JANCY: Возвращение Указателей

Contents

[Введение 1](#_Toc369725192)

[С 1](#_Toc369725193)

[С++ 1](#_Toc369725194)

[Java/C# 2](#_Toc369725195)

[D 3](#_Toc369725196)

[Jancy 4](#_Toc369725197)

[Мотивация 4](#_Toc369725198)

[Дизайн языка 5](#_Toc369725199)

[Архитектура компилятора 8](#_Toc369725200)

# Введение

### С

Давным давно, в далёкой далёкой галактике... а если конкретнее, в конце шестидесятых в AT&T Bell Labs Деннисом Ритчи был создан самый главный язык программирования всех времён и народов: язык C. Значение его трудно переоценить. Даже сейчас, спустя больше 40 лет после появления, старый добрый C остаётся одним из наиболее используемых языков – независимо от выбора конкретного рейтинга популярности/используемости языков программирования. Причём первые места на вершине любого подобного рейтинга он будет делить со своим непосредственным потомком C++ и другим родственными языками, синтаксис которых был порождён из синтаксиса С: Java, Javascript, C#, Objective C. Если же оставить в рейтинге только использование в системном программировании, то лидирующий отрыв C и С++ от остальных языков будет ну просто неприличным.

Я люблю C. Синтаксис C прост, элегантен, выразителен и универсален. И, что очень важно в системном программировании, прослойка кода, генерируемая компилятором, тонка и предсказуема: опытный программист, взгялнув на кусочек кода на C может сразу приблизительно прикинуть как этот же кусочек будет выглядеть на ассемблере после компиляции.

### С++

Появившийся спустя десятилетие после C его прямой наследник C++, к сожелению уже не может похвастаться этими качествами. У языка C++ есть горячие сторонники и не менее горячие непримиримые противники. Священные войны, посвященные C++, нередки на любом программистком форуме в любое время года. Споры обычно строятся вокруг следующих аргументов.

* Крайне широкие выразительные возможности синтаксиса C++
* Высокая производительность программ, написанных на С++

Возможно, второе является отчасти следствием первого. Именно вследствие выразительности синтаксиса разработчик может создавать высокоэффективный код.

* Сложность C++. Язык сложен настолько, что вошёл в программистские шутки.

Вместе с замечательными выразительными возможностями язык предоставляет многочисленные и разнообразные возможности «выстрелить себе в ногу». Написать плохой код на C++ гораздо легче чем на написать плохой код на C. Кроме того, неявно генерируемый компилятором код больше, требует больше памяти, и его уже не так легко предсказать даже опытным разработчикам.

Вполне вероятно, что именно этими причинами и обусловлен тот факт что даже после появления хороших компиляторов C++ язык C остался главным языком системного программирования. Windows, Linux, Doom и Quake написаны на C в то время как C++ существовал уже многие годы.

* В С++ отсутствуют многие возможности, являющиеся встроенными в большинство современных языков, такие как автоматическое управление памятью или reflection.

Как следствие всего вышеизложенного, эффективно программировать на C++ безусловно можно, однако требуется высокая квалификация разработчика. Кроме того, даже квалифицированный специалист временами стреляет в ногу себе и/или коллегам.

### Java/C#

На защиту ног разработчиков от случайных выстрелов пришли managed языки. Спустя ещё одно десятилетие после создания C++ появилась Java. Созданная Microsoft managed-платформа .NET с флагманским языком C# изначально выглядела как клон Java, но, впоследствии, надо сказать, была весьма остроумно и удачно расширена и дополнена. В дальнейшем я буду рассматривать только Java, но подавляющая часть всего, о чем будет сказано ниже, в той или иной мере относится и к C#

Компилятор Java генерирует не машинный код для конкретного процессора, а Java байт-код, который затем интерпретируется или JIT-компилируется виртуальной машиной Java. Это позволяет использовать один и тот же бинарный модуль на различных архитектурах без перекомпиляции.

Помимо этого, виртуальная машина Java обладает встроенным механизмом автоматического управления памятью. Java разработчикам не нужно беспокоиться о явном удалении выделенных под объекты блоков: об этом позаботится сборщик мусора. Это, конечно, не означает что утечки памяти в Java невозможны. Кроме того, иногда желательно управлять памятью вручную, а не полагаться на удобный, но недетерминированный механизм сборки мусора. Но в целом наличие встроенного сборщика мусора это неоспоримый плюс, который признают даже самые закоренелые сторонники C/C++:

* Значительная порция кода C/C++ программы посвящена освобождению выделенной памяти (в частности, подавляющая часть деструкторов C++ классов становится просто не нужна, так как занимается исключительно освобождением памяти)
* Альтернативные механизмы автоматического управления памятью, доступные C++ разработчикам (а именно различные вариации подсчёта ссылок или консервативные сборщики мусора) во многих отношениях уступают встроенному точному сборщику мусора (accurate gc)
* Передача объектов между С++ модулями затруднена в значительной мере именно из-за отсутствия стандартизированного механизма освобождения памяти
* Исчезает целый класс багов, связанных с преждевременным удалением или напротив, неудалением блоков памяти

Как очевидный итог, сборщик мусора упрощает, а значит, ускоряет и удешевляет процесс разработки.

Синтаксис Java представяет собой сильно модифицированный в сторону упрощения C++: синтаксис Java может быть выражан context-free грамматикой, в то время время как для корректного разбора С++ требуется знать значения идентификаторов в данной точке исходного кода (причём контекстная чувствительность встречается далеко не в единственном правиле грамматики C++).

Под волну сокращений также попали:

* Пространства имён
* Псевдонимы типов (typedef)
* Глобальные функции и переменные
* Указатели и адресная арифметика
* Структуры (struct) и объединения (union)
* Явное создание экземпляров классов на стеке, RAII и вообще ручное управление памятью
* Перегрузка операторов
* Множественное наследование
* const-корректность
* Умолчальные значения аргументов

Рискуя вступить на почву священных споров, всё-таки скажу что большинство сокращений были незаслуженными. Да, возможно, обоснованными: ради упрощения синтаксиса / компилятора / виртуальной машины и ради повышения общей безопасности языка за счёт отъёма у программистов «опасных» инструментов. Но в то же время легко привести примеры, в которых использование той или иной возможности из этого списка позволяет сделать код логичнее/элегантнее/эффективнее.

В целом автоматическая сборка мусора, знакомый С-подобный синтаксис, безопасность и возможность работы собранного бинарника под любой платформой обусловили несомненный успех Java. В упомянутых в самом начале статьи рейтингах популярности языков Java также занимает лидирующие позиции и улучшает их от года к году.

### D

Несмотря на то, что в тех же рейтингах язык программирования D зачастую даже не попадает в десятку-двадцатку лидеров, было бы несправедливым не сказать хотя бы пару слов про этот интересный язык. Созданный Уолтером Брайтом как переосмысление C++, язык D появился приблизительно в одно время с C#. Основным мотивом в дизайне D стало объединение двух миров: производительность мира С/C++ и удобство managed мира.

D поддерживает полную совместимость с ABI (application binary interface) С/С++ программ, предоставляет указатели на данные и функции, позволяет делать вставки на ассемблере, и в то же время предоставляет такие вкусности из managed мира, как автоматическое управление памятью, вложенные функции и замыкания, плюс предлагает сильные возможности метапрограммирования и многое другое.

Одно время я серьёзно рассматривал использование D, как скриптового языка в наших приложениях, но молодость языка, а главным образом отсутствие на тот момент доступного для коммерческого использования backend-а, заставили отказаться от этой идеи.

Кроме того, ряд моментов в дизайне по прежнему сомнительны, а небезопасные указатели или ассемблерные вставки в значительной мере закрывают возможность использования D как безопасного скриптового языка.

# Jancy

### Мотивация

#### Мотивация №1, велосипедная

Одним из замечательных свойств российского образования было и, надеюсь, остаётся, привитие критического взгляда на вещи. Начиная со школы и, конечно, в институте, нас учили не брать слепо на веру то, что пишется в книжках или говорится лектором. Помню, на самой первой лекции по физике нам с серьёзным видом доказали что 2 = 5, и сразу после этого – что тупой угол равен острому. Затем лектор озвучил свою мысль: старайтесь не выучить то что сделано или сказано умными людьми, а осмыслить и понять почему сделано именно так, а не иначе. Возможно, вы найдёте ошибки в их рассуждениях. Возможно, вы найдёте способ сделать то же самое лучше, проще, элегантнее. А возможно, вы поймёте что лучше потратить свои усилия в другом месте, а не на почве улучшения чего то, что работает и так.

С одной стороны, я безумно благодарен за привитый тип мышления. С другой стороны, из меня в определённой мере получился изобретатель велосипедов – а это, конечно, не то звание, которым следует гордиться. В свою защиту добавлю, что другое правило, привитое в институте – правило изучать типоразмеры уже существующих велосипедов перед изготовлением своего собственного, – несомненно сократило количество сделанных мной велосипедов на порядок, а может и два.

Я занимался программированием в течение 20 последних лет, из них 15 – профессионально. В каждом из языков, на которых мне приходилось писать, мой внутренний изобретатель велосипедов непременно находил вещи, которые можно сделать лучше. Так получилось, что областью моей специализации стало системное программирование, в частности создание компиляторов. Поэтому неудивительно, что уже много лет у меня были свои игрушечные языки как полигоны для экспериментов. В свете моего признания в любви к C, также неудивительно что синтаксис в них чаще всего был C-подобным.

До недавнего времени названия у этих языков не было – несмотря на то что предыдущие версии (а скорее, инкарнации, т.к. при переходе к новой версии от старого кода, как правило, не осталавалось ни строчки) использовались в коммерческих проектах как scripting engines. Однако в какой то момент я решил что это не дело, и после сравнительно недолгого периода поисков и душевных метаний, остановился на имени Jancy.

Как несложно догадаться, Jancy это акроним: [in between] JAva aNd C. Можно сказать, что за основу была взята Java и я постарался привнести в неё те возможности, которые я люблю в C/C++. Либо наоборот, что за основу был взят C/C++ и расширен удачными решениями из Java. Плюс, конечно же, я добавил в язык фишки, ещё пока нигде не реализованные (или, если скромнее, то возможно и реализованные, но пока не получившие распространения в mainstream-языках) – фишки о которых я всегда мечтал и с которыми экспериментировал ранее в своих игрушечных языках.

Итак, должен признать, что одним из главных мотиваторов создания Jancy было насыщение моего внутреннего изобретателя велосипедов.

#### Мотивация №2, практическая

Однако было и практическое обоснование для начала полномасштабных работ. Несколько лет назад наша компания выпустила продукт под названием I/O Ninja, один из модулей которого должен был анализировать блоки бинарных данных и записывать в лог результат анализа. Хочется избежать несправедливых обвинений в рекламе, поэтому я намеренно не расписываю подробности функционала, и зачем вообще такое потребовалось. Вот надо было нам анализировать и генерировать массивы произвольных байтов, будем исходить отсюда. Так как анализ хотелось сделать максимально гибким, надо было вынести логику анализа либо в плагины, либо в скрипт. А так как в идеале мы хотели дать пользователям возможность дописывать свои собственные анализаторы/модифицировать существующие, скрипт являлся более правильным подходом (плагины означали бы необходимосить публикации SDK для плагинов, плюс наличия у клиентов средств разработки на С/C++ и людских ресурсов с необходимым уровнем квалификации, чтоб эту разработку вести)

Проблема состояла в том, что разбор бинарных данных в языке без указателей и возможности описания структур и объединений (struct/union) сводится к вытаскиванию байтов по фиксированным индексам из буфера и складыванию их в слова, что порождает нечитабельный и неподдерживаемый код. Для работы с бинарными данными (анализа или генерации) нужны структуры, указатели и адресная арифметика. Точка. Общепризнанных скриптовых языков для встраивания в C/C++ приложения с поддержкой указателей или какого то другого хитрого механизма для работы с бинарными данными не нашлось ни несколько лет назад, ни сейчас, в момент написания статьи. Значит, если они и существуют, то только в экспериментальных вариантах, и любой самодельный велосипед имеет не меньшее право на жизнь.

Первая версия I/O Ninja вышла-таки на архитектуре плагинов, вторая – на скриптах, написанных на прототипе Jancy (опасные указатели, управление памятью на основе подсчёта ссылок, рудиментарная поддержка классов и самописная виртуальная машина). Выход третьей версии на полноценной Jancy запланирован на этот год.

Этот продукт – практический мотиватор создания Jancy и прекрасный полигон для тестирования языка в реальных условиях.

#### Мотивация №3, философская

Перед тем как перейти к принципам дизайна языка Jancy, позволю себе ещё немного пофилософствовать. Я ни в коей мере не хочу сказать, что создания языка было «вынужденным». Что без него невозможно было реализовать всё то же самое на других языках. Можно было. И я прекрасно это понимаю.

Но так работает прогресс: большинство нововведений просто делают некоторый доступный уже человечеству процесс немного более удобным, эффективным, лёгким и т.д. Можно было бы вообще всё написать на старом добром С. И даже прямо на ассемблере! Как можно и перемещаться из пункта А в пункт Б на лошадях. Или освещать помещение керосиновой лампой. Но любой процесс можно чуть-чуть улучшить. А потом ещё чуть-чуть улучшить. Из таких маленьких улучшений, каждое из которых отнюдь не претендует на революционность и не открывает по-настоящему новых возможностей (то же самое ведь можно было сделать и раньше, только по-другому!) – складываются большие: такие, которые позволяют сделать что-то, ранее принципиально невозможное.

Этими инкрементальными НЕ-революционными (без всякого сарказма) улучшениями мы и пытаемся заниматься.

### Дизайн языка

При проектировании языка Jancy за основную модель использования было взято использование как скриптового языка в хостовом приложении на C/C++. В долгосрочных перспективах, конечно, хочется надеяться на использование Jancy как managed языка общего назначения, заточенного под системное программирование.

Основными целями, преследовавшимися при дизайне Jancy, являлись:

* Использование как скриптового языка в хостовом приложении на C/C++
* Императивный объектно-ориентированный С-подобный синтаксис
* Безопасные указатели и адресная арифметика
* Автоматическое управление памятью через сборку мусора

Помимо этого, Jancy содержит целый ряд нововведений/усовершенствованний/особенностей разной степени значимости, сводная таблица которых представлена ниже.

* Полная ABI-совместимость с хостовым приложением на C/C++
  + Правильно описав типы данных в хостовом приложении и jancy скрипте, можно напрямую передавать данные через аргументы и возвращаемые значения, без необходимости явного заталкивания на стек виртуальной машины, упаковки в variant-подобные контейнеры и т.д. Поддерживаются
    - Все примитивные типы C/C++ (а также целочисленные типы с обратным порядком следования байтов)
    - Структуры (с произвольным фактором упаковки)
    - Объединения
    - Массивы
    - С/С++ указатели на данные и функции
* Классы (представляют собой особый тип данных со специальным заголовком)
  + Поддерживаются стандартные для современных языков возможности, такие как
    - конструкторы и деструкторы (статические и нестатические)
    - инициализиция полей по месту объявления
    - перегрузка операторов
    - виртуальные функции
  + Простое множественное наследование (без виртуальной модели C++)
  + Реализация методов может располагаться как по месту объявления, так и быть вынесена за пределы класса (как в C++)
  + Поддерживаются т.н. преконструкторы (preconstruct) – функции, которые будут автоматически выполнены перед каждым из перегруженных конструкторов. Аналог инициализирующих блоков в Java
* RAII, а также полный контроль над размещением данных
  + Возможность явно выделять память под объекты со стека и использовать парадигму RAII (как в C++)
  + Возможность выделять память под множество объектов сразу (объявляя поля-эклемпляры классов как в C++), без необходимости явного вызова ‘new’ под каждый объект.
  + Встроенная поддержка потоковых переменных (thread)
  + Ручное управление паматью с помощью heapu new/delete
  + Спецификаторы размещения: static, heap, heapu, stack, thread – могут быть указаны как при объявлении полей и переменных, так и при использовании оператора new
* Указатели на данные и классы
  + Const-корректность
  + Безопасная адресная арифметика
  + Встроенная поддержка слабые указателей на классы
* Свойства (без ложной скромности – самая полнофункциональная на текущий момент реализация)
  + Естественная форма объявления простых свойств
  + Полная форма для объявления свойств произвольной сложности (с перегруженными сеттерами, полями данных, вспомогательными методами и тд)
  + Индексируемые свойства (свойства, имеющие семантику массивов)
  + Autoget-свойства избавляют от рутинной необходимости прописывать геттеры-дегенераты
  + Bindable-свойства (свойства с событием onChange, т.е. оповещающие о своём изменении)
  + Bindable-данные для автоматической генерации свойств, единственным назначением которых является отслеживание изменений.
* Указатели на функции и свойства
  + Указатели на функции и свойства умеют захватывать значения аргументов
  + Автоматическое создание функций-переходников при несоответствии сигнатур аргументов
  + Слабые указатели на функции и свойства
  + Оператор планировки (schedule): создание указателя на функцию, которая будет вызвана в нужном контексте
* Мультикасты и события
  + Мультикасты позволяют накопить массив указателей на функции и затем вызвать их все сразу
  + События это особый вид указателя на мультикаст, который позволяет только подписаться/отписаться, но не вызвать
  + Два вида преобразовния мультикаста к указателю на функцию: live и snapshot
  + Встроенная поддержка слабых мультикастов (то есть тех, для которых не нужен явный вызов Unsubscribe)
* Поддержка реакционного программирования
  + Особым образом объявленные выражения автоматически пересчитываются при изменении bindable-свойств, входящих в правую часть
  + Специальные конструкции для назначения блоков кода как обработчиков изменения заданных bindable-свойств (без необходимости явной подписки/отписки).
* Специальная модель обработки исключений
  + Исключения в Jancy это просто синтаксический сахар над стандартной моделью кодов ошибок
  + Как следствие, модель исключений Jancy совместима с любым внешним языком
  + catch и finally могут быть объявлены в любом scope
  + При вызове одной и той же функции можно применять как проверку кода ошибки, так и исключения – в зависимости от того, какой подход более удобен в данном конкретном случае.
* Дуальная модель доступа
  + В Jancy только два спецификатора доступа: public и protected. Для каждого пространства имён весь остальной мир распадается на «своих» и «чужих». Свои имеют доступ ко всему, чужие – только к public
  + Дуальный модификатор constd позволяет объявить тип данных, который выглядит как const для «чужих», и в то же время доступен для изменения «своими». Предназначен для избавления от необходимости прописывать свойства-дегенераты, единственное назначение которых это предоставление доступа только-на-чтение.
  + Дуальный модификатор eventd для объявления событий или указателей на события. «Чужие» могут только подписатья/отписаться (семантика «event»), свои же должны уметь вызвать всех подписчиков (семантика «multicast»)
* Специальные литералы
  + Hex-литералы для удобного описания блоков константных двоичных данных
  + Форматирующие литералы позволяют динамически генерировать строки с подстановкой значений выражений, используемых внутри литерала (perl-style formatting)
* Другое
  + once/thread once
  + Полноценные curly-инициализаторы
  + break-n/continue-n для управления внешними циклами
  + basetype-n для удобной адресации базовых типов
  + enumf для описания флаговых перечислений
  + extend – расширения типов после объявления
  + alias – псевдонимы выражений
  + Scopes-in-switch
  + Конструкторы/деструкторы у юнитов компиляции

### Архитектура компилятора

Для генерации лексического анализатора используется Ragel – универсальный компилятор конечных автоматов, который замечательно подходит для создания лексеров в силу удобства входного языка и эффективности выходного кода – затыкающей, кстати говоря, за пояс все аналоги.

В качестве backend используется LLVM, что позволяет задействовать отлаженный оптимизатор и кодогенериратор сразу для широкого спектра платформ, а также существенно упрощает обеспечение совместимости с ABI хостового C/C++ приложения. Использование LLVM, а не рукописной виртуальной машины, или же рукописного генератора целевого кода – это решение которое не вызывало вопросов с самого начала.

А вот для синтаксического анализа я таки не удержался и написал свой ~~велосипед~~ генератор табличных нисходящих LL (k) анализаторов. Чем меня не устраивали ANTLR, Coco, Yacc/Bison, Lemon и другие безусловно уважаемые и проверенные в бою парсер-генераторы, это тема для отдельного и интересного разговора. Пока же я просто хотел сказать что в качестве синтаксического анализатора в компиляторе Jancy используется не ручной рекурсивный спуск, а сгенерированный парсер, а значит имеется всегда релевантная BNF грамматика которую можно распечатать, обсудить и с лёгкостью модифицировать. Грамматика Jancy относится к классу контескстно-зависимых LL (2).

Выходом парсера является сразу LLVM IR без промежуточной генерации AST: в нисходящих парсерах удобно проводить семантический анализ и генерировать код прямо по ходу разбора.

Модель взаимодействия лексера и парсера подсмотрена в Lemon: парсер Jancy НЕ вызывает лексер; вместо этого имеется внешний цикл выборки токенов, каждый из которых скармливается табличному парсеру. Данная модель позволяет разбирать неполные юниты компиляции, останавливать и возобновлять процесс разбора и т.д., плюс, как и в любом табличном парсере, память под аттрибуты правил выделяется не на стеке (что чревато stack overflow или искусственными ограничениями на уровень вложенности), а в куче.

Синтаксический/семантический анализ двух-, а в некоторых правилах трёх - проходный (что, однако, не означает повторного запуска лексера). Второй проход сделан для возможности использовать глобальные типы и данные до парсинга их объявления (которое может находиться вообще в другом юните компиляции). Третий проход используется для предварительного рассчёта реакторов.