, Yacc/Bison, Lemon and other undoubtedly respectable and tried-and-true parser generators is a an interesting topic – but for a separate discussion. What’s important here is Jancy compiler uses not a hand-written recursive descent but rather a generated parser. Among other things it also means that we have permanently relevant BNF grammar which can be printed out, discussed and easily adjusted. Grammar of Jancy belongs to context-sensitive LL (2).

The output of parser is LLVM IR without intermediate generation of AST: unlike bottom-up parsers in top-down parsers it is rather convenient to perform semantic analysis and generate IR parallel to parsing, right as the parser matches the rules.

The model of collaboration between parser and lexer is peeped in Lemon. Jancy parser does not call lexer. Instead, the external loop asks lexer to tokenize the next chunk of source and feeds tokens to table-driven parser. This model allows incremental parsing of incomplete compilation units, supports pause and resume, plus, like in any table-driven parsers, the memory for rule attributes is allocated not on stack (which requires artificial nesting level limitations cause of danger of stack overflow) but rather on heap.

Syntactic/semantic analyses is mostly performed in 2 passes (some rules require 3 passes). This, however, does not mean re-tokenizing the source multiple times. The second pass is needed to allow usage of global entities (namespaces, types, variables, functions and properties) before their declaration (which could happen to be in a separate compilation unit at all). The third pass is necessary for preliminary calculation of layouts of reactor classes.