JANCY: The Return of the Pointers

Contents

[Introduction 1](#_Toc374884837)

[Inspiration 2](#_Toc374884838)

[C 2](#_Toc374884839)

[C++ 2](#_Toc374884840)

[Java/C# 3](#_Toc374884841)

[D 4](#_Toc374884842)

[Motivation 4](#_Toc374884843)

[Motivation №1, wheel re-inventive 4](#_Toc374884844)

[Motivation №2, practical 5](#_Toc374884845)

[Motivation №3, philosophical 6](#_Toc374884846)

[Full feature list 6](#_Toc374884847)

[Compiler architecture 8](#_Toc374884848)

[Current status & conclusion 9](#_Toc374884849)

[Appendix 9](#_Toc374884850)

[Full language grammar 9](#_Toc374884851)

# Introduction

What?! Yet another programming language? Well… yes. Meet Jancy. Before we even start, let me make a list of the features that might help you decide whether you are interested enough to keep reading.

### Main Design Goals

* Object-oriented language with C-family syntax
* ABI (application-binary-interface) compatibility with C/C++
* Tailor-suited for being used as a scripting language from C/C++ host application
* Automatic memory management via accurate GC (garbage collection)

### Key Features

* Safe pointers and pointer arithmetic
* Properties (one of the most comprehensive implementations)
* RAII (resource-acquisition-is-initialization) paradigm support
* Reactive programming paradigm (one of the first existing implementations in a general-purpose imperative language)

### Other notable features

* Closures of functions and properties
* Weak pointers (do not retain the object)
* Multicasts and weak multicasts (do not require to unsubscribe)
* Explicit stack allocation, manual memory management, thread local storage
* Const-correctness support
* Multiple inheritance support
* Perl-style formatting support
* …and many more – no, seriously! ;)

Now let’s slow down a bit and take a look back.

# Inspiration

### C

A long time ago, in a galaxy far, far away… and to be more precise, in the end of the 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 the 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 rating of programming language popularity one would choose. Moreover, C always shares the top of the list 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 only consider the usage in systems programming, then the gap between C/C++ and the nearest competitor becomes simply embarrassing.

I love C. The syntax of C is simple, elegant, expressive and universal. And what’s very important in system programming, the layer of code generated by the compiler is thin and predictable: an experienced programmer could take a look at the C snippet and be able to infer how the same snippet will look in assembly language after compilation.

### C++

The 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++ still go on all the time, on any programming forum, any weather any season. The arguments usually revolve around the following statements:

* C++ is, indeed, extremely expressive and powerful
* C++ programs have high performance

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

* C++ is difficult!..

…So difficult that it earned itself a 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 grown significantly 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 system programming. Windows, Linux, Doom and Quake are all written in plain C while C++ has been around for many, many years.

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

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

### Java/C#

Managed languages came to finally protect the feet of developers. Fast forward another decade past the creation of C++, and Java has emerged (in 1995). Microsoft-created managed language C# initially looked like a clone of Java that mindlessly copied both good and bad. It was, however, quite ingeniously extended later. In the following paragraphs though, I will write about Java only. Still, most of it also applies to C#.

The Java compiler doesn’t generate the machine code for the specific processor. Instead, it produces the Java byte-code, which is then interpreted or JIT-compiled by the Java Virtual Machine at run-time. This allows reusing the same binary module on multiple architectures without re-compiling.

The Java Virtual Machine also provides a 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 since the 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 the 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:

* A large part of any C/C++ program is dedicated to freeing memory. For example, if memory management was automatic, most (not all, of course, but most) C++ destructors would simply become irrelevant because 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 for 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, the built-in garbage collection simplifies and, therefore, speeds up and reduces the cost of the development cycle.

Java syntax resembles drastically simplified C++. As a result and also as a good evidence of this simplification, Java grammar could be expressed with context-free LR (1) grammar – unlike the context-sensitive C++ grammar, which is widely considered one of, if not the most, complicated grammars in modern programming languages. This simplification is a big advantage for everyone, from regular Java coders to the architects of code-assisting IDEs. Simple is good!

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

* Namespaces
* Type aliases (typedefs)
* 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. Sure, there was reasoning on the Java designers’ behalf: the simplifications of syntax, compiler and virtual machine; the improvements in language safety (by not giving developers potentially dangerous tools), and so forth. But it’s also easy to come up with the examples where each of the above facility could be very helpful in making the code sometimes more elegant, sometimes more logical, and oftentimes more efficient!

All in all, the built-in accurate garbage collector, familiar and simple C-like syntax, as well as the safety and ability to run the same binary on any platform (provided that there is a Java VM available) have led to unquestionably huge success of Java. If we take a look at ratings of the programming language popularity (which I’ve started this article with) we will see that Java not only consistently remains among the top contenders but also slowly but steadily improves its ratings year by year.

### D

Although D often doesn’t even make it into top ten or top twenty of the “language charts”, it would be unfair to say nothing about this interesting language. The D programming language was created by Walter Bright as a re-thinking of C++ at 1999 – about the same time Microsoft released its first version of C#. The main motto of the D language design was “combining the performance of C/C++ with the convenience and safety of the 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 of assembly language and at the same time provides facilities from the managed world such as garbage collection, nested functions and closures, extensive meta-programming tools and more.

D is an awesome language and it gave me tons of inspiration. Years back I was seriously considering and was running experiments on using D as a scripting language in a product I was designing, but D was still rough around the edges at the time. Most importantly, the absence (again, at that time) of a back-end available for commercial usage has put D on hold.

Besides, I also think that D has its share of rather questionable design decisions (for example, scope guards or auto-conversion of functions into properties). Further, unsafe pointers and the possibility of mixing in assembly snippets largely prevent using D as a safe scripting language.

# Motivation

### Motivation №1, wheel re-inventive

One of the remarkable qualities of the Russian education was, and I hope still is, the nurturing of a critical view on things. Starting with Senior High and certainly in the university we were conditioned to not blindly accept what was written in the textbooks or what was told by a lecturer at the blackboard. I recall that 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 grades in the Russian education system by the way). He followed that up with the proof that the acute angle equals the obtuse one. Then he put his idea into words: do not just mindlessly 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 the way it works, and so on. Perhaps, you will find an error in their reasoning. Perhaps, you will find a better way to do the same. Or, perhaps, quite the opposite – you will realize that it’s better to apply your time and efforts elsewhere instead of trying to fix something that works already.

On the one hand, I am extremely grateful for being taught and trained to use this over-analytical way of thinking. On the other hand, to a certain extent I have become an irredeemable wheel inventor – and that isn’t usually the title to be proud of. In my defense I can say that another rule taught in the university – namely, the rule to study existing types of wheels before starting to build your own one – without a doubt has reduced the number of wheels I have created throughout my life by an order of a magnitude.

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

Until recently, my creations had no name – despite having been used in commercial products as DSLs (domain-specific scripting languages). At one point I decided to put an end to this anonymity and after a rather brief (lucky!) linguistic odyssey I found the name: Jancy.

It’s easy to see that Jancy is an acronym: [in between] JAva aNd C. You could say that I took the Java as a 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 the managed world. In that respect Jancy targets similar design goals as D. Plus, I implemented the features which are not found anywhere else (ok, maybe they could be found, but they certainly have not made their way into any mainstream programming language 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 for creating Jancy was feeding my internal wheel inventor.

### Motivation №2, practical

There was also a practical motivation for turning leisure development into a full-scale project. A couple of years ago our company, Tibbo, has released a product called I/O Ninja. This product can be used for network packet analysis. Hence, it has to parse the binary data and output results into a log file. To avoid accusations that I am advertising Ninja I won’t dive into the product specifics. The key point here is that I had to deal with the binary data.

Since we wanted to make data parsing as flexible as possible we had to move the parsing logic into plugins or scripts. And since in the long run we planned to give our users the ability to add new analyzers or modify existing ones – scripts 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 skills to undertake C/C++ development).

So, scripting it was. But herein was the problem: it’s difficult to deal with the binary data blocks without data pointers and pointer arithmetic. Without these facilities it’s all about taking the bytes at fixed offsets, gluing them into words, and this generates unreadable and unsupportable code. Working with binary data is best done with structs and pointer arithmetic. Period. There are no well-known/recognized scripting languages with data pointer support suitable for embedding into C/C++ host applications – not back then, not now. And even if there are some experimental languages Google knows nothing about, they are at no advantage over an in-house-invented wheel.

The first version of I/O Ninja was actually built on a plug-in architecture; the second one – on scripts written in a Jancy prototype (unsafe pointers, automatic memory management based on reference-counting, rudimentary class support and a hand-made virtual machine). The third version is under active development right now and is based on scripts in the “real” 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 into Jancy’s design goals and feature list. I am not saying that the creation of Jancy was “forced”, that the problem could not have been solved with a different language, a different tool, a different approach. It certainly could have been. There always is another way.

But that’s the story of progress: most innovations simply make some process just a little bit better, a little bit faster, or easier, or more efficient. It was totally possible to write everything in good old C, or even directly in assembly language! Just like it’s still totally possible to move from point A to point B on horseback, or by walking. Or use candles to illuminate the house.

However, any process can always be improved a little bit. And then improved some more. These little 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. Finally they add up to something really big, something that allows doing what was simply impossible earlier. But it always starts with a little tiny improvement.

These little improvements are what we are trying to introduce.

# Full feature list

* Full ABI-compatibility with C/C++
  + After proper declaration of data types in Jancy scripts and in the host C/C++ application it’s possible to directly pass data through arguments and return values without the need to explicitly push and pop the stack of the virtual machine, packing/unpacking them into variant-like containers etc. All of 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 infamous C++ virtual inheritance)
  + Implementation of methods is possible both in-class and out-of-class (like in C++).
  + Support for preconstructors – special methods that should be called before any of the overloaded constructors; these are analogous to Java initializer blocks.
* RAII and full control over data storage
  + Support for explicit stack allocation and RAII paradigm (like in C++)
  + Support for 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 for 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 entities (field, variable, “new” operator)
* Pointers to data
  + Const-correctness
  + Safe pointer arithmetic
  + Support of weak class pointers (the ones that do not retain the object)
* Properties
  + One of the most comprehensive implementations of properties ever!
  + 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 the desired context (in the proper thread, with mutex held, etc.)
* Multicasts
  + 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 a multicast to a function pointer: live and snapshot
  + Support of weak multicasts (the ones that do not need explicit unsubscribe)
* Reactive programming
  + One of the first implementations of the reactive programming paradigm in an imperative language
  + The dilemma of mixing reactive paradigm with imperative data flow is solved using dedicated blocks of reactive code – so-called “reactors”
  + Expressions within reactors are 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
  + Exception model of Jancy is merely syntactic sugar over the old C-style error code model
  + As a result, it is completely transparent and compatible with any external language
  + “catch” and “finally” can be declared in any scope
  + When calling the same function, the developer can use either error code check, or exception semantics – depending on what’s more appropriate or convenient in this 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 groups: “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 event pointers. Aliens can only subscribe/unsubscribe (“event” semantics) while friends are also able to fire the event (“multicast” semantics)
* Special literals
  + Hex-literals for convenient declaration of constant binary data blocks
  + Formatting literals dynamically generate strings by 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 functionality of existing classes with methods
  + “alias”: expression aliases
  + Scopes-in-switch
  + Constructors/destructors of compilation units

# Compiler architecture

The Jancy compiler uses a lexical analyzer generated by Ragel (a universal finite state machine compiler). It is perfectly suited for building lexers thanks to the 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 never questioned.

As for the syntax analyzer I couldn’t resist creating my own ~~wheel~~ 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 an interesting subject – but it is the one for a separate discussion. What’s important here is that the Jancy compiler doesn’t use a hand-written recursive descent but a generated parser. Among other things it also means that we have permanently relevant BNF grammar which can be, let’s say, printed out, discussed and easily adjusted. Grammar of Jancy belongs to context-sensitive LL (2).

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

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

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

# Conclusion and Current Status

Jancy is a cross-platform project that targets Windows, Linux, as well as Mac (planned) and will generate native code for any architecture supported by LLVM. Jancy is not currently open-source, but may become so in the future. A Jancy plugin for NetBeans with code assist support and debugging capabilities via GDB is under active development and is likely to be released together with the new Jancy-based version of I/O Ninja.

At present time we have a working demo of the compiler available for testing online, without any download or/and installation. Certain features which could be expected from a modern language (such as reflection, generics/templates, lambda-functions etc.) are not implemented yet, but they certainly will be there in one of the future releases. At the same time, the features that are implemented and thought to be ready might not work quite as expected after being exposed to rigorous testing through the web – it’s a first public release after all. Bug reports are, of course, more than welcome. We believe that during the development of Jancy-based I/O Ninja and with the help of web-based testing we will be able to polish both the language and the compiler to make it suitable for use in a wide class of applications.

We certainly hope that we have managed to spark your interest and would be happy to receive bug reports, questions and general feedback from the community of developers.

# Appendix

### Full language grammar

axl\_jnc\_Lexer.rl

axl\_jnc\_Decl.llk

axl\_jnc\_DeclarationSpecifier.llk

axl\_jnc\_Declarator.llk

axl\_jnc\_Expr.llk

axl\_jnc\_Expr\_s.llk

axl\_jnc\_NamedTypeSpecifier.llk

axl\_jnc\_Parser.llk

axl\_jnc\_Stmt.llk

axl\_jnc\_Stmt\_0.llk