JANCY: Возвращение Указателей

Contents

[Введение 2](#_Toc405907002)

[Принципы дизайна 2](#_Toc405907003)

[Ключевые возможности 2](#_Toc405907004)

[Другие значимые особенности 2](#_Toc405907005)

[Мотивация 3](#_Toc405907006)

[Мотивация №1, велосипедная 3](#_Toc405907007)

[Мотивация №2, практическая 4](#_Toc405907008)

[Мотивация №3, философская 4](#_Toc405907009)

[Огласите весь список, пожалуйста! 5](#_Toc405907010)

[ABI-совместимость с C/C++ 5](#_Toc405907011)

[Классы 6](#_Toc405907012)

[RAII и контроль над размещением данных 12](#_Toc405907013)

[Безопасные указатели на данные 13](#_Toc405907014)

[Указатели на функции 19](#_Toc405907015)

[Мультикасты и события 22](#_Toc405907016)

[Свойства 25](#_Toc405907017)

[Реактивное программирование 29](#_Toc405907018)

[Исключения 31](#_Toc405907019)

[Строковые литералы 35](#_Toc405907020)

[Дуальные модификаторы 36](#_Toc405907021)

[Перечисления 38](#_Toc405907022)

[Ленивая инициализация 40](#_Toc405907023)

[Конструкции break-n, continue-n 40](#_Toc405907024)

[Добавление методов в существующие типы 41](#_Toc405907025)

[Области видимости в switch 42](#_Toc405907026)

[Curly-инициализаторы 43](#_Toc405907027)

[Модуль-конструкторы и деструкторы 44](#_Toc405907028)

[Архитектура компилятора 45](#_Toc405907029)

[Заключение и статус 46](#_Toc405907030)

[Внешние ссылки и список литературы 46](#_Toc405907031)

# Введение

Что, ещё один язык программирования? Зачем? Их что, недостаточно много?

Поверьте, я прекрасно представляю себе все аргументы против разработки нового языка программирования. Но всё дело в том, что Jancy был создан НЕ для исправления печально известного «фатального недостатка»[[1]](#footnote-1) других языков – хотя, конечно, было бы лукавством отрицать, что творческие устремления играли немаловажную роль в развитии проекта.

Если максимально кратко сформулировать практическую сторону нашей (как компании) мотивации, то она будет звучать так: мы искали скриптовый движок с поддержкой структур, указателей на данные и безопасной адресной арифметикой для встраивания в приложения на C++. Не нашли! И поэтому сделали что-то своё, в чём есть и всё вышеперечисленное, и многое другое.

Знакомьтесь – это Jancy.

## Принципы дизайна

* Объектно-ориентированный язык для IO и UI программирования с C-подобным синтаксисом;
* Основная модель использования – скриптовый движок для встраивания в приложения на C++;
* ABI (application-binary-interface) совместимость с C/C++;
* Автоматическое управление памятью через точную сборку мусора (accurate garbage collection);
* LLVM в качестве backend.

## Ключевые возможности

* Безопасные указатели и адресная арифметика;
* Свойства (properties) – самая полная реализация;
* Мультикасты (multicasts) и события (events), включая слабые, от которых необязательно отписываться;
* Реактивное программирование (reactive programming) – одна из первых реализаций в императивном языке программирования общего назначения;
* Исключения как синтаксический сахар над моделью кодов ошибок.

## Другие значимые особенности

* Множественное наследование;
* Const-корректность;
* Поддержка парадигмы RAII (resource-acquisition-is-initialization);
* Локальная память потоков (thread local storage);
* Слабые указатели, которые не влияют на время жизни объектов;
* Частичное применение (partial application) для функций и свойств;
* Оператор планирования (schedule operator) для создания указателей на функции, которые гарантировано будут вызваны в заданном окружении (например, в нужном рабочем потоке);
* Перечисления для битовых флагов (bitflag enums);
* Perl-подобное форматирование строк;
* …и многое другое – нет, правда! ;)

А теперь, с вашего позволения, немного замедлюсь и обернусь назад.

# Мотивация

## Мотивация №1, велосипедная

Одним из замечательных свойств российского образования было и, надеюсь, остаётся привитие критического взгляда на вещи. Начиная со школы нас учили не брать слепо на веру то, что пишется в книжках или говорится лектором. Помню, на одной из самых первой лекций в лицее нам с серьёзным видом доказали, что тупой угол равен острому, а затем лектор озвучил озадаченному коллективу свою мысль: старайтесь не запомнить то, что сказано или сделано умными людьми (имел он в виду, очевидно, себя), а осмыслить и понять, почему сделано именно так, а не иначе. Возможно, вы найдёте ошибки в их рассуждениях. Возможно, найдёте способ сделать то же самое, но лучше или проще. Хотя может быть и наоборот: вы поймёте, что лучше приложить свои усилия в другой области, а не на почве улучшения того, что работает и так.

Я глубоко благодарен за привитый тип мышления. Но должен признать, что во многом благодаря привычке пропускать всё через голову и вместо бездумного запоминания «проходить путь первооткрывателя», из меня в определённой мере получился неисправимый изобретатель велосипедов, а это не то звание, которым следует гордиться. Хотя, конечно, другое правило, привитое уже в институте – правило изучать типоразмеры существующих велосипедов перед изготовлением своего собственного – несомненно сократило количество сделанных мной велосипедов не менее, чем на порядок.

Я занимаюсь программированием в течение 20 последних лет, из них 15 – профессионально. В каждом из языков, на которых мне доводилось писать, мой внутренний изобретатель велосипедов непременно искал и находил вещи, которые можно добавить, убрать или сделать лучше. А так как областью моей специализации стало системное программирование, и в частности, создание компиляторов, то неудивительно, что уже много лет у меня были свои игрушечные языки. Я использовал их как полигоны для экспериментов.

До недавнего времени названия ни у одного из этих разрозненных языков не было – это несмотря на то, что предыдущие версии (а скорее, реинкарнации, т.к. при переходе к новой версии от старого кода, как правило, не оставалось ни строчки) уже использовались в коммерческих проектах в качестве DSL (domain-specific languages). В определённый момент я решил, что это не дело, свёл всё воедино и, после сравнительно недолгой лингвистической одиссеи, нашёл нужное имя – Jancy.

Как несложно догадаться, Jancy – это акроним: [in between] JAva aNd C. Можно сказать, что за основу был взят гибрид Java и C/C++, вобравший в себя любимые мной в обоих языках концепции, возможности и элементы синтаксиса. На этом фундаменте были достроены дополнительные, ещё пока нигде не реализованные (а если и реализованные, то не получившие достаточного распространения) возможности, о которых я всегда мечтал и которые успешно прошли полигоны моих игрушечных языков.

Итак, надо 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 IO Ninja, который, в частности, выполнял анализ сетевых (IP, TCP, и т.д.) пакетов и должен был записывать в журнал результаты анализа. Хочется избежать несправедливых обвинений в рекламе, поэтому я намеренно не расписываю подробности функционала, и зачем вообще такое потребовалось. Будем исходить из предположения, что нам надо было анализировать и генерировать бинарные пакеты.

Поскольку анализ хотелось сделать максимально гибким, необходимо было вынести логику анализа либо в плагины, либо в скрипт. А так как в конечном итоге мы хотели дать пользователям возможность дописывать свои собственные анализаторы и модифицировать существующие, скрипт выглядел как более правильный подход: плагины означали бы необходимость публикации некоего SDK, плюс наличия у клиентов средств разработки на С/C++, а также людских ресурсов с необходимым уровнем квалификации, чтобы эту разработку вести.

Основная проблема состояла в том, что разбор бинарных данных в языке без указателей и возможности описания структур (struct) и объединений (union) сводится по большому счёту к извлечению байтов по фиксированным смещениям из буфера и складыванию их в слова, что порождает нечитабельный и неподдерживаемый код. Для работы с бинарными данными – анализа или генерации – лучше всего подходят структуры, указатели и адресная арифметика. Кроме этого, в идеале хотелось бы также иметь возможность с достаточной легкостью импортировать описания структур на языке C (т.к. несложно найти С-описания заголовков подавляющего большинства существующих коммуникационных протоколов).

Общепризнанных скриптовых языков с поддержкой указателей и адресной арифметики, при этом подходящих для встраивания в приложения на C/C++, не нашлось тогда, и нет даже 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 IO Ninja вышла-таки на архитектуре плагинов; вторая – на скриптах, написанных на прототипе Jancy (небезопасные указатели, управление памятью на основе подсчёта ссылок, рудиментарная поддержка классов и самописная виртуальная машина). Последняя, третья версия IO Ninja основана на полноценном языке Jancy.

IO Ninja – это практический мотиватор создания Jancy и прекрасный полигон для тестирования языка в реальных условиях.

## Мотивация №3, философская

Перед тем как перейти к техническим подробностям и полному списку возможностей языка Jancy, позволю себе ещё немного пофилософствовать. Я ни в коей мере не хочу сказать, что создание языка было «вынужденным» и без него невозможно было бы реализовать всё то же самое на других языках или технологиях. Можно было. Альтернативный путь есть всегда. Более того, использование существующих технологий наверняка упростило и убыстрило бы цикл разработки, а значит, и получение конечного результата (выпуск IO Ninja на рынок).

Но так уж работает прогресс: редко какие из нововведений делают возможным недоступное ранее. Подавляющее же большинство просто чуть-чуть улучшают уже доступный человечеству процесс – причём далеко не всегда очевидно, стоило ли достигнутое улучшение затраченных усилий.

Однако со временем из множества этих маленьких и нереволюционных улучшений инкрементально складываются большие – такие, которые уже позволяют сделать что-то, что ранее было принципиально невозможно, и под которыми все и понимают слово «прогресс».

Нам, как компании, и мне лично приятно осознавать, что, занимаясь нашими маленькими инкрементальными улучшениями, мы вносим свой вклад в прогресс языков программирования в частности и информационных технологий в общем.

Ну а теперь…

# Огласите весь список, пожалуйста!

В данном разделе будут кратко описаны интересные возможности и особенности языка Jancy. Но сначала, по традиции, надо сказать миру «здравствуй!»:

int main ()

{

printf ("hello world!\n");

return 0;

}

Здесь всё ясно без дополнительных комментариев, не правда ли? При проектировании грамматики Jancy я старался по возможности сохранять привычный синтаксис дабы человек, хорошо знающий C++ или Java, мог читать и понимать подавляющую часть Jancy-кода без подготовки. Насколько это удалось – судить вам.

## ABI-совместимость с C/C++

Jancy-скрипты JIT-компилируются и могут быть напрямую вызваны из хостового приложения на С++. Jancy поддерживает все основные модели вызовов (calling conventions) и совместим с компиляторами MSVC и GCC в плане размещения полей структур (structs) и объединений (unions).

Это означает, что после правильного описания типов данных и прототипов функций в скриптах Jancy и приложении на C++ становится возможно передавать данные естественным образом, через аргументы функций и возвращаемые значения, без необходимости явного заталкивания на стек виртуальной машины или упаковки в variant-подобные контейнеры.

Вот типы, совместимые с Jancy:

* Все примитивные типы из C/C++;
* Целые типы с обратным порядком следования байтов, или *bigendians*;
* Структуры (structs) и объединения (unions)с произвольным фактором упаковки;
* Битовые поля (bitfields) в структурах и объединениях;
* Массивы (arrays);
* Указатели на функции.

Jancy поддерживает следующие модели вызовов (calling conventions):

* cdecl (Microsoft/GCC);
* stdcall (Microsoft/GCC);
* Microsoft x64;
* System V.

Всё вышеперечисленное значительно упрощает и делает гораздо более эффективным взаимодействие между хостовым приложением и скриптом.

Другим немаловажным следствием высокой степени бинарной совместимости между Jancy и C/C++ является возможность копировать из общедоступных источников (таких как Linux, React OS или других проектов с открытым исходным кодом) и использовать – возможно, после косметических исправлений – определения заголовков коммуникационных протоколов на языке C.

## Классы

Классы в Jancy представляют собой особый тип данных, в котором явно объявленные разработчиком поля предваряются специальным неявным заголовком с мета-данными. В них входят тип, указатель на таблицу виртуальных функций, указатель на корневой объект, флаги, используемые при сборке мусора, и некоторые другие поля. Из-за этих метаданных размещение полей в классе Jancy будет отличаться от такового в идентичном на уровне исходных кодов классе C++, поэтому для ABI-совместимости классы и экземпляры классов требуют специального объявления в хостовом приложении на C++.

В отличие от C++, объявление класса (а также любых других именованных типов) в Jancy не является спецификатором типа. С одной стороны это означает, что нельзя объявить класс и сразу же использовать его для создания поля или переменной; зато с другой стороны, это позволяет опускать точку с запятой после объявления без конфликтов в грамматике.

В Jancy есть всего два спецификатора доступа: public и protected (более подробно о том, почему так сделано, читайте в разделе «дуальные модификаторы»). Спецификаторы доступа могут быть указаны как в стиле C++, так и в стиле Java, при этом, в отличие от подавляющего большинства объектно-ориентированных языков, умолчальным типом доступа является public:

class C1

{

int m\_x; // public by default

protected: // C++-style of access specification

int m\_y;

public int m\_z; // Java-style of access specification

// ...

}

Реализация методов может располагаться как по месту объявления, так и быть вынесена за пределы класса, как в C++. При этом в Jancy, вместо применяемого в С++ оператора разрешения контекста “::”, используется просто оператор взятия члена “.”:

class C1

{

foo () // in-class method implementation

{

// ...

}

bar ();

}

// out-of-class method implementation

C1.bar ()

{

// ...

}

Jancy поддерживает статические и нестатические конструкторы, деструкторы и инициализацию полей по месту объявления. Синтаксис объявления конструкторов и деструкторов отличается от традиционного подхода с использованием имени типа – вместо этого применяются ключевые слова construct и destruct.

Поддерживаются также так называемые преконструкторы (preconstruct) – функции, которые будут автоматически выполнены перед каждым из перегруженных конструкторов (аналог инициализирующих блоков в Java):

class C1

{

int m\_x = 100; // in-place initializer

static construct ();

static destruct ();

preconstruct (); // will be called before every overloaded constructor

construct ();

construct (int x);

construct (double x);

destruct ();

// ...

}

В Jancy есть указатели и нет разделения на типы значений (value types) и ссылочные типы (reference types). Что *есть* указатель, то должно *выглядеть* как указатель. Другими словами, объявляя поле или переменную типа класса, вы даёте компилятору инструкцию создать экземпляр класса, как в C++ (а не указатель на класс, как в Java). Чтобы объявить указатель на класс, надо явно использовать тип «указатель на класс»:

class C1

{

construct (int x);

foo ();

// ...

}

bar ()

{

C1 c (100);

c.foo ();

C1\* p = new C1 (200); // use class pointer type explicitly

// ...

}

Поддерживается перегрузка операторов, как и в C++. Унарные и бинарные операторы, операторы вызова, индексации и приведения могут быть перегружены в любых именованных типах (кроме enum):

class C1

{

operator += (int d) // overloaded '+=' operator

// ...

}

foo ()

{

C1 c;

c += 10;

// ...

}

Jancy использует простую модель множественного наследования, без печально известного виртуального наследования C++. Простое множественное наследование означает, что если несколько родительских типов имеют общих предков, то в дочернем типе будут несколько экземпляров этого общего предка, а приоритет наследования виртуальных методов идёт в порядке объявления родительских типов.

Простое множественное наследование – это прекрасный инструмент для создания общей (разделяемой) реализации, который был, как мне кажется, совершенно незаслуженно отброшен проектировщиками большинства управляемых языков из-за раздутой diamond problem (которая на практике практически никогда не причиняет серьёзных неудобств). Впрочем, меньше всего я хочу вызвать обострения каких бы то ни было «священных войн», поэтому предлагаю просто остановиться на том, что в Jancy принято простое множественное наследование.

Для объявления виртуальных методов используются ключевые слова abstract, virtual и override. Наследовать классы можно как от других классов, так и от структур и даже объединений:

class I1

{

abstract foo ();

}

class C1: I1

{

override foo ()

{

// ...

}

}

class I2

{

abstract bar ();

abstract baz ();

}

class C2: I2

{

override bar ()

{

// ...

}

}

struct S1

{

int m\_x;

// ...

}

class C3:

C1,

C2,

S1 // it's OK to inherit from structs and even unions

{

override baz ()

{

// ...

}

}

Jancy предлагает ключевые слова basetype и basetype-n для удобной адресации базовых типов, например, из конструкторов или для вызова родительской реализации переопределённого виртуального или же перегруженного метода:

class Base1

{

construct (int x);

virtual foo ();

// ...

}

class Base2

{

construct (int x);

// ...

}

class Derived:

Base1,

Base2

{

construct (int x)

{

basetype1.construct (x);

basetype2.construct (x);

// ...

}

override foo ()

{

basetype1.foo ();

// ...

}

}

Jancy предоставляет встроенную поддержку слабых указателей на классы, то есть указателей, которые не влияют на время жизни объекта. Основная (хотя и не единственная) область применения слабых указателей – это различные вариации паттерна «событие-подписчик».

Нельзя обращаться к объекту по слабому указателю. Слабый указатель можно только попытаться привести к обычному (сильному) указателю: если объект ещё жив, то будет возвращён указатель на него, и null в противном случае:

class C1

{

// ...

}

foo ()

{

C1\* c = new C1 (10);

C1 weak\* wc = c;

wc.m\_x = 20; // <-- error

c = null; // lose strong pointer

// ...

c = wc; // try to recover strong pointer

if (c)

{

// object is still alive

}

return 0;

}

В свете ориентированности Jancy на использование в качестве скриптового языка из приложения C++, особое значение приобретает возможность прятать от скрипта реализацию класса на C++. Скрипт должен видеть и уметь использовать интерфейс класса, не зная при этом деталей низкоуровневой реализации.

Jancy не требует реализации методов по месту объявления (поддерживается реализация вне класса, как в C++), что решает проблему с сокрытием кода функций: объявление класса на Jancy может содержать только объявления методов, а реализация этих методов может быть выполнена на C++. При этом совершенно необязательно делать эти методы виртуальными (избегая тем самым незначительных, но совершенно ненужных накладных расходов, связанных с косвенными вызовами).

А как быть с полями данных класса? Например, хостовое приложение может экспортировать в пространство имён скрипта классы виджетов для организации пользовательского интерфейса, реализация чего на C++ скорее всего будет содержать экземпляры QT, WTL, MFC или каких-то других «тяжёлых» классов, которые должны быть спрятаны от скриптов Jancy.

Jancy предлагает решение в виде непрозрачных (opaque) классов. Компилятор Jancy не имеет полных данных о строении непрозрачного класса и поэтому не может сгенерировать код для создания его экземпляра – будь то в виде переменной или поля.

Экземпляры непрозрачных классов могут быть созданы только на GC-heap с помощью оператора new и только если разработчик предоставит специальный метод operator new (возвращающий указатель на новый объект). Впрочем, разработчик может использовать любые иные способы передачи в скрипт указателей на уже созданные экземпляры непрозрачных классов:

opaque class Serial

{

uint\_t autoget property m\_baudRate;

SerialFlowControl autoget property m\_flowControl;

uint\_t autoget property m\_dataBits; // typically 5..8

SerialStopBits autoget property m\_stopBits;

SerialParity autoget property m\_parity;

Serial\* operator new ();

// ...

}

Реализация данного класса в C++может выглядеть как-то так:

class Serial: public jnc::IfaceHdr

{

public:

JNC\_API\_BEGIN\_CLASS ("io.Serial", ApiSlot\_io\_Serial)

JNC\_API\_AUTOGET\_PROPERTY ("m\_baudRate", &Serial::setBaudRate)

JNC\_API\_AUTOGET\_PROPERTY ("m\_flowControl", &Serial::setFlowControl)

JNC\_API\_AUTOGET\_PROPERTY ("m\_dataBits", &Serial::setDataBits)

JNC\_API\_AUTOGET\_PROPERTY ("m\_stopBits", &Serial::setStopBits)

JNC\_API\_AUTOGET\_PROPERTY ("m\_parity", &Serial::setParity)

JNC\_API\_OPERATOR\_NEW (&Serial::operatorNew)

// ...

JNC\_API\_END\_CLASS ()

public: // fields directly accessible from Jancy

uint\_t m\_baudRate;

axl::io::SerialFlowControl m\_flowControl;

uint\_t m\_dataBits;

axl::io::SerialStopBits m\_stopBits;

axl::io::SerialParity m\_parity;

protected: // opaque part invisible from Jancy

axl::io::Serial m\_serial;

mt::Lock m\_ioLock;

uint\_t m\_ioFlags;

IoThread m\_ioThread;

// ...

protected:

static

Serial\*

operatorNew ();

// ...

};

## RAII и контроль над размещением данных

Jancy предоставляет разработчику следующие спецификаторы размещения данных:

* static
* heap
* thread
* stack

Глобальные переменные могут иметь спецификаторы static (по умолчанию) и thread. Поля по умолчанию выделяются из родительского блока (т.е. объявляя поле, вы тем самым увеличиваете размер родительского типа на размер поля), а также static и thread:

class C1

{

int m\_memberField;

static int m\_staticField = 2;

thread int m\_threadField;

}

int g\_staticGlobal; // for global variables, default storage is 'static'

thread int g\_threadGlobal;

Локальные переменные могут иметь любой спецификатор размещения из вышеперечисленных. Умолчальное размещение локальных переменных зависит от типа переменной: в настоящий момент это heap для классов и stack для всего остального (здесь есть что улучшать, и, скорее всего, последующие версии Jancy будут обладать более хитрой логикой по выбору умолчального размещения локальных переменных).

Потоковые (thread) переменные не могут быть агрегатами (aggregate). Кроме того, инициализировать можно только локальные потоковые переменные (эти ограничения естественным образом вытекают из соображений «тяжести» сопутствующих накладных расходов):

thread int g\_threadGlobal2 = 10; // <-- error

thread int g\_threadGlobal3 [10]; // <-- error

foo ()

{

int x; // for local non-class variables, default storage is 'stack'

C1 cl; // for local class variables, default storage is 'heap'

static int s = 100;

thread int t = 200; // for local 'thread' variables, initialization is permitted

heap char a [256]; // use 'heap' storage to allocate large local variables on GC-heap

}

Как было сказано выше, в Jancy объявление переменной или поля типа класса не превращается неявно в объявление указателя на этот класс. Поэтому в Jancy возможно разделять блок памяти между несколькими экземплярами классов так же, как и в C++ – просто объявляя поля типа класса, без необходимости явного вызова new под каждый объект:

class C1

{

// ...

}

class C2

{

// ...

C1 m\_classField1; // allocated as part of C2 layout

C1 m\_classField2;

}

Поддержка создания классов на стеке означает, что в Jancy можно использовать парадигму RAII (resource-acquisition-is-initialization) так же, как и в C++. Стековые экземпляры классов будут разрушены в момент выхода из области видимости, в которой они были объявлены, обеспечивая тем самым возможность детерминистического освобождения захваченных ресурсов:

class Resource

{

construct (int id)

{

// acquire resource

}

destruct ()

{

// release resource

}

}

foo (int x)

{

stack Resource resource (1);

if (x >= 0)

{

stack Resource resource2 (2);

// ...

} // resource2 is released

// ...

return 0; // resource is released

}

## Безопасные указатели на данные

Как ясно из названия статьи и раздела, посвящённого практической мотивации, безопасные указатели и адресная арифметика – это одни из основных достоинств Jancy.

Указатели всегда справедливо считались небезопасным инструментом, способным в не самом плохом варианте вызвать аварийный останов программы, а в худшем случае ещё и успеть перед этим молча повредить данные пользователя. Несмотря на опасность их использования, полностью отказаться от указателей просто невозможно. Указатели есть даже в безопасных языках без указателей, таких как Java или Basic – это указатели на интерфейсы.

Действительно, ограничением доступных разработчику видов указателей и операций над ними можно значительно обезопасить язык, упростить обработку неблагоприятных ситуаций во время исполнения (runtime) и даже отлавливать некорректные операции в момент компиляции при помощи статического анализа.

Когда же в игру вступает адресная арифметика, полностью переложить анализ на этап компиляции просто невозможно. Обращения по невалидным указателям могут и будут случаться во время исполнения. Поэтому прежде всего оговорим, что понимается под «безопасными указателями».

Безопасными будем называть указатели, проведение над которыми разрешенных операций не может вызвать крах процесса и/или повреждение данных пользователя. Любая некорректная операция с безопасным указателем должна быть поймана системой времени исполнения (runtime). Что с этим делать дальше и как обрабатывать подобные исключения решать разработчику: может быть, выдать сообщение пользователю, записать в журнал и завершить или перезапустить приложение (это значительно лучше, чем дать испортить данные!); может быть, перезапустить только песочницу со скриптами и попытаться продолжить работу, начиная с последней успешной транзакции; может быть что-то третье.

Для проверки корректности операций безопасные указатели в Jancy содержат валидатор – специальную структуру мета-данных, в которой лежат разрешённый диапазон адресов, тип данных, а также целочисленный уровень вложенности (scope level).

Jancy runtime не допускает косвенного обращения по указателю, не прошедшему тест на принадлежность разрешённому диапазону. Упреждая возможный вопрос, надо признать, что да, данный механизм не бесплатен и действительно выливается в определённые накладные расходы во время исполнения. Но во-первых, даже в самом наивном варианте, без каких-либо оптимизаций, два целочисленных сравнения на косвенное обращение – это не так страшно, особенно принимая во внимание JIT-компиляцию и тот факт, что Jancy – это прежде всего скриптовый язык (по крайней мере на данном этапе). Во-вторых, в дальнейшем с помощью статического анализа можно будет избавиться от многих ненужных проверок ещё на этапе компиляции. Ну и в-третьих, для критических по производительности участков кода уже сейчас можно использовать небезопасные (тонкие, thin) указатели без валидаторов – проверки при операциях с тонкими указателями не производятся.

Проверки допустимого диапазона адресов производятся как в случае явного использования указателя…

foo (

char\* p,

size\_t size

)

{

p += size;

\*p = 10; // bang!!

}

…так и в случае использования оператора индексации:

foo (size\_t i)

{

static int a [] = { 10, 20, 30 };

int x = a [i]; // bang!!

}

bar ()

{

foo (3);

}

Несложно видеть, что в случае указателей на данные, выделенные со стека, одной проверки допустимого диапазона недостаточно:

foo ()

{

int\* p;

{

int a = 10;

p = &a;

}

// ...

\*p = 20; // bang!!

}

Указатель из вышеприведённого примера пройдёт проверку на попадание в допустимый диапазон, но косвенное обращение по такому указателю ни к чему хорошему не приведёт. Другими словами, указатель сначала был валидным, а затем – вообще без какой-либо модификации – стал невалидным. Буквально испортился от времени!

Таким образом, помимо проверок на диапазон, необходимо также предотвратить любую утечку адреса за границы времени жизни этого адреса. К сожалению, одним только статическим анализом на этапе компиляции не обойтись и тут:

int g\_global;

class C1

{

abstract int\* foo (int\* p);

}

class C2: C1

{

override int\* foo (int\* p)

{

return &g\_global;

}

}

class C3: C1

{

override int\* foo (int\* p)

{

return p;

}

}

int\* foo (C1\* c)

{

int a;

return c.foo (&a); // bang or no bang?

}

Для решения проблемы в самом общем случае Jancy runtime поддерживает потоковую переменную, которая содержит целочисленный уровень вложенности и обновляет её при вызовах функций. Всем стековым адресам назначается этот уровень вложенности, начиная с 2 и выше (в зависимости от глубины области видимости и положения в стеке вызовов). Всем статическим и GC-heap адресам назначается 0, а потоковым адресам – 1.

Jancy runtime не даёт сохранить адрес с более высоким уровнем вложенности в локации с более низким уровнем вложенности. Тут придётся повторить сентенцию про сопутствующие накладные расходы, укороченным вариантом которой будет: да, это не бесплатно, но нет, это не так страшно. Кроме того, в данном случае статический анализ (который несомненно будет реализован в дальнейшем), сможет выявить гораздо более значительную часть ошибок на этапе компиляции, чем это было бы возможно с проверками диапазонов:

foo ()

{

int\* p;

int\*\* pp = &p;

{

int x;

int\* p2 = &x; // OK

p = &x; // <-- error

\*pp = &x; // <-- error

}

}

С понятием указателя рука об руку идёт понятие const-корректности. К сожалению, проектировщики большинства современных языков решили полностью отказаться от поддержки этого замечательного инструмента, позволяющего отлавливать целые классы логических ошибок на этапе компиляции и проектировать более дуракобезопасные библиотеки.

В Jancy как языке, одним из столпов которого являются указатели, const-корректность реализована полностью. Там, где необходимо обеспечить гарантию неизменности значений, доступных через указатель, указатель требуется снабдить модификатором const – как в C++. Однако, в отличие от C++, Jancy не даёт жульничать и приводить const-указатели к не-const-указателям:

transpose (

Point\* dst,

Point const\* src

)

{

int x = src.m\_x;

dst.m\_x = src.m\_y;

dst.m\_y = x;

src.m\_x = 0; // <-- error

Point\* p2 = (Point\*) src; // <-- error

}

Что касается const-корректности в отношении методов и полей, то Jancy и тут использует синтаксис C++:

class C1

{

int m\_field;

mutable int m\_mutableField;

foo () const;

bar ();

}

baz (C1 const\* p)

{

p.foo ();

p.m\_mutableField = 100;

p.m\_field = 200; // <-- error

p.bar (); // <-- error

}

Вернёмся к вопросу безопасности указателей. Jancy обеспечивает безопасность операций с указателями с помощью валидаторов, но кто обеспечивает безопасность самих валидаторов?

Что если мы создадим указатель на указатель, приведём его к указателю на char и затем байт за байтом затрём валидатор мусором? Или наоборот, подготовим буфер с «фальшивым» валидатором и затем приведём указатель на этот буфер к указателю на указатель – опять таки, валидатор будет, как это говорится в шпионских фильмах, скомпрометирован.

Такая опасность действительно есть. Для решения проблемы Jancy делит все типы на категории POD (plain-old-data) и не-POD. Понятие POD в Jancy отличается от аналогичного в C++ – возможно, в этой связи стоило придумать новый термин, чтобы избежать путаницы, но в итоге я решил не плодить новые сокращения. Кроме того, мне кажется, что POD в Jancy гораздо точнее отражает смысл понятия “plain-old-data”.

В Jancy POD – это данные без мета-данных. Их можно смело побайтно копировать и модифицировать и при этом ничего не сломать. Агрегация POD данных, будь то включения полей, наследование (тут отличие от C++) или объединение в массивы – тоже приводит к POD. Всё, что содержит мета-данные, а именно классы, безопасные указатели на данные и их любые агрегаты – это не-POD.

Компилятор Jancy разрешает приведения не-POD типов тогда и только тогда, когда это безопасно, то есть в результате приведения не появляется возможность разрушить или подменить мета-данные. Для ситуаций, в которых на этапе компиляции это неизвестно (например, мы производим приведение к дочернему типу, так называемый downcast) – существует специальный оператор динамического приведения. Оператор динамического приведения компилируется в вызов встроенной функции, которая возвращает указатель на запрошенный тип, или null, если приведение невозможно.

Для примера подготовим тестовые типы, которые мы будем приводить друг к другу:

struct A

{

int m\_a;

}

struct B

{

int m\_b;

}

struct C: A, B

{

int m\_c;

}

struct D: C

{

char const\* m\_s;

}

Здесь A, B, C – это POD (причём последний тип не был бы POD в C++), D – не POD, т.к. этот тип содержит мета-данные в виде валидатора указателя m\_s. Теперь рассмотрим возможные операции приведения.

Приведения к родительским типам (upcast) всегда разрешены и не требуют явного оператора приведения ни для POD, ни для не-POD:

foo (D\* d)

{

C\* c = d;

A\* a = с;

}

POD-типы могут быть произвольно приведены друг к другу с помощью оператора приведения:

bar (B\* b)

{

char\* p = (char\*) b;

C\* c = (C\*) b; // <-- unlike C++ no pointer shift

}

Приведения от POD-типов к не-POD-типам разрешены только в случае результирующего константного указателя:

foo (D\* d)

{

char\* p = (char\*) d; // <-- error

char const\* p2 = (char const\*) d; // OK

}

Приведение к дочерним типам (downcast) возможно с помощью оператора динамического приведения:

bar (B\* b)

{

D\* d = dynamic (D\*) b;

A\* a = dynamic (A\*) b; // not a downcast, but still OK

}

Динамическое приведение возможно благодаря содержащемуся в указателе валидатору, а значит и информации о типе. Помимо динамического приведения, Jancy также предлагает операцию динамического определения размера, который доступен из того же валидатора. В определенных ситуациях это является очень удобным инструментом:

foo (int\* a)

{

size\_t size = dynamic sizeof (\*a);

size\_t count = dynamic countof (\*a);

}

В текущей версии Jancy не реализована автоматическая защита валидаторов в многопоточных приложениях, так что теоретически валидаторы-таки могут быть разрушены при «умелом» использовании гонки между потоками (race conditions).

В дальнейших версиях проблема скорее всего будет решена. Но даже в нынешнем варианте достигнут, я считаю, хороший уровень безопасности указателей – особенно учитывая тот факт, что при встраивании скрипта в приложения чаще всего используется всё-таки однопоточная модель.

## Указатели на функции

Синтаксис C элегантен и краток, и совсем неудивительно, что он является основой для стольких языков программирования. Но вот вложенные деклараторы (nested declarators)... Бррр…

Понятно, что вложенные деклараторы появились в первую очередь как способ разрешить неоднозначности при объявлении указателей на функции, которые в свою очередь возвращают указатели (возможно, снова на функции, которые опять таки возвращают указатели и так далее и так далее). Но решение получилось, согласитесь, невероятно трудночитаемым и тяжеловесным. В программистских шутках типа «и это тоже валидный C код» я как минимум несколько раз видел использования вложенных деклараторов, как одного из самых верных способов превратить код в нечитаемую кашу из символов.

К счастью, вложенные деклараторы – это не единственное решение проблемы. В Jancy используется другой подход, в котором неоднозначности разрешаются с использованием модификатора function:

int foo ();

bar ()

{

int function\* f () = foo;

}

Надо заметить, что этот подход нисколько не сужает общность объявлений. В Jancy, точно так же как и в C, можно, например, объявить указатель на указатель на функцию, которая возвращает указатель на функцию, которая возвращает указатель на int:

int\* function\* function\*\* a () (int);

Указатели на функции могут быть толстыми и тонкими. Тонкие указатели на функции в Jancy, как и указатели на функции в C/C++, просто содержат в себе адрес кода:

foo (int a);

bar ()

{

function thin\* p (int) = foo;

p (10);

}

В отличие от C/C++, для автоматической конверсии аргументов можно использовать оператор приведения – Jancy умеет генерировать переходники там, где это требуется.

foo (int a);

bar ()

{

typedef FpFunc (double);

FpFunc thin\* f = (FpFunc thin\*) foo; // explicit cast is required to generate a thunk

f (3.14);

}

Толстые указатели открывают ещё более интересные возможности. Помимо адреса кода, толстые указатели содержат ещё и адрес объекта-замыкания, который представляет из себя некий контекст, захваченный в момент создания указателя на функцию. В нём можно сохранять значения контекстных аргументов, что позволяет реализовать парадигму частичного применения.

В простейшем варианте это выглядит так:

class C1

{

foo ();

}

bar ()

{

C1 c;

function\* f () = c.foo; // pointer to 'c' was captured

// ...

}

Jancy также позволяет захватывать в объекте-замыкании произвольные аргументы с помощью оператора частичного применения operator ~(), при этом допускается пропускать аргументы. Таким образом, можно создавать произвольные редукции аргументов, скажем такие, как в примере ниже: аргументы 1 и 4 переданы и захвачены в момент создания указателя, аргументы 2 и 3 будут переданы в момент вызова указателя.

class C1

{

foo (

int x,

int y,

int z

);

}

bar ()

{

C1 c;

function\* f (int, int) = c.foo ~(,, 300);

// ...

f (100, 200); // => c.foo (100, 200, 300);

}

Логическим продолжением идеи со слабыми указателями на классы являются слабые указатели на функции. Они не удерживают выбранных разработчиком захваченных объектов (это означает, что понятие «слабости» применимо только к толстым, но не к тонким указателям на функции). Вызывать функцию по слабому указателю нельзя, можно только попытаться привести её к сильному. В текущей версии Jancy слабым можно сделать только захваченный аргумент this.

class C1

{

foo ();

}

bar ()

{

C1\* c = new C1;

function weak\* wf () = c.weak foo ();

wf (); // <-- error

c = null;

// ...

function\* f () = wf;

if (f)

{

// object is still alive

f ();

}

}

Возможность провести захват и редукцию аргументов, интересная и практически полезная сама по себе, в то же время позволяет элегантно реализовать ещё более интересную концепцию: запуск функции в выбранном окружении: в новом потоке, в уже существующем рабочем потоке, под захваченным локом/мутексом, в обработчике оконного сообщения, и т.д.

В Jancy это может быть достигнуто с помощью оператора планировки @. Первым аргументом оператора выступает функция или указатель на функцию, вторым – экземпляр класса, реализующего интерфейс jnc.Scheduler:

class Scheduler

{

abstract schedule (function\* proc ());

}

Результатом оператора является указатель на функцию, возвращающую void и принимающую те же аргументы, что и первый операнд. Созданный переходник захватывает все переданные аргументы, упаковывает их в объект-замыкание и передаёт полученный указатель на функцию планировщику в Scheduler.schedule (). Что с этим указателем будет происходить дальше и в какой момент он будет в конечном итоге вызван – зависит от логики конкретного планировщика:

class WorkerThread: jnc.Scheduler

{

override schedule (function\* f ())

{

// enqueue f and signal worker thread event

}

// ...

threadProc ()

{

for (;;)

{

// wait for worker thread event

function\* f () = getNextRequest ();

f ();

}

}

}

foo (int x)

{

// ...

}

bar ()

{

WorkerThread workerThread;

function\* f (int) = foo @ workerThread; // create a scheduled pointer

// ...

f (200); // call through a scheduled pointer

(foo @ workerThread) (100); // or schedule immediatly

}

## Мультикасты и события

Jancy предоставляет встроенную поддержку мультикастов (multicast) и событий (event), включая слабые, т.е. такие, от которых вручную отписываться необязательно.

Мультикаст – это генерируемый компилятором специальный класс, позволяющий аккумулировать указатели на функции и затем вызывать их все сразу. Основным (хотя не единственным) применением мультикастов является реализация модели «событие-подписчик».

Объявление мультикаста очень похоже на объявление указателя на функцию, что неудивительно – ведь оно должно однозначно определять, указатели какого типа будут храниться в данном мультикасте. Соответственно, как и указатели на функции, мультикасты могут быть толстыми, тонкими и слабыми.

multicast m (int);

Класс мультикаста, сгенерированный в примере выше, будет иметь следующие методы:

void clear ();

intptr set (function\* (int)); // returns cookie

intptr add (function\* (int)); // returns cookie

function\* remove (intptr cookie) (int);

function\* getSnapshot () (int);

void call (int);

Методы set и add возвращают cookie, который может быть использован в методе remove для эффективного удаления указателя из мультикаста.

Некоторые из методов имеют также псевдонимы в виде операторов:

multicast m ();

m = foo; // same as m.set (foo);

m += bar; // same as m.add (bar);

m -= cookie; // same as m.remove (cookie);

m = null; // same as m.clear ();

m (10); // same as m.call (10);

Ниже приведён пример, демонстрирующий базовые операции с мультикастами:

foo (int x);

bar (

int x,

int y

);

baz ()

{

multicast m (int);

intptr fooCookie = m.add (foo);

m += bar ~(, 200);

m (100); // => foo (100); bar (100, 200);

m -= fooCookie;

m (300); // => bar (300, 200);

m.clear ();

}

Мультикаст можно привести к указателю на функцию, которая вызовет все накопленные в мультикасте указатели. Но тут имеется неоднозначность, а именно: должно ли подобное приведение быть живым (live) или же снимком (snapshot)? Другими словами, если после создания указателя на функцию мы модифицируем исходный мультикаст, должен ли этот указатель видеть изменения?

Для разрешения неоднозначности мультикасты предоставляют метод getSnapshot, возвращающий снимок, а приведение даёт живой указатель:

foo ();

bar ();

baz ()

{

multicast m ();

m += foo;

function\* f1 () = m.getSnapshot ();

function\* f2 () = m;

m += bar;

f1 (45); // => foo ();

f2 (55); // => foo (); bar ();

return 0;

События (event) в Jancy представляют собой специальные указатели на мультикасты с ограничением доступа к методам set, clear и call (то есть по своей природе они походят на const-указатели):

foo ()

{

multicast m (int);

event\* p (int) = m;

p += bar; // OK

p (100); // <-- error

}

Объявление переменной или поля типа «событие» неявно создаёт дуальный тип: для «своих» этот дуальный тип ведёт себя так, как если бы был использован модификатор multicast, а для «чужих» – это event с ограниченным доступом к методам. Подробнее читайте в разделе «дуальные модификаторы».

class C1

{

event m\_onComplete (); // dual type

bool work ()

{

// ...

m\_onComplete (); // OK, friends have multicast-access to m\_onComplete

return true;

}

}

foo ()

{

C1 c;

c.m\_onComplete += foo; // ok, aliens have event-access to m\_onComplete

c.m\_onComplete (); // <-- error

}

В заключение раздела приводу пример реального применения событий и планировки из проекта IO Ninja: событие сокета выстреливает в контексте IO-потока, но с использованием оператора планировки обрабатывается в главном потоке плагина:

TcpListenerSession.construct (doc.PluginHost\* pluginHost)

{

// ...

m\_listenerSocket = new io.Socket ();

m\_listenerSocket.m\_onSocketEvent += onListenerSocketEvent @ pluginHost.m\_mainThreadScheduler;

}

TcpListenerSession.onListenerSocketEvent (io.SocketEventParams const\* params)

{

// we are in main thread, handle socket event...

}

## Свойства

В контексте языков программирования свойство – это нечто, что выглядит и ведёт себя как данные, но при этом позволяет выполнять задаваемые разработчиком действия при считывании и записи. Jancy предоставляет самую полную на сегодняшний день реализацию концепции свойств. Прежде всего дадим необходимые определения.

Функции, инкапсулирующие действия при чтении и записи, называются аксессорами (accessors): аксессор чтения свойства называется геттером (getter), записи – сеттером (setter).

Каждое свойство обладает одним геттером и опционально – одним или несколькими (перегруженными) сеттерами. Если сеттер перегружен, то выбор конкретного сеттера будет произведён во время присвоения значения свойству по тем же правилам, по которым производится выбор перегруженной функции.

Если свойство не имеет сеттера, то оно называется константным (const-property). В других языках программирования свойства без сеттеров иногда называются «только-для-чтения» (read-only), но так как в Jancy понятия const и readonly сосуществуют (почему это так читайте в разделе «дуальные модификаторы»), то переопределить устоявшиеся определения пришлось бы так или иначе. Итак, в Jancy свойство без сеттера – это const-свойство.

Для простых свойств без перегруженных сеттеров (к которым сводится большинство практических задач) предлагается наиболее естественная форма объявления:

int property g\_simpleProp;

int const property g\_simpleConstProp;

Данная форма идеально подходит для объявления интерфейсов, или же если разработчик предпочитает принятый в C++ стиль разнесения объявления и реализации методов:

int g\_simpleProp.get ()

{

// ...

}

g\_simpleProp.set (int x)

{

// ...

}

int g\_simpleConstProp.get ()

{

// ...

}

Для свойств произвольной сложности (т.е. свойств с перегруженными сеттерами, полями данных, вспомогательными методами и т.д.) имеется полная форма объявления:

property g\_prop

{

int m\_x = 5; // member field with in-place initializer

int get ()

{

return m\_x;

}

set (int x)

{

m\_x = x;

update ();

}

set (double x); // overloaded setter

update (); // helper method

}

Jancy также поддерживает индексируемые свойства, т.е. свойства с семантикой массивов. Аксессоры таких свойств принимает дополнительные индексные аргументы. Однако в отличие от настоящих массивов, индексные аргументы свойств не обязаны быть целочисленными, и, строго говоря, не обязаны вообще иметь смысл «индекса» – их использование полностью определяется разработчиком:

int indexed property g\_simpleProp (size\_t i);

property g\_prop

{

int get (

size\_t i,

size\_t j

);

set (

size\_t i,

size\_t j,

int x

);

set (

size\_t i,

size\_t j,

double x

);

}

foo ()

{

int x = g\_simpleProp [10];

g\_prop [x] [20] = 100;

}

В подавляющем большинстве случаев геттер просто должен возвращать значение некоей переменной или поля, где хранится текущее значение свойства, а собственно логика поведения свойства воплощается в сеттере. Очевидно, что создание таких тривиальных геттеров можно переложить на компилятор – что и сделано в Jancy. Для autoget-свойств компилятор автоматически создаёт геттер и поле для хранения данных. Более того, компилятор автоматически генерирует прямой доступ к полю в обход геттера везде, где это возможно:

int autoget property g\_simpleProp;

g\_simpleProp.set (int x)

{

m\_value = x; // the name of a compiler-generated field is 'm\_value'

}

property g\_prop

{

int autoget m\_x; // declaring an autoget field makes the whole property autoget

set (int x);

set (double x);

}

Обратная ситуация, когда специальная логика поведения свойства заложена в геттере, а сеттер-пустышка должен просто класть данные в ячейку памяти, встречается редко и не заслуживает создания специального синтаксиса.

Jancy также предоставляет разработчику связываемы (bindable) свойства, т.е. свойства, которые должны оповещать подписчиков после изменения. Связываемые свойства Jancy играют важную роль в реализации парадигмы реактивного программирования:

int autoget bindable property g\_simpleProp;

g\_simpleProp.set (int x)

{

if (x == m\_value)

return;

m\_value = x;

m\_onChanged (); // the name of a compiler-generated event is 'm\_onChanged'

}

property g\_prop

{

autoget int m\_x;

bindable event m\_e ();// declaring a bindable event makes the whole property bindable

set (int x);

set (double x);

}

Для доступа к событиям, оповещающим об изменениях свойств, применяется оператор bindingof:

onSimplePropChanged ()

{

// ...

}

foo ()

{

bindingof (g\_simpleProp) += onSimplePropChanged;

g\_simpleProp = 100; // bindable event is going to get fired

}

Наконец, Jancy поддерживает связываемые свойства с полностью сгенерированными компилятором аксессорами – так называемые связываемые данные. Эти в некотором роде дегенеративные свойства служат единственной цели – отслеживать изменения.

int bindable g\_data;

onDataChanged ()

{

// ...

}

foo ()

{

// ...

bindingof (g\_data) += onDataChanged;

g\_data = 100; // onDataChanged will get called

g\_data = 100; // onDataChanged will NOT get called

// ...

}

Свойства присутствуют во многих языках программирования, и Jancy не является в этом плане уникальным языком. А вот указателей на свойства нигде не предусмотрено. Для реализации указателей на свойства требуется развитый синтаксис объявлений и операций как со свойствами, так и с указателями, а этим могут похвастаться далеко не многие языки. Jancy является первым языком, в котором поддерживаются указатели на свойства.

Указатели на свойства похожи на и тесно взаимосвязаны с указателями на функции. Они также бывают толстыми, тонкими и слабыми, только вместо адреса кода функции в указателе на свойство лежит адрес таблицы аксессоров. Главное внешнее отличие состоит в том, что при работе с указателями на свойства Jancy требует явной операции взятия адреса & или разыменования \*. Связано это с неявным вызовом аксессоров при обращении к свойству и возникающей из-за этого неоднозначности. К слову сказать, операции взятия адреса и разыменования также применимы и к указателям на функции, просто там они необязательны – с функциями подобной неоднозначности не возникает:

int autoget property g\_prop;

foo ()

{

int property thin\* p = &g\_prop;

// ...

\*p = 10;

}

Толстые указатели на свойства, как и указатели на функции, способны захватывать значения индексных аргументов, тем самым реализуя частичное применение в отношении свойств – это выражается в сокращении размерности или полной де-индексации свойства. В отличие от функций специальный оператор частичного применения тут не нужен, достаточно использовать оператор индексации (где это необходимо, индексы можно оставлять пустыми):

class C1

{

int indexed property m\_prop (

size\_t i,

size\_t j

);

// ...

}

foo ()

{

C1 c;

int indexed property\* p (size\_t) = &(c.m\_prop [] [20]);

// ...

\*p [10] = 100; // => c.m\_prop [10] [20] = 100;

}

## Реактивное программирование

Jancy – это один из первых императивных языков со встроенной поддержкой реактивного программирования. Объяснять концепцию реактивного программирования лучше всего на примере.

Все мы когда-нибудь пользовались электронными таблицами типа Microsoft Excel. В ячейке таблицы пользователь может написать формулу, которая ссылается на другие ячейки. Если значение любой из ячеек, от которых зависит эта формула, изменится – формула будет пересчитана, и наша ячейка автоматически обновится.

Теперь представим на секунду, что вместо этого каждая ячейка предоставляет событие onValueChanged, и пользователю необходимо вручную прописать и привязать обработчики этих событий, в каждом из которых, опять-таки вручную, обновить значения зависимых ячеек. Данный пример перестанет вам казаться абсурдным и искусственным как только вы осознаете, что именно так сейчас и работает подавляющее большинство моделей построения пользовательского интерфейса!

Я долго экспериментировал в надежде найти способ использовать эту концепцию реактивного программирования из мира императивных языков. Несмотря на очевидность реализации в таблицах типа Excel, было совершенно непонятно в каком объёме, с какими допущениями, и как именно реактивное программирование должно сочетаться и взаимодействовать с основным объёмом программы. Тем не менее, после многочисленных фальстартов, приемлемый подход для интеграции парадигмы реактивного программирования в обычный императивный язык был найден.

Сосуществование реактивного и императивного кода становится возможно с введением специальных зон реактивного кода – реакторов (reactors). Внешне реактор выглядит как обычная функция, разве что в объявлении указан модификатор reactor. В отличие от функций каждый реактор создаёт переменную или поле особого реакторного класса с двумя методами: start и stop, позволяющими запустить и остановить реактор. Вместо инструкций (statements), из которых состоит тело обычной функции, тело реактора состоит из последовательности выражений, каждое из которые должно использовать в своей правой части связываемые свойства:

State bindable m\_state;

reactor m\_uiReactor ()

{

// ...

m\_isTransmitEnabled = m\_state == State.Connected;

m\_actionTable [ActionId.Disconnect].m\_isEnabled = m\_state != State.Closed;

// ...

}

При старте реактор строит граф зависимостей и подписывается на все связываемые события всех «управляющих» свойств. При изменении любого из них все зависимые выражения пересчитываются (что, естественно, может вызвать лавинообразное изменение других свойств). В настоящий момент циклические обновления зависимостей игнорируются, то есть если какое-либо из управляющих свойств изменилось и вызвало пересчёт выражений в реакторе, которые в свою очередь поменяли значение этого же свойства, то это повторное изменение не вызовет рекурсивного запуска вычислений в реакторе. В дальнейшем такие ситуации циклических зависимостей будут отлавливаться на этапе компиляции (а где невозможно – выбрасывать ошибку во время выполнения), так как они, скорее всего, свидетельствуют о плохо спроектированном реакторе.

При останове реактор отписывается от всех событий, на которые успел подписаться. Если реактор является членом класса, то останов автоматически происходит в момент разрушения родительского объекта (для контроля момента разрушения можно использовать парадигму RAII). Таким образом, разработчик имеет возможность детально определять и *где* использовать реактивный подход (зоны реакторов), и *когда* использовать такой подход (старт/стоп).

К сожалению, одними только выражениями обойтись можно не всегда. Что если в ответ на изменение управляющего свойства надо пробежаться в цикле по массиву или выполнить какую-либо другую последовательность действий? Кроме того, подписка на *каждое* связываемое свойство в выражении может оказаться ковровой бомбардировкой, в то время как на самом деле нужен снайперский выстрел – у разработчика должна быть возможность тонкой настройки, т.е. определения, за какими именно свойствами требуется следить.

Для решения вышеназванных проблем реакторные блоки Jancy используют инструкцию onevent. Данный подход даёт полный контроль над тем, какие именно события отслеживать. В то же время он избавляет от необходимости вручную подписываться на каждое событие – вместо этого в интуитивно-понятном синтаксисе увязываются событие и код его обработки:

reactor m\_uiReactor ()

{

onevent bindingof (m\_state) ()

{

// handle state change...

}

onevent (bindingof (m\_userEdit.m\_text), bindingof (m\_passwordEdit.m\_text)) ()

{

// handle login change...

}

onevent m\_startButton.m\_onClicked ()

{

// handle start button click...

}

}

Как видно из примера, onevent позволяет привязывать обработчики к нескольким событиям сразу, при этом события могут быть произвольными, а не только связываемыми.

В заключение приведу пример реального использование реакторов в проекте IO Ninja:

reactor FileSession.m\_uiReactor ()

{

m\_title = $"File $(m\_fileNameCombo.m\_editText)";

m\_isTransmitEnabled = m\_state == State.Opened;

m\_actionTable [ActionId.Open].m\_text = m\_state ? "Close" : "Open";

m\_actionTable [ActionId.Open].m\_icon = m\_iconTable [m\_state ? IconId.Close : IconId.Open];

m\_actionTable [ActionId.Clear].m\_isEnabled =

m\_state && m\_file.m\_kind == io.FileStreamKind.Disk;

m\_statusPaneTable [StatusPaneId.State].m\_text = m\_state ? "Opened" : "Closed";

}

## Исключения

В Jancy нет исключений в том виде, в каком их привыкли использовать программисты на C++ или Java. Вместо этого Jancy предлагает синтаксический сахар над старой доброй моделью кодов ошибок.

Вспомним, как обрабатываются ошибки в мире C. Функция, которая может завершиться неудачно, сигнализирует о статусе операции через возвращаемое значение. Тут возможны варианты. Иногда напрямую возвращается код ошибки (или 0, если операция завершилась успешно), иногда – наоборот, в случае успеха возвращается некий результат работы функции (например, указатель на созданный объект), а в случае ошибки – специально оговоренное значение (в случае указателя логичным «ошибочным» значением выглядит null). Во втором варианте подробности об ошибке обычно кладутся в выделенную потоковую (TLS) переменную (как в случае с общеизвестными POSIX errno или Windows GetLastError); иногда же разработчики API заставляют предварительно заготовить и передавать в функцию буфер для сообщения об ошибке, если таковая вдруг возникнет.

Основная проблема в данном подходе состоит в том, что при его использовании разработчик вынужден после каждого вызова функции проверять возвращаемые значения на предмет ошибки. Также крайне неудобно и то, что вся информация об ошибке, как правило, ограничена целочисленным кодом.

Исключения C++ и Java полностью устраняют вышеперечисленные недостатки, но, как это часто бывает, одновременно с решением старых проблем привносят новые. Исключения, особенно в C++ – это далеко не бесплатный в плане накладных расходов механизм, очень непростой в реализации, а в случаях мультиязыкового стека вызовов – совсем уже неподъёмный. При этом исключения обладают рядом других недостатков, например, делают поведение написанного кода значительно менее прозрачным и предсказуемым, возращение ошибки становится гораздо более «дорогим», чем успешное завершение (что неприемлемо для ситуаций, когда возврат ошибки – это типовой сценарий), поощряют создание непоследовательных API, в которых какие-то методы сигнализируют об ошибках с помощью исключений, а какие-то – с помощью кодов возврата, и т.д. Не хочется повторять многократно сказанное другими, желающие легко найдут горячие обсуждения достоинств и недостатков исключений на любом программистском форуме, включая RSDN.

При дизайне Jancy я решил остановиться на гибридной модели. В основе её лежит проверка возвращаемых значений, но компилятор избавляет от необходимости делать это вручную. В итоге всё выглядит *почти* как исключения в C++ или Java, но при этом поведение программы при ошибках на порядок более прозрачно и предсказуемо, а поддержка исключений при межязыковых взаимодействиях (таких, как вызов функций C++ из скриптов на Jancy) становится настолько простой, насколько это вообще возможно.

Помимо этого, в данной модели разработчик волен выбирать, как именно обрабатывать ошибки в каждом конкретном случае. Иногда это удобнее делать проверкой кода возврата вручную, иногда – использовать семантику исключений. В Jancy при вызове одной и той же функции можно делать и так и так, в зависимости от ситуации.

Возвращаемые значения функций, помеченных модификатором throws, будут трактоваться как коды ошибок. В Jancy приняты интуитивные условия ошибки для стандартных типов: false для булева типа, null для указателей, -1 для беззнаковых целых, и < 0 для знаковых. Остальные типы приводятся к булеву (если это невозможно, то выдаётся ошибка компиляции). К слову сказать, в предыдущих версиях Jancy был специальный синтаксис для задания произвольных условий ошибки, но на практике он ни разу не использовался и к настоящему моменту убран за ненадобностью.

bool foo (int a) throws

{

if (a < -100 || a > 200) // invalid argument

return false;

// ...

return true;

}

Очевидно, что функция, возвращающая void, в данной модели не может возвращать ошибки:

void foo () throws; // <-- error

Если тип возвращаемых функциями значений совпадает, то при возникновении ошибки возвращённое значение будет автоматически передано вниз по стеку вызовов:

int foo (int a) throws

{

if (a < -100)

return -1;

if (a > 200)

return -2;

// ...

return 0;

}

int bar () throws

{

for (size\_t i = 0; i < 3; i++)

foo (rand ()); // if foo returns error, it will be propagated down the call stack

return 0;

}

Если типы не совпадают, то вниз по стеку автоматически будет передан умолчальный код ошибки (false для булева типа, null для указателей, -1 для целых; для остальных типов будет ошибка компиляции – требуется передача ошибки вниз вручную).

Как было сказано выше, для обработки ошибок возможны два подхода. Как правило, удобно заставить компилятор проверять коды возврата автоматически и обрабатывать возникающие ошибки в едином блоке catch:

bool foo (int a) throws;

bar ()

{

foo (10);

foo (20);

foo (-200); // bar will fail so we're not going to get to the next line

foo (300);

catch:

// handle error

}

Если же где-то необходимо проверить условие вручную, то это тоже не проблема:

bool foo (int a) throws;

baz (int x)

{

bool result = try foo (x);

if (!result)

{

// handle error

}

}

Естественным образом возникает вопрос о передаче информации об ошибке – ведь в отличие от исключений в C++ и Java, блок catch в Jancy не принимает никаких дополнительных аргументов.

Вместо аргумента можно воспользоваться потоковыми переменными, в которые разработчик волен класть произвольные данные о возникающих ошибках. В Jancy также имеются стандартные функции для сохранения и получения информации о последней ошибке в текущем потоке:

bool foo (int a) throws

{

if (a < -100)

{

jnc.setPosixError (22); // EINVAL

return false;

}

if (a > 200)

{

jnc.setStringError ("detailed-description-of-error");

return false;

}

return true;

}

bar ()

{

foo (-1);

foo (150);

catch:

printf ("error caught (%s)\n", jnc.getLastError ().m\_description);

}

Конструкция finally в большинстве языков традиционно ассоциируется с исключениями. В Jancy область применения данной конструкции гораздо шире. В конце концов, убирать за собой надо даже если никаких ошибок не возникало, не правда ли?

Предположим, что разработчик хочет написать код освобождения всех захваченных ресурсов и отправки какого-нибудь уведомления о завершении операции. Этот код надо выполнить по выходу из заданного блока независимо от того, каким путём этот выход был совершён. Зачем заставлять разработчика использовать конструкцию try только для этого (как в Java)? Ведь никаких исключений может и не предвидится!?

Если вы задавали себе этот вопрос, то у меня есть хорошие новости: в Jancy finally может быть добавлен в любой блок по желанию разработчика:

foo ()

{

// ...

// nothing to do with exceptions here, just a 'finally' block

finally:

printf ("foo () finalization\n");

}

Конечно, допускается и более традиционное использование конструкции finally в случаях, когда исключения-таки ожидаются:

foo (char const\* address)

{

try

{

open (address);

transact (1);

transact (2);

transact (3);

catch:

addErrorToLog (jnc.getLastError ());

finally:

close ();

}

// ...

}

## Строковые литералы

Jancy предлагает разработчику три вида строковых литералов.

Первый – это старый добрый строковый C-литерал, определяющий константный массив из char в статической памяти:

char s [] = "hello world!";

Второй вид – это шестнадцатеричный литерал (hex-literal). Он предназначен для удобства при определении жестко прописанных (hardcoded) блоков константных бинарных данных. Реалистичными примерами ситуаций, когда это может понадобиться, могут быть иконки, курсоры, публичные ключи и другие ресурсы.

Разумеется, в C/C++ для подобных целей можно использовать константные массивы байтов со статической инициализацией, однако hex-литералы Jancy значительно компактнее и нагляднее:

char s1 [] = 0x"61 62 63 20 64 65 66 00";

char s2 [] = { 0x61, 0x62, 0x63, 0x20, 0x64, 0x65, 0x66, 0x00 };

При задании hex-литерала можно использовать как заглавные, так и строчные буквы, а также по желанию расставлять пробелы между байтами или не использовать пробелы вообще:

char s [] = 0x"696a 6b6c 6D6E 6F70 0000";

Hex-литерал определяет статический константный массивы из char. Допускается написание нескольких литералов подряд для получения «склеенного» массива. В отличие от C-литералов, hex-литералы не завершаются неявным нулевым байтом, и это можно использовать для удаления завершающего нулевого байта из C-литерала:

char s [] = "non-zero-terminated" 0x"";

Третий и последний вид строковых литералов Jancy – это форматирующий литерал. Форматирующие литералы позволяют динамически подставлять значения выражений, используемых внутри литерала, тем самым реализуя форматирование через интерполяции (interpolation), как в языке Perl.

В Jancy данный подход называется «Perl-подобное форматирование», или «форматирование в стиле Perl», что, на мой взгляд, наиболее понятным для всех образом отражает суть форматирующих литералов:

int i = rand ();

char const\* s = $"i = $i";

Вместо статических массивов форматирующие литералы генерируют массив в GC-heap и возвращают указатель на этот массив (для получения размера массива можно воспользоваться оператором dynamic sizeof). Форматирующие литералы неявно завершаются нулевым байтом.

При подстановке допускается использование произвольных выражений внутри скобок:

char const\* s = $"random = $(rand () % 100)";

Также поддерживаются спецификаторы форматирования printf – их можно указывать после выражения (для разделения спецификатора и самого выражения необходимо использовать точку с запятой):

uint\_t color = m\_colorTable [rand () % countof (m\_colorTable)];

char const\* s = $"color value = #$(color; 06x)\n";

При подстановке нескольких достаточно развёрнутых выражений Perl-подобное форматирование резко теряет своём самом главном качестве – наглядности. Для решения данной проблемы в Jancy допускается переносить выражения из тела литерала в список аргументов, сразу за ним следующий. При этом в самом литерале эти внешние выражения адресуются по индексу. Данный подход также позволяет подставлять одно и то же выражение многократно (возможно, с разными спецификаторами форматирования):

char const\* s = $"rgb dec = (%1, %2, %3)\nrgb hex = (%(1;x), %(2;x), %(3;x))\n" (

(color & 0xff0000) >> 16,

(color & 0x00ff00) >> 8,

(color & 0x0000ff)

);

В заключение следует отметить, что все виды литералов можно «склеивать» друг с другом в любых комбинациях. При этом, если среди склеенных элементов отсутствуют форматирующие литералы, то образуется статический массив из char, в противном случае результатом склеивания будет указатель на массив в GC-heap:

char const\* s =

"stats\n"

0x"e29480 e29480 e29480 e29480 e29480 e29480 e29480 e29480 e29480 0d"

$"size = $(sizeof (colorTable))\n"

$"count = $(countof (colorTable))\n"

);

## Дуальные модификаторы

Про модель контроля доступа в Jancy уже было сказано несколько слов в разделе, посвящённом классам: спецификатор может быть указан либо в стиле C++, либо в стиле Java, типов доступа всего два – public и protected, умолчальный тип доступа – public. Теперь поговорим немного более подробно о причинах подобного дизайна.

Основным отличием модели Jancy от большинства других объектно-ориентированных языков является сокращение количества спецификаторов доступа до двух. С одной стороны, это предоставляет разработчикам значительно меньшую гибкость в определении того, кто и к чему имеет доступ. С другой стороны, данная упрощённая модель позволяет делить всех на «свой-чужой», а это, свою очередь, открывает возможность дуальных модификаторов, т.е. модификаторов, имеющих разное значение для «своих» и «чужих».

Итак, в модели Jancy для каждого отдельно взятого пространства имён A весь остальной мир распадается на две категории: «свои» и «чужие». Помимо самого пространства имён A, к «своим» относятся:

* Пространства имён классов или структур, унаследованных от A;
* Пространства имён, объявленные как дружественные (friend);
* Дочерние по отношению к A пространства имён;
* Расширения (extension namespaces) A.

Все остальные являются «чужими». «Свои» имеют доступ и к публичным (public), и к защищённым (protected) членам пространства имён, в то время как «чужие» – только к публичным членам.

Дуальный модификатор readonly может быть использован для элегантной организации доступа только на чтение. Вместо написания тривиальных геттеров, единственным назначением которых был бы контроль доступа, разработчик на Jancy может объявлять поля с модификатором readonly. Для «своих» модификатор readonly как бы невидим, для «чужих» readonly трактуется как const (см. const-корректность):

class C1

{

int readonly m\_progress;

foo ()

{

m\_progress += 25;

// ...

}

}

bar (C1\* c)

{

c.m\_progress = 100; // <-- error

}

Главное преимущество данного подхода – это то, что он делает код короче и естественнее; как побочный положительный эффект можно назвать упрощение, а значит и ускорение работы оптимизатора, которому не нужно анализировать и выкидывать пустышки-геттеры.

Второй дуальный модификатор в Jancy – это event. Владелец события должен иметь над ним полный контроль, включая возможность вызвать всех подписчиков или очистить их список. Клиент события должен иметь возможность *только* добавить или удалить подписчика. Для «своих» поле с модификатором event работает так же, как и мультикаст с соответствующей сигнатурой аргументов. Для «чужих» такое поле ограничивает доступ к методам мультикаста: разрешены только вызовы add и remove, запрещены call, set и clear:

class C1

{

event m\_onCompleted ();

work ()

{

// ...

m\_onCompleted ();

}

}

bar ();

baz ()

{

C1 c;

c.m\_onCompleted += baz;

c.m\_onCompleted (); // <-- error

}

## Перечисления

Jancy также предлагает ряд усовершенствований в перечислениях (enum).

Перечисления в Jancy не добавляют идентификаторы элементов в родительское пространство имён, тем самым предотвращая его загрязнение и делая возможным использование коротких идентификаторов для элементов. Для адресации элемента перечисления требуется полное имя (qualified name) с квалификатором родительского типа.

Помимо этого, перечисления могут быть «унаследованы» от целочисленных типов для указания знаковости перечисления, а также точного количества занимаемых им байтов (что может быть крайне важно при описаниях заголовков коммуникационных протоколов):

enum IcmpType: uint8\_t

{

EchoReply = 0,

DestinationUnreachable = 3,

SourceQuench = 4,

Redirect = 5,

Echo = 8,

RouterAdvertisement = 9,

RouterSelection = 10,

TimeExceeded = 11,

ParameterProblem = 12,

TimestampRequest = 13,

TimestampReply = 14,

InformationRequest = 15,

InformationReply = 16,

AddressMaskRequest = 17,

AddressMaskReply = 18,

TraceRoute = 30,

}

Для упрощения переноса существующего кода с С/С++ на Jancy поддерживаются т.н. открытые (exposed) перечисления. Как и перечисления в C/С++, они добавляют идентификаторы элементов в родительское пространство имён:

exposed enum State

{

State\_Idle, // = 0

State\_Connecting,

State\_Connected,

State\_Disconnecting,

}

foo ()

{

State state = State\_Idle; // no need to qualify

// ...

}

Наиболее значительным новшеством Jancy в области перечислений является встроенная поддержка флаговых (bitflag) перечислений.

Большинство существующих языков программирования не предлагают встроенных флаговых перечислений, а их эмуляция штатными методами, как правило, и нетривиальна в реализации, и не столь удобна в практическом использовании. Jancy же предлагает готовое решение «прямо из коробки».

Флаговые перечисления отличаются от обычных перечислений в следующих аспектах:

* Автоматическое присвоение значений элементам идёт по степеням двойки, т.е. 1, 2, 4, 8 и т.д. Если быть более формальным, то для определения значения каждого последующего элемента берётся самый старший бит значения предыдущего и сдвигается влево; самый первый элемент по умолчанию имеет значение «1»;
* Битовое ИЛИ (|) над двумя операндами, оба из которых имеют один и тот же тип флагового перечисления, в результате даёт этот же тип;
* Битовое И (&) над двумя операндами, один из которых имеет тип флагового перечисления, даёт этот же тип; если оба операнда имеют типы флаговых перечислений, которые при этом не совпадают, то типом результата будет тип первого операнда;
* Разрешается назначать 0 в переменную или поле с типом флагового перечисления.

bitflag enum OpenFlags: uint8\_t

{

ReadOnly, // = 0x01

Exclusive = 0x20,

DeleteOnClose, // = 0x40

}

foo ()

{

OpenFlags flags = OpenFlags.ReadOnly | OpenFlags.Exclusive;

// ...

flags &= ~OpenFlags.Exclusive;

// ...

flags = 0; // OK

}

По умолчанию флаговые перечисления не добавляют идентификаторы элементов в родительское пространство имён, однако это можно изменить сочетанием модификаторов bitflag и exposed – на выходе, как можно догадаться, получится открытое флаговое перечисление.

## Ленивая инициализация

Jancy предоставляет встроенное средство для ленивой инициализации. Будучи предварённой ключевым словом once любая инструкция (statement), в том числе и составная (compound statement { … }), будет заключена в сгенерированный компилятором аналог boost::call\_once (), гарантируя тем самым однократное её исполнение:

foo ()

{

once

{

// lazy initialize...

}

// ...

}

Вариация thread once, соответственно, гарантирует исполнение инициализационного кода один раз в каждом потоке:

foo ()

{

thread once initializeThread ();

// ...

}

Конструкторы и инициализаторы статических и потоковых переменных также исполняются однократно. Это гарантируется и нет необходимости явного использования once:

class C1

{

construct ()

{

// ...

}

}

foo ()

{

static C1 c; // 'C1.construct' will be called once per program run

thread int x = bar (); // 'bar' will be called once per thread

}

## Конструкции break-n, continue-n

Помню, на RSDN была целая «священная война» с полномасштабными ожесточёнными дебатами и даже комиксами (!), навеянная темой «выйти из двух циклов сразу». В существующих языках реализовать подобное можно либо с помощью оператора goto, что справедливо считается дурным тоном, либо с помощью исключений, что, в свою очередь, слишком «дорого» и громоздко для такой задачи.

Между тем, с точки зрения компилятора выйти из нескольких циклов сразу ничуть не сложнее, чем из одного. Для удобного управления внешними циклами Jancy предоставляет операторы break-n и continue-n:

int a [3] [4] =

{

{ 1, 2, 3, 4 },

{ 5, 6, -7, 8 },

{ 9, 10, 11, 12 },

};

for (size\_t i = 0; i < countof (a); i++)

for (size\_t j = 0; j < countof (a [0]); j++)

if (a [i] [j] < 0)

{

// negative element is found, process it...

break2; // exit 2 loops at once

}

Разумеется, break-n может также быть использован для выхода из конструкции switch, а не только циклов. Реалистичным примером применения этой возможности будет, например, обработка запросов в цикле, крутящемся в рабочем потоке; один из запросов должен данный цикл завершать:

for (;;)

{

Request request = getNextRequest ();

switch (request)

{

case Request.Terminate:

break2; // out of the loop

case Request.Open:

// ...

break;

case Request.Connect:

// ...

break;

// ...

}

}

## Добавление методов в существующие типы

Jancy поддерживает расширение существующих именованных типов посредством добавления невиртуальных методов уже после того, как тип был объявлен. Это является аналогом методов расширения (extension methods) в C#.

Основным способом расширения функциональности существующих типов в объектно-ориентированных языках является наследование. Существует, однако, множество сценариев, в которых наследование будет излишне тяжеловесно, неудобно в использовании, либо совсем неприменимо.

Пространства имён расширения (extension namespace) в Jancy призваны дать разработчику возможность легко дописывать недостающие утилитарные методы у существующих типов:

class C1

{

protected int m\_x;

foo ();

}

extension C1Ext: C1

{

bar (int x);

{

m\_x = -m\_x; // extension has access to protected members

}

static baz ();

int property m\_prop;

}

Для использования методов расширения необходимо «включить» соответствующее пространство имён с помощью директивы using:

foo (C1\* c)

{

c.foo ();

using extension C1Ext;

c.bar (500);

c.m\_prop = 100;

C1.baz ();

}

Очевидно, что пространство имён расширения не должно иметь права *модифицировать* существующий тип, а только иметь возможность его расширять. Соответственно, запрещено добавлять:

* Деструкторы;
* Поля данных;
* Виртуальные методы;
* Autoget-свойства (следует из запрещения на добавление поля);
* Связываемые свойства (следует из запрещения на добавление поля).

Теоретически, добавление конструкторов и перегруженных операторов не должно вызывать проблем и, скорее всего, будет реализовано в последующих версиях Jancy. В текущей версии разрешены только именованные невиртуальные методы и невиртуальные свойства без полей.

## Области видимости в switch

Области видимости в инструкции switch языков C и C++ – это моя больная мозоль. Разработчик пишет себе код внутри switch, рано или поздно пытается объявить локальную переменную с конструктором (или даже простой int с инициализацией!) – и тут же получает ошибкой по лбу: инициализация переменной может быть пропущена из-за переходов на метки case или default. Приходится либо выносить переменную за switch, нарушая принцип локальности в отношении объявления переменной и её использования, либо городить switch-уродец с явно указанными кое-где областями видимости и нарушенной структурой табуляции.

В Java компилятор не ругается на инициализацию локальных переменных в switch, и одновременно не даёт использовать или переопределить их из других меток case. Кто, когда и зачем захочет использовать локальную переменную, объявленную в соседней метке case? И главное, что мешает компилятору просто создавать неявные области видимости для каждой метки case?

Правильный ответ: ничто не мешает. Как и в языке D, Jancy помещает код каждой метки case в неявную область видимости:

switch (state)

{

case State.Idle:

int i = 10;

break;

case State.Connecting:

int i = 20; // no problem: we are in different scope

case State.Disconnecting:

int i = 30; // no problem even when we fall-through from previous case label

break;

default:

int i = 40; // still OK -- you've got the idea

}

## Curly-инициализаторы

Одна из моих горячо любимых возможностей синтаксиса С/С++ – это инициализация массива или структуры сразу после объявления при помощи т.н. curly-инициализаторов. Лаконично, интуитивно понятно, и при этом предельно просто в реализации в компиляторе:

struct Point

{

int m\_x;

int m\_y;

};

Point point = { 10, 20 };

int a [] = { 100, 200, 300 };

В С# добавили возможность использования имён членов структур и классов в момент создания с помощью оператора new, но при этом последовательная инициализация полей структур стала невозможной из-за разделения на инициализаторы коллекций и инициализаторы объектов.

В Jancy я постарался довести curly-инициализаторы до ума:

int a [10] = { ,, 3, 4,,, 7 };

Point point = { , 20, m\_z = 30 };

point = { 1000,, m\_z = 3000 };

Point\* point2 = new Point { , 200, m\_z = 300 };

Как видно из примера выше, в Jancy можно:

* Использовать curly-инициализаторы после произвольных присвоений (а не только в объявлениях);
* …а также и в операторе new;
* Допускается пропускать поля/элементы массива: при инициализации в объявлении они останутся нулевыми, при присвоении будут просто не тронуты;
* Для адресации полей можно использовать как имя (стиль C#) так и индекс (стиль C/С++), и даже комбинировать эти два метода.

При комбинации двух видов адресации в одном curly-инициализаторе допускается использовать именную адресацию после индексной, но не наоборот. Это искусственное ограничение введено для избежания потенциальных неочевидных эффектов и неоднозначности при инициализации вложенных структур:

struct Neighborhood: Point

{

int m\_radius;

}

Neighborhood n = { m\_x = 10, 20 }; // does 20 go to m\_y or to m\_radius?

Впрочем, трудно представить себе ситуацию, в которой это ограничение причиняло бы сколь-нибудь существенные неудобства.

## Модуль-конструкторы и деструкторы

Jancy поддерживает модуль-конструкторы и деструкторы как удобный и естественный способ написания кода инициализации и очистки для юнитов компиляции.

Модуль-конструктор вызывается после конструкции и инициализации всех глобальных статических переменных, модуль-деструктор – перед вызовом деструкторов глобальных статических переменных (если таковые имеются):

class C1

{

construct ();

destruct ();

}

int foo ();

C1 g\_c;

int g\_x = foo ();

construct ()

{

// module constructor

}

destruct ()

{

// module destructor

}

В примере выше модуль-конструктор будет вызван после C1.construct (для конструирования g\_c) и после foo (для инициализации g\_x), а модуль-деструктор будет выполнен перед вызовом C1.destruct.

# Архитектура компилятора

К сожалению в рамках статьи, целью которой является рассказ о возможностях языка, очень трудно подробно обсудить технические детали компилятора. В связи с этим лишь кратко коснусь мотивации в отношении основных архитектурных решений.

Для генерации лексического анализатора используется универсальный компилятор конечных автоматов Ragel. Ragel замечательно подходит для создания лексеров в силу удобства входного языка и эффективности выходного кода, превосходящей все общеизвестные аналоги. С помощью Ragel также можно писать лексеры со множественными переключаемыми режимами распознавания (machine в терминологии Ragel), что необходимо в Jancy для токенизации форматирующих литералов.

Использование в качестве backend компилятора LLVM, а не рукописной виртуальной машины, или же рукописного генератора целевого кода – это решение, которое не вызывало сомнений с самого начала. LLVM – это инфраструктура для создания backend компиляторов, выросшая из исследовательского проекта университета Иллинойса. В настоящий момент LLVM находится в состоянии активной разработки c полугодовым циклом релизов и используется в продуктах таких гигантов IT-индустрии, как Apple, Adobe, Intel, Sony и т.д. С практической точки зрения использование LLVM позволяет задействовать готовый отлаженный оптимизатор и кодогенератор сразу для широкого спектра платформ, а также существенно упрощает обеспечение совместимости с ABI C/C++.

А вот для синтаксического анализа я-таки не удержался и написал свой велосипед – генератор табличных нисходящих LL (k) анализаторов Bulldozer. Чем меня не устраивали ANTLR, Coco, Yacc/Bison, Lemon и другие безусловно уважаемые и проверенные парсер-генераторы – это тема для отдельного и интересного разговора. Если совсем кратко, то нужен был нисходящий парсер-генератор с поддержкой anytoken и, что самое главное, с настраиваемым и предсказуемым механизмом разрешения конфликтов; готовые продукты, к сожалению, не справлялись в тех или иных ситуациях.

Использование в компиляторе Jancy сгенерированного синтаксического анализатора, а не ручного рекурсивного спуска, было крайнее желательным именно по причине творческой работы над синтаксисом языка. Хотелось создать максимально красивый и естественный синтаксис, а это значит, что предстояли бесконечные эксперименты и переписывания, переписывания и ещё раз переписывания грамматики.

Плюсы сгенерированного парсера:

* Просто менять синтаксис языковых конструкций;
* Наличие EBNF-грамматики как документа, в любой момент времени отражающего *действительный* синтаксис языка;
* Более строгий контроль грамматики – естественные ограничения компилятора грамматики не дают сорваться и наваять в творческом угаре «бешеный» синтаксис (в то время как рекурсивный спуск простит и съест всё).

Грамматика Jancy относится к классу контекстно-зависимых LL (2). Выходом парсера является сразу LLVM IR (intermediate representation – промежуточное представление) без предварительной генерации AST (abstract syntax tree): 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 делает возможным инкрементальный разбор (кусок за куском, а не весь юнит компиляции сразу); помимо этого, отсутствует необходимость искусственных ограничений на уровень вложенности или иных механизмов предотвращения переполнения стека, так как нет прямой зависимости нагрузки на стек от входного кода.

Синтаксический/семантический анализ многопроходный (два или три прохода), что, однако, не означает повторного запуска лексера. Второй проход сделан для возможности использовать глобальные типы и данные до парсинга их объявления – Jancy не требует строгого следования парадигме «объявление-перед-использованием» в глобальной области видимости. Третий проход необходим, в частности, для предварительного расчёта реакторных классов.

# Заключение и статус

Jancy – это кросс-платформенный проект, нацеленный в первую очередь на Windows, Linux и Mac (порт на Mac пока не реализован). JIT-компилятор Jancy способен выдавать целевой код для любой архитектуры процессоров, поддерживаемой LLVM.

Основная модель применения в ближайшей перспективе – использование Jancy как скриптового движка из приложений на C++.

Версия IO Ninja, написанная с использованием Jancy, уже выпущена. IO Ninja служит первым примером реального применения языка и всех его инновационных концепций. Основанная на NetBeans среда разработки (IDE), в которую входят плагины для Jancy и IO Ninja с поддержкой code assist и отладки через GDB, также уже доступна для скачивания. К сожалению, отладка под Windows в настоящее время невозможна – LLVM пока не поддерживает (хочется надеяться, что только «пока») генерацию отладочной информации, воспринимаемой Visual Studio или CDB.

На сайте проекта доступна тестовая страничка компилятора Jancy, которая позволяет играться с языком без необходимости что-либо скачивать и устанавливать. Определённые возможности, которые можно ожидать от современного языка (такие как reflection, generics/templates, лямбда-функции и т.д.) пока не реализованы, но обязательно появятся в будущих релизах. В то же время возможности, которые, как мы полагаем, уже готовы и отлажены, после ударного тестирования через веб скорее всего обнаружат недоделки, недоработки и просто элементарные баги. В конце концов, это первый публичный релиз, и баг-репортам мы будем не менее рады, чем положительным отзывам!

Наша компания и я лично надеемся, что нам удалось пробудить интерес к проекту и мы будем счастливы получить от уважаемого сообщества разработчиков вопросы, предложения и отзывы любой степени доброжелательности.

# Внешние ссылки и список литературы

1. Страничка проекта Jancy <http://tibbo.com/jancy/>
2. Страничка проекта IO Ninja [http://tibbo.com/ioninja/](http://tibbo.com/ioninja)
3. Ragel State Machine Compiler User Guide by Adrian Thurston <http://www.complang.org/ragel/ragel-guide-6.8.pdf>
4. Ragel wins! Fatality! <http://www.wincent.com/a/about/wincent/weblog/archives/2008/02/ragel_wins_fata.php>
5. The LLVM Compiler Infrastructure Documentation <http://llvm.org/docs/>
6. Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman (Jan 1, 1986)
7. Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam,Ravi Sethi and Jeffrey D. Ullman (Sep 10, 2006)
8. Engineering a Compiler, Second Edition by Keith Cooper and Linda Torczon (Feb 21, 2011)
9. A Retargetable C Compiler: Design and Implementation by David R. Hanson and Christopher (Feb 10, 1995)
10. Garbage Collection: Algorithms for Automatic Dynamic Memory Management by Richard Jones and Rafael D Lins (Aug 16, 1996)
11. The Garbage Collection Handbook: The Art of Automatic Memory Management (Chapman & Hall/CRC Applied Algorithms and Data Structures series) by Richard Jones, Antony Hosking and Eliot Moss (Aug 17, 2011)

1. They didn’t write it. Полный текст: <http://www.drdobbs.com/windows/a-brief-history-of-windows-programming-r/225701475> [↑](#footnote-ref-1)