JANCY: Возвращение Указателей

Contents

[Объявления 1](#_Toc369725422)

[Категории типов данных 2](#_Toc369725423)

[Категории памяти данных 3](#_Toc369725424)

[Классы (class), структуры (struct) и объединения (union) 4](#_Toc369725425)

[Указатели 8](#_Toc369725426)

[Функции, замыкания (closures), мультикасты (multicasts) 14](#_Toc369725427)

[Свойства (properties) 21](#_Toc369725428)

[Поддержка реакционного программирования 29](#_Toc369725429)

[Пространства имён и модель контроля доступа 32](#_Toc369725430)

[Другое 35](#_Toc369725431)

[Curly-инициализаторы 35](#_Toc369725432)

[Единократное выполнение кода (once) 36](#_Toc369725433)

[Псеводонимы выражений (alias) 36](#_Toc369725434)

[Break-n/Continue-n 37](#_Toc369725435)

[Scopes-in-switch 37](#_Toc369725436)

# Объявления

Синтаксис объявлений в Jancy С-подобный (спецификатор-декларатор-инициализатор), то есть в целом такой же как в C, C++, Java и D. Детали реализации, однако, ближе к Java чем к C/C++, а именно: объявление именованых типов НЕ является спецификтором типа. Это означает, что, например, нельзя описать новую структуру и тут же использовать её в выражении или в качестве типа для переменной – что, конечно, не очень хорошо (хоть и не так страшно). Зато объявление именованого типа не обязательно завершать точкой с запятой, и это стало решающим фактором при выборе модели.

class C1

{

} // no need to ‘;’-terminate declaration of named types

Если спецификтор типа не указан, он считается ‘void’ по умолчанию. Это позволяет опускать спецификатор для ничего не возвращающих функций и методов, а также указателей на них; кроме того, это унифицирует грамматику и семантику объявления функций и специальных методов (конструкторов, деструкторов и тд).

Одно и то же объявление может содержать один или несколько деклараторов с опциональными инициализаторами.

main ()

{

int a, \*b, c = 10, d [10] = { ,, 3, 4 };

}

Как и в C/C++, в Jancy поддерживаются объявления псевдонимов типов (typedef)

typedef unsigned char TEthernetAddr [6];

# Категории типов данных

#### Примитивные типы

Система примитивных типов в Jancy похожа на таковую в C++, за одним исключением: нативно поддерживаются целочисленные типы с обратным порядком следования байтов (big-endian).

* void
* bool
* int8
* unsigned int8
* int16
* unsigned int16
* int32
* unsigned int32
* int64
* unsigned int64
* bigendian int16
* bigendian unsigned int16
* bigendian int32
* bigendian unsigned int32
* bigendian int64
* bigendian unsigned int64
* float
* double

#### POD-типы

Простая структура данных (POD, plain-old-data) – это термин, обозначающий структуру данных, которая используется только для пассивного хранения значений без инкапсуляции операций над ними и допускает простое копирование байт-в-байт.

В Jancy определние POD почти такое же, но немного строже. На пальцах, под POD понимаются такие данные, которые можно смело побайтно копировать и модифицировать и при этом ничего не сломать. Все примитивные типы, а также составленные из них массивы, структуры и объединения– POD. Всё, что содержит безопасные указатели – не POD. Забегая вперёд, скажу, что в Jancy есть и небезопасные указатели для обеспечения 100% ABI-совместимости с C/С++, и они снова являются POD.

Принадлежность и не-принадлежность к категории POD влияет на доступность присвоений и приведений типов (type casts), а также на доступный диапазон адресной арифметики безопасных указателей.

#### Value-типы и Reference-типы

Как в Java, C# или D, типы языка Jancy принадлежат одной из двух категорий: value-тип и reference-тип. Объявляя переменную или поле value-типа, разработчик сразу выделяет место, в котором будут располагаться байты данных. Объявляя переменную или поле reference-типа, он выделяет место только под ссылку. Классы в Jancy reference-типа (и, как следствие, не-POD-типа). Всё остальное – value-типа.

#### Неперемещаемые (non-moveable) типы

В Jancy имеются два особенных value-типа, которые нельзя перемещать простым копированием байт-в-байт: multicast и autoev. Обсуждение этих типов состоится позже в соответствующих разделах, пока же просто отметим существование особой категории типов, которые по свойствам находятся где-то между value-типами и reference-типами: объявляя переменную, вы сразу выделяете место под байты данных, однако обращаться к ней можете только по ссылке или указателю.

#### Именованные (named) типы

В Jancy 4 вида именованных типов:

* Перечисления (enum & enumc)
* Структуры (struct)
* Объединения (union)
* Классы (class)

Я буду называть их именованными даже если у них нет имени (анонимное объявление). Разговор про структуры, объединения и классы пойдёт ниже, сейчас же отметим особенности перечислений в Jancy.

Перечисления могут быть двух подвидов. Неанонимное перечисление, объявленное с использованием ключевого слова ‘enum’ работает как перечисления из C# или D: объявленные внутри константы не засоряют родительское пространство имён, и для доступа к ним необходимо использовать квалифицированное имя с явным указанием имени перечисления. Перечисления, обявленные через ‘enumc’ или же анонимные ‘enum’-перечисления, как и перечисления в C/С++ добавляют идентификаторы констант в родительское пространство имён. Нижележащим примитивным типом по умолчанию является ‘int’, однако умолчание можно переопределить, «унаследовав» перечисление от другого типа, например ‘char’.

# Категории памяти данных

Память под данные в Jancy может быть выделена из 4 источников

* static: статическая (выделяется при загрузке модуля)
* stack: стековая
* heap: куча под управлением сборщика мусора
* uheap: куча под ручным управлением (unmanaged heap)

В Jancy 2 добавится пятый источник: ‘thread’, который в целом похож на static, но при этом каждый поток имеет свою копию данных (thread local storage).

Выделяемая память всегда 0-инициализируется.

Память под глобальные переменные всегда статическая.

Память под локальные переменные по умолчанию выделяется на стеке (‘stack’), однако также можно указать ‘static’ или ‘heap’ (‘uheap’ как источник памяти для локальной переменной использовать нельзя)

main ()

{

/\* stack \*/ int a;

static int b;

heap char c [16 \* 1024];

}

В случае со статическим выделением гарантируется в многопоточной среде единократность исполнения инициализатора, если оный имеются: компилятор генерирует код, аналогичный std::call\_once.

Указание управляемой кучи в качестве категории памяти для локальной переменной может быть полезно для выделения больших буферов в куче, а не на стеке. Конечно, аналогичного эффекта можно было бы достичь переписав код с использованием указателей и оператора ‘new’ однако элегантнее просто указать спецификатор категории памяти, не правда ли?

Память под данные класса (reference-типа) можно выделить только с помощью оператора ‘new’, причём опционально можно указывать из какого источника необходимо выделить память (по умолчанию из управляемой кучи ‘heap’)

class C1

{

}

main ()

{

C1 a = /\* heap \*/ new C1;

C1 b = static new C1;

C1 c = stack new C1;

C1 d = uheap new C1;

delete d;

}

Память, выделенную на неуправляемой куче с помощью ‘uheap new’, необходимо вручную освободить оператором ‘delete’, как и в C++.

Оператор ‘static new’ гарантирует в многопоточной среде единократное выполнение конструкторов (если оные имеются).

Операторы ‘new’ можно применять и к value-типам тоже, однако такие операторы возвращают указатель на выделенный и проинициализированный блок, поэтому отложим это обсуждение до раздела «указатели»

# Классы (class), структуры (struct) и объединения (union)

Язык Jancy реализует объектно-ориентированную парадигму в полном объёме, причём поддерживаются как классы (reference-типа), так и структуры и объединеия (value-типа). Классы могут иметь:

* Множественных предков (multiple inheritance)
  + Классы (class)
  + Структуры (struct)
  + Объединения (union)
* Поля данных (field) с опциональными инициализаторами
  + Нормальные (member)
  + Статические (static)
* Конструкторы
  + Умолчальный (default constructor)
  + С аргументами
  + Статический (static constructor)
  + Пре-конструктор (pre-constructor)
* Деструкторы
  + Нормальный
  + Статический (static destructor)
* Методы (method)
  + Нормальные (member)
  + Виртуальные (virtual & override)
  + Абстрактные (abstract)
  + Статические (static)
* Свойства (property)
  + Нормальные (member)
  + Виртуальные (virtual & override)
  + Абстрактные (abstract)
  + Статические (static)
* Перегруженные операторы
  + Унарные (unary)
  + Бинарные (binary)
  + Приведения (cast)

Поля могут быть инициализированы в момент объявления. В инициализаторах полей классов можно использовать оператор ‘member new’ – такой оператор создаёт невидимое поле, в котором будут располагаться байты данных нового объекта. Это позволяет выделять один блок памяти под несколько объектов сразу.

class C1

{

}

class C2

{

C1 m = member new C1;

foo ();

}

Реализация методов может как располагаться сразу за объявлением (Java-стиль), так и быть вынесена вне объявления класса (С++ стиль)

class C1

{

foo ()

{

printf (“C1.foo ()\n”);

}

bar ();

}

C1.bar ()

{

printf (“C1.bar ()\n”);

}

Синтаксис конструкторов и деструкторов позаимствован из D

class C1

{

this ();

this (int a);

~this ();

static this ();

static ~this ();

}

Классы могут содержать пре-конструктор – блок кода, который будет выполнен перед каждым из конструкторов, что может быть удобно при наличии у класса перегруженных конструкторов.

class C1

{

preconstruct ()

{

printf (“C1.preconstruct () is called before each constructor overload\n”);

}

this ();

this (int a);

}

Последовательность инициализации: выделенный блок памяти 0-инициализируется, затем инициализируются базовые типы в порядке наследования, затем исполняются инициализаторы нестатических полей (если есть), затем пре-конструктор (если есть), затем конструктор, соответствующий фактическим аргументам, либо умолчальный конструктор (если есть).

Статическая инициализация исполняется гарантировано единократно в многопоточной среде. Последовательность статической инициализации: статические поля 0-инициализируются при загрузке модуля, при первом создании экземпляра класса выполняется статическая инициализация базовых типов в порядке наследования, затем инициализаторы статических полей (если есть), затем статический конструктор (если есть)

Деструкторы выделенные на стеке объектов вызываются в момент выхода из области видимости (scope), что позволяет использовать парадигму RAII (resource-acquisition-is-initialization) точно так же, как и в C++.

class C1

{

this ()

{

printf ("C1.this () – acquire stuff\n");

}

~this ()

{

printf ("C1.~this () – release stuff\n");

}

}

main ()

{

stack new C1;

}

Потенциальные проблемы, связанные с сохранением указателя на стековый объект или его поля и использованием этого указателя после разрушения объекта, будут подробно разобраны в разделе «указатели»

Наследоваться классы могут от других классов, структур и объединений, поддерживается простая модель множественного наследования (без виртуального наследования С++)

Конфликты имён при множественном наследовании разрешаются в порядке объявления наследования, с возможностью явного указания через qualified name. Конфликт наследования реализации виртуальных/абстрактных функций (diamond problem) разрешается в порядке объявления наследования.

Структура в Jancy представляют собой блок данных, расположение байтов которого полностью соответствует аналогичной структуре, объявленной в C/C++ (с учётом выбранного фактора упаковки). В отличие от классов, структуры относятся к value-типу и не предваряются никакими неявными заголовками. Структуры могут иметь:

* Множественных предков (multiple inheritance)
  + Структуры (struct)
  + Объединения (union)
* Поля данных (field) с опциональными инициализаторами\*
  + Нормальные (member)
  + Статические (static)
* Конструкторы без аргументов
  + Умолчальный (default constructor)
  + Статический (static constructor)
  + Статический деструктор (static destructor)
* Невиртуальные методы (method)
  + Нормальные (member)
  + Статические (static)
* Невиртуальные свойства (property)
  + Нормальные (member)
  + Статические (static)
* Невиртуальные перегруженные операторы
  + Унарные (unary)
  + Бинарные (binary)
  + Приведения (cast)

\* инициализаторы полей структур не могут использовать ‘member new’

Структуры могут быть как POD, так и не-POD, а также как перемещаемыми, так и не перемещаемыми -- в зависимости от POD- или moveable- принадлежности полей.

При объявлении структуры (как, впрочем, и класса) можно явно указывать фактор упаковки полей; фактор упаковки полей можно также указывать в виде прагма-директивы

struct pack (1) T1

{

char a;

long b;

} // sizeof (T1) == 5

#pack (2)

// fields in subseqent declarations will be packed using 2-byte alignment

#pack (default) // restore default for this compilation unit

Структуры могут иметь битовые поля, как и в C/C++

struct T1

{

unsigned long a:1;

unsigned long b:1;

unsigned long c:14;

unsigned long d:16;

}

Объединения в Jancy аналогичны структурам, за следующими исключениями:

* Все поля должны быть POD-типа, соответвенно все объединения – POD
* Данные всех полей начинаются с одного адреса
* Наследоваться объединения не могут
* Инициализатором может обладать максимум одно поле

# Указатели

Переходим к описанию безопасных указателей: одного из основных нововведений Jancy.

Указатели всегда считались небезопасным инструментом, способным в более благоприятном варианте вызвать крах программы, а в худшем – молча повредить данные пользователя. Однако полностью отказаться от указателей просто невозможно. Указатели есть даже в языках-без-указателей (таких как Java или Visual Basic). Это указатели на классы и интерфейсы. И хотя ограничением доступных разработчику видов указателей и операций над ними можно значительно упростить обработку неблагоприятных ситуаций во время исполнения (runtime) -- очевидно, что полностью переложить анализ на этап компиляции невозможно. Обращения по невалидным указателям могут и будут случаться во время исполнения, как бы ни был спроектирован язык (если, конечно, не придираться к словам и не рассматривать вырожденные случаи). Поэтому прежде всего оговорим, что понимается под «безопасностью».

Безопасными будем называть указатели, разрешёнными операциями над которыми невозможно

* вызвать крах процесса
* повредить данные пользователя

Любое обращение по невалидному безопасному указателю должно быть поймано системой времени исполнения (runtime). Что с этим делать дальше – зависит от многих факторов (может быть, записать в лог и завершить приложение, или отключить скрипты и выдать сообщение пользователю, или попытаться перезапустить песочницу со скриптами и т.д.)

Приведём полную иерархию видов указателей в Jancy и затем рассмотрим каждый из них более подробно

* Указатели на данные
  + Обычные (с валидатором)
  + Тонкие (thin)
  + Небезопасные (unsafe)
* Указатели на классы
  + Обычные
  + Слабые (weak)
  + Небезопасные (unsafe)
* Указатели на функции
  + Обычные (с замыканием)
  + Слабые (weak)
  + Тонкие (thin)
  + Небезопасные (unsafe)
* Указатели на свойства
  + Обычные (с замыканием)
  + Слабые (weak)
  + Тонкие (thin)
  + Небезопасные (unsafe)
* Указатели на autoev
  + Обычные (с замыканием)
  + Слабые (weak)
  + Тонкие (thin)
  + Небезопасные (unsafe)

#### Указатели на данные

Зачем вообще нужны указатели на данные? В мире C/C++ такой вопрос даже не стоит: указатели на данные это единственный способ работы с динамической памятью. А нужны ли они в managed мире? Оказывается, да. Указатели на данные и сопустствующая им адресная арифметика это самый эффективный способ работы с бинарными данными. Попробуйте написать на Java код, разбирающий стек протоколов в Ethernet-фрейме, или создающей образ бинарного файла, пусть даже такого элементарного как BMP – и сравните полученную громоздкую и неподдерживаемую тарабарщину из обращений к элементам массива-буфера – с элегантным доступом на С/С++: через структуры, указатели и адресную арифметику.

Как использование указателей на данные и адресной арифметики уживается с требованием безопасности? Очень просто: безопасные указатели имеют валидатор, в который входит разрешённый диапазон адресов (range) и целочисленный уровень вложенности (scope level).

Jancy runtime не допускает косвенного обращения по указателю, не прошедшему тест на принадлежность разрешённому диапазону (оптимизатор LLVM должен быть в состоянии выкинуть ненужные проверки, хотя это пока тщательно не проверялось. Но даже если нет – два целочисленных сравнения на косвенное обращение – это не так страшно)

Однако несложно видеть, что проверки допустимого диапазона недостаточно в случае указателей на данные, выделенные со стека (локальные переменные или объекты, созданные через ‘stack new’):

int\*

foo ()

{

int a = 10;

return &a; // dang!

}

main ()

{

int\* p;

{

int a = 10;

p = &a;

}

\*p = 20; // dang!

}

Очевидно, указатели из вышеприведённого примера пройдут проверки на попадание в допустимый диапазон, но косвенное обращение по такому указателю ни к чему хорошему не приведёт. Возвращение указателя на стековую переменную можно запретить даже если оно опосредованное через другие переменные: утечку стекового адреса можно найти с помощью анализа потока данных. Так, кстати и попытались сделать в малоизвестном диалекте C под названием Cyclone. Однако попытка решить проблему *только* статическим анализом на этапе компиляции обречена на провал:

int g\_GlobalInt;

class IGuessWhat

{

abstract

int\*

ArgumentOrGlobal (int\* p);

}

int\*

foo (IGuessWhat g)

{

int a;

return g.ArgumentOrGlobal (&a); // dang?

}

Поэтому недопускание утечки стековых адресов необходимо производить также во время исполнения. В настоящее время компилятор Jancy не проводит анализ потока данных и как следствие на этапе компиляции отлавливает только простейшие случаи явной утечки адресов за области видимости – что будет исправлено в Jancy 2. Runtime, однако, не допустит сохранения стекового адреса в ячейке памяти с *меньшим* уровнем вложенности (scope level). Чтобы правильно определить уровень вложенности конкретного указателя с учётом стека вызовов (call stack), Jancy runtime поддерживает невидимую целочисленную TLS (thread local storage) переменную, которую обновляет перед каждым вызовом каждой функции.

Два этих механизма: проверка диапазона при косвенном обращении по указателю и проверка уровня вложенности при записи указателя в ячейку памяти вместе гарантируют безопасность косвенных обращений.

int\*

foo (int\* p)

{

return p + 1;

}

main ()

{

int a [10];

int i = 11;

int x = a [i]; // ERuntimeError\_LoadOutOfRange

int\* p = a;

p--;

\*p = 20; // ERuntimeError\_StoreOutOfRange

{

int b;

p = &b; // ERuntimeError\_ScopeMismatch

}

p = foo (a); // no problem

}

Алгоритм назначения диапазона допустимых адресов безопасного указателя на данные учитывает ряд факторов, как то: порождён ли указатель из переменной или другого указателя, если это указатель на поле, является ли родительский тип POD или нет, и т.д.

Для обеспечения 100% совместимости с ABI хостового C/C++ приложения Jancy также поддерживает небезопасные (unsafe) указатели на данные – которые занимают столько же места и ведут себя так же, как и в C/C++. Опасные указатели не имеют валидаторов, а значит, не проверяются ни на диапазон, ни на уровень вложенности, и, кроме того, являются POD.

foo ()

{

int a = 10;

unsafe int\* p = &a;

\*(p + 2) = rand (); // have a happy return

}

Конверсия из безопасных указателей в опасные разрешена, обратно – нет.

Первые эксперименты с реализацией безопасных указателей вместе с лёгкой эйфорией от осознания работоспособности алгоритма принесли понимание того, что лобовое применение валидаторов порождает слишком громоздкий и нечитабельный LLVM IR (intermediate representation). Поэтому решено было увеличить читаемость промежуточного представления и разгрузить оптимизатор LLVM с помощью отказа от безусловной генерации валидаторов в тех случаях, когда валидатор может быть легко восстановлен: для переменных и их полей, для указателей на поля класса внутри нестатических методов и т.д. Причём восстановление валидатора зачастую вообще может не потребоваться! Так появился третий подвид указателей на данные: тонкий указатель (thin). Тонкие указатели безопасны, в то же время они не загромождают промежуточное представление генерацией и пересылкой валидаторов.

Тонкие указатели на данные существует только на уровне компилятора и недоступны разработчику для объявления переменных или полей.

#### Указатели на классы (class\*)

Классы в Jancy являются reference-типами, то есть объявляя любую переменную или поле с типом класса разработчик де-факто объявляет указатель.

class C1

{

int m\_x;

int m\_y;

int m\_z;

}

main ()

{

C1 c;

// sizeof (c) != sizeof (C1);

// sizeof (c) == sizeof (unsafe void\*);

}

Указатель на класс безопасен, хотя и не содержит валидатора: к указателям на классы нельзя применять адресную арифметику (а значит хранить допустимый диапазон не надо), а уровень вложенности (scope level) содержится в заголовке объекта. Этот уровень вложенности используется при проверках разрешённости присвоений указателей на классы, а также для реконструкции валидаторов указателей на поля. Проверка на NULL при обращении по указателю на класс, разумеется, также производится.

Как и в случае с указателями на данные, имеются опасные (unsafe) указатели на классы, которые никак не проверяются на валидность и являются POD-типами.

Тонких указателей на классы не существует (валидатора нет и так, куда уж ещё тоньше?). Однако существует особый вид указателей, неприменимый к обычным данным: слабые (weak) указатели. Слабые указатели не влияют на время жизни объекта. Если на объект ссылаются только слабые указатели, при следующей сборке мусора его члены-указатели не будут оттрассированы, память под него будет по возможности освобождена, и будет вызван деструктор (если таковой имеется). Оставлены будут только поля заголовка, необходимые для корректного функционирования существующих слабых указателей.

Очевидно, по слабому указателю нельзя напрямую обратиться к объекту (ведь он может быть уже разрушен к этому моменту). Поэтому единственная операция, которую можно произвести со слабым указателем – это приведение (cast) к обычному, сильному указателю на класс. Если объект уже разрушен, то возвращённый указатель будет нулевым. В противном случае, будет возвращен сильный указатель.

class C1

{

}

main ()

{

C1 c = new C1;

weak C1 w = c;

c = null;

// do a lot of time- and memory- consuming stuff

c = w;

if (c)

{

// object is still alive!

}

}

Зачем и где нужны слабые указатели? Очевидно, там, где с одной стороны требуется запомнить объект, чтобы обратиться к нему позже по наступлении какого то события, но с другой стороны, при этом не «удерживать» его и всё на что он ссылается. Классическим примером будет модель взаимодействия «событие-подписчик»

Несколько слов про синтаксис объявлений указателей на классы. Класс это reference-тип, а значит любое объявление переменной (или поля) с типом класса будет де-факто объявлением с типом указатель на класс. Эта неявная конверсия типов порождает неоднозначность в случаях, когда требуется получить именно тип класса, а не тип указателя на класс. В объявлениях переменных или полей всё однозначно, но что делать, например с reflection?

class C1

{

}

main ()

{

C1 c; // obviously, we mean a class pointer here

CType t = typeof (C1); // what about this? should it be a class? or a class pointer?

}

Для случаев, когда может быть использован как тип класса, так и тип указателя на класс, был введён обобщённый синтаксис объявляние указателей на класс:

class C1

{

}

main ()

{

C1 class\* c; // same as: C1 c;

C1 weak class\* w; // same as: weak C1 w;

}

Cоответственно, в предыдущем примере typeof (C1) возвращает тип класса. Чтобы получить тип указателя на класс, необходимо написать typeof (С1 class\*).

#### Const-корректность

С понятием указателя рука в руку идёт понятие const-корректности. К сожалению, в Java и C# решили полностью отказаться от поддержки этого замечательного инструмента, позволяющего отлавливать целые классы логических ошибок на этапе компиляции и проектировать более дуракобезопасные библиотеки. Однако Уолтер Брайт принял верное решение восстановить справедливость, и const-корректность вернулась в языке D в виде модификатора ‘immutable’.

В Jancy, как языке, поддерживающем указатели, const-корректность также реализована. Там, где необходимо обеспечить гарантию неизменности значений, доступных через указатель, можно снабдить указатель модификатором ‘const’. Понятно, что данный модификатор применим *только* к указателям на данные и к указателям на классы.

foo (const int\* p)

{

int x = \*p; // no problem

\*p = 10; // error: cannot write to a const location

}

class C1

{

int m\_Field;

mutable int m\_MutableField;

int property BufferSize;

}

foo (const C1 c)

{

int x = c.m\_Field; // no problem

c.m\_Field = 10; // error: cannot write to a const location

c.m\_MutableField = 20; // no problem

x = c.BufferSize; // no problem

c.BufferSize = 100; // error: cannot cast from ‘C1 const class\*’ to ‘C1 class\*’

}

Так как нестатические методы имеют один дополнительный неявно создаваемый аргумент ‘this’ типа указатель на класс, необходим механизм указания неизменяемости ‘this’. Я решил использовать подход C++, хотя никогда не считал синтаксис С++ в части объявления константных методов особенно логичным.

class C1

{

foo () const; // const-method, same as C++

bar ()

}

foo (const C1 c)

{

c.foo (); // no problem

c.bar (); // error: cannot cast from ‘C1 const class\*’ to ‘C1 class\*’

}

# Функции, замыкания (closures), мультикасты (multicasts)

Понятие функции является ключевым в любом императивном языке, поэтому не будем давать определений и приводить очевидные примеры, и вместо этого сконцентрируемся на различиях.

В отличие от Java и C#, (и так же, как в C/C++ или D), в Jancy разрешено объявлять глобальные функции, а не только методы классов.

Модуль может иметь конструктор и деструктор – функции, которые будут выполняться после загрузки / перед выгрузкой модуля.

// global scope

this ()

{

// module constructor

}

~this ()

{

// module destructor

}

Переменное количество параметров разрешено, но только в небезопасном стиле C/C++. Безопасный vararg вместе со встроенной поддержкой вариантов (variants) будет реализован в Jancy 2.

int

printf (

unsafe const char\* pFormat,

unsafe ...

);

Jancy поддерживает произвольные умолчальные выражения для опциональных (optional) аргументов. В отличие от остальных C-подобных языков, разрешается произвольное чередование опциональных и обязательных аргументов в списке. Соответственно, при вызове функции, разрешается опускать опциональные аргументы как в конце списка, так и в его начале или середине.

foo (

int a = 1,

int b = 2,

int c, // required arguments can follow optional ones

int d = 4

)

{

printf ("foo (%d, %d, %d, %d)\n", a, b, c, d);

}

main ()

{

// it is allowed to omit optional arguments anywhere (head, mid-list, tail)

foo (,, 300); // foo (1, 2, 300, 4)

}

#### Указатели на функции (function\*)

Я уже признавался в любви к синтаксису C? Синтаксис C действительно краток и элегантен. Но вложенные деклараторы?? Бррр...

Понятно, что вложенные деклараторы (nested declarator) появились как способ разрешить неоднозначности при объявлении указателей на функции, которые в свою очередь возвращают указатели. Возможно, снова на функции. Которые возвращают указатели. На указатели на функции. Ну вы поняли.

К счастью, вложенные деклараторы это не единственное решение проблемы. В Jancy используется другой подход, покрасивше. Но который при этом нисколько не сужает общность объявлений. Тут так же можно объявить указатель на указатель на функцию, который возвращает указатель на функцию, который возвращает указатель и т.д.

foo ();

int\*

bar (int x)

{

printf (“bar (%d)\n”, x);

return null;

}

int\*

function\*

ChooseBar () (int)

{

return bar;

}

main ()

{

function\* pfn () = foo; // boooring

int\* function\* pfn2 (int) = bar;

int\* function\* function\* pfn3 () (int) = ChooseBar; // hmmm.. keep going

int\* function\* function\* pfn4 () (int) = ChooseAnotherBar;

int\* function\* function\*\* ppfn [2] () (int) = { &pfn3, &pfn4 }; // now we are talking!!

(\*ppfn [0]) () (100); // bar (100)

}

Конечно, совершенно необязатльно писать эти страшные строки каждый раз, когда надо использовать хитрый указатель на функцию: как и в C/С++, программист может использовать старый добрый typedef.

typedef double function\* FBinaryOp (double, double);

bool

ApplyToDataSet (

FBinaryOp pfnPreOp,

FBinaryOp pfnPostOp

);

#### Замыкания

Нормальные указатели на функции на самом деле содержат 2 указателя. Первый из них – конечно, собственно указатель на функцию. Второй – указатель на объект-замыкание, и тут требуются пояснения. Рассмотрим следующий пример:

class C1

{

int m\_Field = 100;

foo (int x)

{

printf (“C1.foo (%d) { m\_Field = %d }\n”, x, m\_Field);

}

}

main ()

{

C1 c = stack new C1;

function\* pfn () = c1.foo;

pfn ();

}

Если мы хотим сохранить и впоследствии вызвать указатель на метод класса C1, то мы обязаны сохранить и указатель на объект – экземпляр класса. В противном случае, как метод получит доступ к полям объекта? Тот самый указатель на объект-замыкание в самом простейшем случае будет указывать на родительский объект метода.

В общем случае, нормальный указатель на фукнцию – это указатель на нестатический метод класса-замыкания плюс указатель на объект-экземпляр этого класса.

Уверен, многие в этот момент пролистают текст вверх до примера со страшными указателями и зададут вопрос – как же так? Ведь в примерах используется один и тот же тип указателя и в случае глобальной функции, и в случае функции-метода? Да, один и тот же. Сигнатуры аргументов глобальной функции и функции-метода разные? Да, разные: нестатический метод, очевидно, принимает один дополнительный неявный аргумент ‘this’. Что, во время вызова указателя проверяется, NULL-ли объект-замыкание и соответственно этому строится ветвление глобальная функция либо функция-метод? Нет, не проверяется.

Разгадка кроется в том, что компилятор Jancy способен строить необходимые функции-переходники (thunk functions) при приведениях (cast) указателей на функции. В примере с указателем на глобальную функцию будет построен переходник с одним неиспользуемым аргументом ‘this’ и прямым вызовом оригинальной функции foo. Если сигнатура аргументов не совпадает, но возможна конвертация – будет построен переходник, выполняющий требуемые приведения аргументов. Если же прямой вызов оригинальной функции невозможен (например, мы пытаемся присвоить один указатель на функцию другому, а сигнатуры аргументов не совпадают), то будет построен переходник выполняющий конвертацию, плюс объект-замыкание, содержащий поле, в которое будет положен оригинальный указатель.

class C1

{

foo (double);

}

foo (double);

main ()

{

C1 c = new C1;

function\* pfn (double) = c.foo; // no thunk generated, closure-object is actually ‘c’

function\* pfn2 (double) = foo; // simple thunk with unused ‘this’

function\* pfn3 (int) = foo; // thunk with unused ‘this’ and conversion int -> double

function\* pfn4 (int) = pfn; // thunk with closure-object containing ‘pfn’ and conversion int -> double

}

Отмечу, что замыкания в Jancy имеют несколько другой смысл, чем замыкания в C# или D. Обычно под замыканием понимается указатель на вложенную функцию, способную обращаться к локальным переменным родительской. Однако существует и более общее понятие замыкания функции – это функция с захваченным контекстом: функция с замыканием получает аргументы из двух источников: аргументы, переданные в момент вызова и некоторые другие значения, сохранённые в момент создания замыкания. В свете этого определния замыкания в Jancy это именно замыкания. Ну и кроме того, когда вложенные функции с замыканиями в классическом понятии C# или D будут реализованы в Jancy 2, использовать они будут вот этот же самый механизм.

#### Оператор замыкания (closure operator)

Логическим развитием идеи с замыканиями и функциями-переходниками стало появление оператора замыкания. Раз при конверсии указателя на функцию мы способны создать объект-замыкание и сохранить в нём оригинальный указатель, что мешает сохранить, скажем, ещё и первый аргумент? Ответ: ничто не мешает!

foo (

int a,

int b,

int c,

int d,

)

{

printf (“foo (%d, %d, %d, %d)\n”);

}

main ()

{

function\* pfn (int, int) = foo ~(, 2,, 4); // capture the 2nd and the 4th arguments

pfn (1, 3); // foo (1, 2, 3, 4)

}

Применяя оператор замыкания, программист способен захватывать значения аргументов, тем самым фактически проводя редукцию количества аргументов выходного указателя на функцию.

#### Оператор планировки (schedule operator)

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 новосозданном потоке, в уже существующем рабочем потоке, под захваченным локом/мутексом, в обработчике оконного сообщения, и т.д.)

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

class IScheduler

{

abstract

Schedule (function\* pfn ());

}

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 closure object и передаёт полученный указатель в IScheduler.Schedule (). Что с ним будет происходить дальше – зависит, конечно, от конкретного Schedulerа. Реалистичным примером будет добавление данного указателя в очередь запросов к рабочему потоку и установка семафора. Или вызов под захваченным мутексом. И т.д.

class CWorkerThread: IScheduler

{

bool

Start ();

void

Stop ();

override

Schedule (function\* pfn ());

}

OnComplete (int Status)

{

printf (“OnComplete (%d)\n”, Status);

}

StartAsyncJob (function\* OnComplete (int Status));

main ()

{

CWorkerThread CompletionThread = new CWorkerThread;

CompletionThread.Start ();

StartAsyncJob (OnComplete @ CompletionThread);

}

Уверен, системные программисты оценят возможность передать указатель на обработчик завершения, который автоматически будет вызван в том окружении, а котором надо.

#### Слабые (weak), тонкие (thin) и опасные (unsafe) указатели на функции

Слабые (weak) указатели на функции будут реализованы в Jancy 2, однако по принципу работы они, естественно, будут походить на слабые указатели на классы. Вызвать слабый указатель на функцию нельзя. Можно только попробовать привести его к сильному. Если все слабые аргументы в замыкании всё ещё живы, то будет возвращён сильный указатель, в противном случае – null.

Тонкий (thin) указатель на функции это нормальный указатель на функцию минус указатель на объект-замыкание. В ряде случаев замыкание функции просто не нужно, достаточно обычного указателя на саму функцию – так, как он понимается в C/C++. В этом случае в Jancy можно воспользоваться тонким указателем. Тонкий указатель безопасен, т.е. проверки на NULL продолжают производиться, однако занимать в памяти он будет ровно столько, сколько и аналогичный указатель на функцию в C/C++.

foo (int x)

{

printf (“foo (%d)\n”, x);

}

bar (int x)

{

printf (“bar (%d)\n”, x);

}

main ()

{

thin function\* a [2] (int) = { foo, bar };

a [0] (10); // foo (10);

a [1] (20); // bar (20);

}

Приведение нормального указателя с замыканием к тонкому указателю с идентичной сигнатурой в общем случае невозможно. Обратное преобразование разрешено – разумеется, при идентичных (или приводимых) аргументах.

Так же как и с другими видами указателей, имеется небезопасный вид указателей на функции, который не проверяется на NULL и является POD-типом.

union T1

{

int m\_Int;

unsafe function\* m\_pfn ();

}

main ()

{

T1 t;

t.m\_Int = rand ();

t.m\_pfn (); // yeah i’m also curious what’s gonna happen

}

Добавим, что и у функций, и у указателей на функции разрешается явно указывать модель вызова (calling convention), что может быть полезно при организации взаимодействия скрипта с хостовым приложением

typedef int thin stdcall function\* FMessageBox (

intptr,

unsafe const char\*,

unsafe const char\*,

int

);

foo (FMessageBox pfnMsgBox)

{

pfnMsgBox (0, “when calling foo ()”, “host app could pass actual WINAPI function MessageBox”, 0);

}

#### Мультикасты (multicast)

Мультикаст позволяет построить набор указателей на функции и вызвать их все сразу. Основным (хотя, конечно, не единственным) применением мультикастов является реализация модели «событие-подписчик». Мультикаст предстваляют собой неперещаемую (unmovable) структуру данных, содержащую массив указателей на функции и хэш-таблицу, выступающую в роли handle-table. Операциями, применимыми к мультикасту, являются

* Добавить (Add)
* Удалить (Remove)
* Установить (Set)
* Очистить (Clear)
* Сделать снимок (Snapshot)
* Вызвать (Call)

Объявление мультикаста очень похоже на объявление указателя на функцию (ведь оно и должно однозначно определять, указатели какого типа будут храниться в данном мультикасте!)

foo ()

{

printf (“foo ()\n”, x);

}

bar ()

{

printf (“bar ()\n”, x);

}

main ()

{

multicast m ();

m += foo;

m += bar;

m ();

m = null;

thin multicast m2 (); // this multicast keeps thin function pointers

m2.Add (foo); // same as: m2 += foo;

m2.Add (bar); // same as: m2 += bar;

m2.Call (); // same as: m2 ();

m2.Clear (); // same as: m2 = null;

}

Мультикаст потокобезопасен – любые операции кроме вызова производятся под локом (который имеется в каждом мультикасте). Вызов мультикаста также потокобезопасен: перед вызовом будет сделан снимок (под локом), после чего лок будет снят и вызван этот самый снимок.

Вызов мультикаста возвращает ‘void’ – в отличие от вызова делегата в C#, который передаёт значение, возвращённое последним методом в списке.

Мультикаст может быть приведён к указателю на функцию, и тут имеется неоднозначность:

foo ()

{

printf (“foo (%d)\n”, x);

}

bar ()

{

printf (“bar (%d)\n”, x);

}

main ()

{

multicast m (int);

m += foo;

function\* pfn = m;

m += bar;

pfn (); // is it foo () and bar () or just foo ()?

}

В Jancy возможны оба варианта.

По умолчанию происходит динамическая привязка к исходному мультикасту, то есть в вышеприведённом примере будут вызваны и foo (), и bar ().

А чтобы были вызваны *только* те указатели, которые были добавлены в мультикаст m в момент приведения, нужно сделать снимок (snapshot) мультикаста:

main ()

{

multicast m (int);

m += foo;

function\* pfn = m;

function\* pfn2 = m.Snapshot ();

m += bar;

pfn (); // foo () and bar ()

pfn2 (); // foo () only

}

Удаление указателей из мультикаста сделано не так, как удаление методов из делегата в C#. В C# чтобы удалить метод из делегата, надо применить оператор ‘-=’ с тем же аргументом, с котороым был ранее вызван ‘+=’. В Jancy из-за автоматической генерации функций-переходников и неявному созданию объектов-замыканий на лету, эффективный поиск по аргументу Add () просто невозможен в общем случае.

Поэтому для возможности эффективного удаления произвольных указателей из мультикастов Jancy реализует модель ‘handle-table’. Метод Add добавляет запись в хэш таблицу и возвращает целочисленный cookie, сохранив который, разработчик впоследствие может удалить данный указатель из мультикаста.

main ()

{

multicast m (int);

intptr Cookie = m.Add (foo); // same as: intptr Cookie = m += foo;

m += bar;

m (); // foo () and bar ();

m -= Cookie;

m (); // bar () only

}

Почему возвращённое целочисленное значение в Jancy называется cookie а не handle? Как уже было сказано, основное применение мультикастов – события. Обработчик события по английски будет “event handler”, и просто чтобы избежать мешанины из похожих слов с разными понятиями, я решил взять название, использовавшееся Microsoft в модели событий COM в методе IConnectionPoint::Advise.

# Свойства (properties)

Свойство это нечто, что выглядит и ведёт себя как данные, но позволяет инкапсулировать действия, выполняемые при считывании и записи. Функции, инкапсулирующие такие действия, называются аксессорами (accessors). Аксессор чтения свойства называется геттером (getter), записи – сеттером (setter).

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

Если свойство не имеет сеттера, то оно называется неизменяемым (const-property). В других языках программирования свойства без сеттеров называются только-для-чтения (read-only), но так как в Jancy понятия ‘const’ и ‘readonly’ сосуществуют, то переопределить устоявшиеся определения пришлось бы так или иначе. Итак, в Jancy свойство без сеттера – это неизменяемое свойство.

Объявлять свойства можно двумя способами: полное объявление и упрощённое объявление. Рассмотрим сначала полное объявление. Оно выглядит и ведёт себя во многом как объявление класса и неявно открывает пространство имён. Внутри объявления свойства разрешается иметь:

* Поля данных (field) с опциональными инициализаторами
  + Нормальные (member)
  + Статические (static)
* Специальное поле ‘value’ для autoget-свойств
* Специальное поле ‘onchange’ для bindable-свойств
* Конструкторы
  + Умолчальный (default constructor)
  + Статический (static constructor)
* Деструкторы
  + Нормальный\*
  + Статический (static destructor)
* Методы (method)
  + Нормальные (member)
  + Виртуальные (virtual)\*\*
  + Абстрактные (abstract)\*\*
  + Статические (static)
* Свойства (property)
  + Нормальные (member)
  + Виртуальные (virtual)\*\*
  + Абстрактные (abstract)\*\*
  + Статические (static)

\*Только для глобальных свойств или свойств-членов класса

\*\*Только для свойств-членов класса

Рассмотрим пример полного объявления двух свойств:

property P1 // const property

{

int

get ()

{

printf (“P1.get ()\n”);

return rand ();

}

}

property P2

{

int m\_Value = 100;

int

get ()

{

printf (“P1.get ()\n”);

return m\_Value;

}

set (int x)

{

printf (“P1.set (int %d)\n”, x);

m\_Value = x;

}

set (double x)

{

printf (“P1.set (double %d)\n”, x);

m\_Value = (int) (x \* 100);

}

}

main ()

{

P1 = 200;

int x = P1;

P2 = 50;

P2 = 0.5;

}

Допускается упрощенное объявление свойств без перегруженных сеттеров, что может быть удобно либо при описании интерфейсов (когда нужно объявить абстрактное свойство без реализации аксессоров), либо если разработчик предпочитает C++ стиль «объявление-реализация».

const int property P1; // const property

int

P1.get ()

{

printf (“P1.get ()\n”);

return rand ();

}

int P2Value = 100;

int property P2;

int

P2.get ()

{

printf (“P2.get ()\n”);

return P2Value;

}

P2.set (int x)

{

printf (“P2.set (int %d)\n”, x);

P2Value = x;

}

Очевидно, создать свойство с перегруженными аксессорами возможно только с использованием «полного» объявления.

#### Индексируемые (indexed)-свойства

Итак, свойство это нечто, что выглядит и ведёт себя как данные. А что, если мы хотим, чтобы свойство выглядело и вело себя как массив (ведь массив – это тоже вроде как данные)? Как это уживается с необходимостью инкапсулировать логику поведения в аксессорах?

Jancy поддерживает индексируемые свойства, то есть такие, аксессоры которых принимают дополнительные аргументы-индексы. В отличие от истинных массивов, аргументы-индексы не обязаны быть целочисленными и, строго говоря, не обязаны иметь смысл «индексов»: логика их использования целиком и полностью определяется аксессорами.

const int indexed property P1 (int); // const indexed property

int

P1.get (int i)

{

printf (“P1.get (%d)\n”, i);

return rand () % i;

}

property P2

{

int m\_Value [3] [3];

int

get (

int i,

int j

)

{

printf (“P1.get (%d, %d)\n”, i, j);

return m\_Value [i] [j];

}

set (

int i,

int j,

int Value

)

{

printf (“P1.set (%d, %d)\n”, i, j);

m\_Value [i] [j] = Value;

}

}

main ()

{

int x = P1 [50];

P2 [1] [2] = 30;

}

Как можно видеть, оба синтаксиса (и полный, и упрощённый) позволяют объявить индексируемое свойство. Jancy требует, чтобы все акссессоры индексируемого свойства должны иметь одинаковую сигнатуру индексных аргументов.

#### Autoget-свойства

В подавляющем большинстве случаев геттер просто возвращает значение переменной или поля, хранящей текущее значение свойства, а собственно логика поведения свойства должна выполняться в сеттере. Очевидно, что создание пустышки геттера можно переложить на компилятор. Jancy предоставляет возможность сокращённого объявления свойств такого рода

property P1

{

int value; // autoget-property

set (int x)

{

printf (“P1.set (int %d)\n”, x);

value = x;

}

}

int autoget property P2;

P2.set (int x)

{

printf (“P2.set (int %d)\n”, x);

value = x;

}

Обратная же ситуация, когда специальная логика поведения свойства заложена в геттере, а сеттер-пустышка должен просто класть данные в ячейку памяти, не настолько часто встречается, чтобы создавать под это специальный синтаксис.

Отметим, что аutoget-свойство не может быть индексируемым.

#### Связываемые (bindable) свойства

Важной категорией свойств являются связываемые (bindable) свойства: такие, которые могут оповещать подписчиков о своём изменении. Jancy предоставляет встроенную поддержку связываемых свойств.

property P1

{

int value; // autoget property

event onchange (); // bindable property

set (int x)

{

printf (“P1.set (int %d)\n”, x);

value = x;

}

}

int autoget bindable property P2;

P2.set (int x)

{

printf (“P2.set (int %d)\n”, x);

if (x > 100)

silent return;

value = x;

}

OnChangeHandler ()

{

printf (“OnChangeHandler\n”);

}

main ()

{

P1.onchange += OnChangeHandler;

P2.onchange += OnChangeHandler;

P1 = 10;

P2 = 101; // OnChangeHandler () will not be called

}

Сеттер связываемых свойств после завершения автоматически вызывает всех подписчиков события ‘onchange’. Подавить автоматический вызов подписчиков можно выйдя из сеттера с помощью специальной конструкции ‘silent return’.

#### Связываемые (bindable) данные

Рассмотрим последний вырожденный случай: простейшее связываемое autoget-свойство, в котором сеттер-пустышка просто кладёт аргумент в ячейку ‘value’. В отличие от простых autoset-свойств, назначение которых туманно и потому не заслуживает специального синтаксиса, *связываемое* autoget-autoset свойство может быть полезно для хранения данных, изменение которых надо отслеживать. Поэтому Jancy предоставляет сокращённый синтаксис для объявления таких свойств (которые будем называть связываемые данные)

bindable int P1;

OnChangeHandler ()

{

printf (“OnChangeHandler\n”);

}

main ()

{

P1.onchange += OnChangeHandler;

P1 = 10; // OnChangeHandler will be called

}

#### Указатели на свойства (property\*)

Указателей на свойства пока нет ни в одном языке. Поэтому сразу начну с приведения примера, когда указатели на свойства могут быть полезны.

class CFont

{

bool property IsBold;

bool property IsItalic;

bool property IsUnderlined;

// ...

}

// we have a checkbox control which can dynamically update it’s target

class ICheckBoxTarget

{

abstract

bool

Read ();

abstract

Write (bool Value);

}

class CAutoCheckBox

{

ICheckBoxTarget m\_Target;

}

class CFontDialog

{

CAutoCheckBox m\_BoldCheckBox;

CAutoCheckBox m\_ItalicCheckBox;

CAutoCheckBox m\_UnderlinedCheckBox;

Create (CFont Font)

{

// hmm.. now how do we connect XxxCheckBox to Font.IsXxx property?

// create 3 thunks (one ICheckBoxTarget-derived class per property)?

}

}

Понятно, что приведённый пример со шрифтом немного искусственный. Можно реализовать всё то же самое **и** по-другому, с использованием событий, или просто считывая / записывая свойства шрифта в OnInitDialog / OnOK, да мало ли ещё как – в конце концов, диалоги шрифтов каким-то образом создавали и раньше.

Однако модель, продемонстированная в этом примере, довольно распространена: один компонент-мастер (контроллер) динамически устанавливает свойства / вызывает методы другого компонента-подчинённого, причём свойства и методы – переопределямые, что позволяет одному мастеру работать с целым множеством подчинённых. Данная модель взаимодействия реализуется с помощью введения интерфейса, с которым умеет работать контроллер. Но иногда такой подход может приводить к созданию большого количества не несущих смысловой нагрузки переходников.

С использованием указателей на свойства, проблема легко и элегантно решается.

class CAutoCheckBox

{

bool property\* m\_pTarget;

}

class CFontDialog

{

CAutoCheckBox m\_BoldCheckBox;

CAutoCheckBox m\_ItalicCheckBox;

CAutoCheckBox m\_UnderlinedCheckBox;

Create (CFont Font)

{

m\_BoldCheckBox.m\_pTarget = &Font.IsBold;

m\_ItalicCheckBox.m\_pTarget = &Font.IsItalic;

m\_UnderlinedCheckBox.m\_pTarget = &Font.IsUnderlined;

}

}

Так же, как и с функциями, Jancy умеет создавать свойства-переходники для автоматических конверсий – в частности, для порождения указателей на свойства из указателей на данные! На практике это означает что возможно проворачивать трюки вроде:

struct TTcpHdr

{

// ...

uchar\_t m\_Fin:1;

uchar\_t m\_Syn:1;

uchar\_t m\_Rst:1;

// ...

}

class CTcpPacketDialog

{

CAutoCheckBox m\_FinCheckBox;

CAutoCheckBox m\_SynCheckBox;

CAutoCheckBox m\_RstCheckBox;

Create (TTcpHdr\* pTcpHdr)

{

m\_FinCheckBox.m\_pTarget = &pTcpHdr->m\_Fin;

m\_SynCheckBox.m\_pTarget = &pTcpHdr->m\_Syn;

m\_RstCheckBox.m\_pTarget = &pTcpHdr->m\_Rst;

}

}

Вообще, указатели на свойства во многом похожи на указатели на функции: ведь свойство с точки зрения реализации свойство и представлят собой набор функций-аксессоров. Но разница, безусловно имеется.

Так как имеется не одна функция, а набор функций-аксессоров, значит первая часть указателя на свойство это не указатель на функцию, а указатель на таблицу функций.

Обращение к указателю на свойство, в отличе от указателя на функцию, может вызывать неоднозначность. Например, если у нас есть указатель на свойство, которое возвращает указатель на свойство, и т.д. И мы кладём null в такой указатель – куда он будет записан?

foo (int\* property\* property\* NobodyIsGonnaDoThisButStill)

{

NobodyIsGonnaDoThisButStill = null; // where does ‘null’ go to?

}

Неоднозначность, понятно, связана с неявным вызовом аксессора при обращении к свойству. Для решения неоднозначности, Jancy требует явной операции взятия адреса ‘&’ или разыменования ‘\*’ при работе с указателями на свойства (так же как и при работе с указателями на данные). К слову сказать, операции взятия адреса и разыменования так же применимы и к указателям на функции (просто там они необязательны в силу отсутствия неоднозначности.

foo (int\* property\* property\* NobodyIsGonnaDoThisButStill)

{

\*\*NobodyIsGonnaDoThisButStill = null; // we are setting the deepest int\* property

function\* pfn = &bar; // same as: pfn = bar

(\*pfn) (); // bar ()

(\*\*pfn) (); // no can do – unlike C/C++

}

bar ()

{

return printf (“bar ()\n”);

}

Замыкания для индексированных свойств создаются не с помощью специального оператора замыкания, а естественным образом с помощью оператора индексации:

const int indexed property P1 (int i)

int

P1.get (int i)

{

printf (“P1.get (%d)\n”, i);

return rand () % i;

}

main ()

{

int property\* py = &P1 [100];

int x = \*py; // P1.get (100)

}

Аугментированные свойства (autoget и bindable), помимо указателя на таблицу аксессоров и указателя на объект-замыкание, требуют так же и указателя на блок аугментации. Как следствие, объявление указателя на аугментированные свойства должно указывать, чем именно аугментировано свойство.

OnChangeHandler (const int property\* py)

{

printf (“OnChangeHandler () Value = %d\n”, \*py);

}

foo (int bindable property\* py) // bindable-augmented property pointer

{

py.onchange += OnChangeHandler ~(py);

\*py = 100;

}

Если требуется описать указатель на свойство в самом общем случае (например, с перегруженными сеттерами, или если тип параметра сеттера которого отличается от типа значения, возвращаемого геттером), – имеется специальный синтаксисис для описания шаблона свойства (property template)

property P1

{

int value;

set (int x)

{

value = x;

}

set (double x)

{

value = (int) (x \* 100);

}

}

main ()

{

property // property template

{

int

get ();

set (int);

set (double);

}\* py = &P1;

typedef property

{

int value; // autoget-augmented property template

set (int);

set (double);

} PAutoGetOverloadedProperty;

PAutoGetOverloadedProperty\* py2 = &P1;

}

В предыдущем примере можно использовать оба шаблона: указатель на аугментированное свойство, конечно, может быть приведён к указателю на обычное свойство (но не наоборот). Указатель на аугментированное свойство будет работать эффективнее, так считывание будет реализовано не через косвенный вызова геттера из таблицы аксессоров, а через косвенное считывание указателя на value.

Так же, как и указатели на функции, указатели на свойства могут быть слабыми, тонкими и небезопасными. Cмысл этих понятий и ситаксис объявлений в применении к указатлям на свойства такой же, как и выше, поэтому не будем пережёвывать по второму разу те же самые слова с заменой «функций» на «свойства».

# Поддержка реакционного программирования

Переходим ещё к одной интересной возможности Jancy, которая больше нигде пока не реализована – по крайней мере, в распространённых языках программирования. Начну с небольшой предыстории.

Одним из самых первых проектов, который я выполнял в начале моей работы в моей нынешней компании, был конфигуратор производимых нами устройств. Устройства имели последовательный и сетевой интерфейсы (в то время – только 10BASE-T), и как, следствие, значительный набор сетевый настроек, который постоянно расширялся. Проблемой было обеспечение обратной совместимости со стремительно нараставшим (вместе с набором поддерживаемых функций) количеством версий прошивки.

Даже если бы в те времена наши устройства имели веб сервер, подход с конфигурированием через web не являлся бы полным решением проблемы: устройство вследствие разных причин может быть просто недоступно по IP, и в этом случае необходимо выполнить настройку через последовательный порт (serial).

Выбранным нами подходом к решению данной проблемы стало введение метаконфигурации: описателя поддерживаемых настроек, и автоматизированного создания юзерского интерфейса по этому описателю (после некоторого количества экспериментов, мы остановились на property grid). Довольно быстро реализовав первую версию, мы столкнулись с естественным желанием ввести в property grid интерактивность, которую логично было бы ожидать от конфигурационного диалога сетевых настроек. Например, включение DHCP, по-хорошему должно задизэблить ручные настройки IP, DNS & Gateway; выбор режима client/server должен разрешать редактирование либо Destination IP/Port, либо Local Port, и т.д. Без ложной скромности скажу, что горжусь тем, как мы решили эту задачу.

К каждой настройке в метаконфигурации было добавлено дополнительное поле S статуса (status), который представлял собой выражение на языке C. Это выражение динамически пересчитывалось при изменении входящих в него настроек и возвращало строку, в которой была закодирована вся необходимая информация о динамическом статусе настройки: доступна ли она для редактирования, или задизэблена, или спрятана, или имеет фиксированное значение, которое нельзя редактировать, и т.д.

Этот подход позволил создавать интерактивные диалоги настроек прямо в простом текстовом редакторе, с применением минимума программирования (нужно было только прописать те самые статусные выражения), и без нужды перекомпилировать основную программу.

Каждый раз, когда мне приходится в первый раз рассказывать про данную возможность, я привожу в пример электронные таблицы, типа Microsoft Excel. В ячейках пользователь может прописать формулы, устанавливающие взаимосвязь между значениями этой ячейки и произвольных других. Когда значения ячеек, от которых зависит данная, меняются – заданная пользователем формула запустится на выполнение, и ячейка автоматически обновится.

Теперь представим на секунду, что вместо этого, каждая ячейка предоставляет событие “OnValueChange”, и пользователю необходимо вручную прописать и привязать обработчики этих событий, в каждом из которых, опять таки вручную, обновить значения зависимых ячеек. Абсурдным и искусственным данный пример перестаёт казаться, когда приходит понимание, что именно так и работают все существующие модели пользовательского интерфейса в императивных языках!

После успешного применения концепции автоматического выполнения выражений в нашем конфигураторе, я долго экспериментировал в надежде найти способ использовать эту концепцию из императивного языка типа C/C++/ Java/C#. Несмотря на очевидность реализации в случае конфигуратора, единственной «программируемостью» которого состояла возможность прописать статусные выражения – было совершенно непонятно в каком объёме, с какими допущениями, и как вообще это будет работать и взаимодействовать в самом общем случае. И тем не менее – после многочисленных фальстартов – могу сказать, что нашёл приемлемый подход для интеграции данной парадигмы в обычный императивный язык, и реализовал этот подход в Jancy.

Объявление блока автоматического выполнения в Jancy похоже на объявление функции

bindable int Red;

bindable int Green;

bindable int Blue;

Purple (int r, int b, int a)

{

printf ("Purple (%d, %d, %d)\n", r, b, a);

}

Yellow (int r, int g, int a)

{

printf ("Yellow (%d, %d, %d)\n", r, g, a);

}

Cyan (int g, int b, int a)

{

printf ("Cyan (%d, %d, %d)\n", g, b, a);

}

autoev ColorAutoEv (int Alpha)

{

UpdatePurple (Red, Blue, Alpha);

UpdateYellow (Red, Green, Alpha);

UpdateCyan (Green, Blue, Alpha);

}

main ()

{

ColorAutoEv (128); // same as: ColorAutoEv.Start (128);

Red = 64; // Purple, Yellow

Green = 128; // Yellow, Cyan

Blue = 255; // Purple, Cyan

}

Внутри блока autoev указываются выражения (не statements!), которые надо уметь пересчитать при изменении правых частей. Каждое из выражений внутри блока порождает независимую функцию, которая автоматически будет привязана к событию ‘onchange’ каждого связываемого свойства, найденного в своей правой части. В вышеприведённом примере будет сгенерировано три функции, первая привязывается к Red и Blue и вызывает UpdatePurple, вторая – к Red и Green и вызывает UpdateYellow, ну и третья – к Green и Blue и вызывакт UpdateCyan.

Аргументы, передаваемые в блок autoev, могут, конечно, использоваться по усмотрению разработчика. В качестве реалистичного примера, с помощью аргумента-указателя на класс можно привязать выражения к свойствам-членам конкретного объекта.

Если список аргументов блока autoev пуст, допускается упрощённый синтаксис:

int Color;

autoev ColorAutoEv

{

Color = RGB (Red, Green, Blue);

}

Как было сказано, в блоке autoev нельзя напрямую писать statements, такие как for или if. Однако что делать, если в ответ на изменение какого то связываемого свойства хочется, например, пробежаться в цикле по массиву и обновить его элементы? Для этой цели Jancy предоставляет специальный синтаксис:

autoev ColorAutoEv

{

onchange (Green, Blue)

for (int i = 0; i < x; i++)

printf ("Red = %d; Green = %d; Blue = %d; x = %d\n", Red, Green, Blue);

UpdatePurple (Red, Blue); // of course can also use expressions in the same autoev

UpdateYellow (Red, Green);

UpdateCyan (Green, Blue);

}

Цикл в вышеприведённом примере будет выполнен при изменении Green и Blue, но не Red (хотя Red и входит в параметры вызова printf () в теле блока onchange). В параметрах конструкции onchange перечисляются связываемые свойства (через запятую, если более одного), при изменении которых надо выполнить заданный statement (допускается использовать произвольный statement, включая, конечно, и compound statement { }). К связываемым свойствам внутри этого statement привязка не производится.

Когда автоматическое выполнение больше не требуется, autoev можно остановить с помощью метода Stop ():

main ()

{

ColorAutoEv (128);

Red = 64; // Purple, Yellow

Green = 128; // Yellow, Cyan

ColorAutoEv.Stop ();

Blue = 255; // nothing gonna be auto-evaluated here

}

В качестве средства контроля за областью применения autoev, можно использовать RAII:

class CRaiiColorAutoEv

{

this ()

{

ColorAutoEv ();

}

autoev ColorAutoEv // member autoevs are stopped in destructor

{

Purple (Red, Blue);

Yellow (Red, Green);

Cyan (Green, Blue);

}

};

main ()

{

{

stack new CRaiiColorAutoEv;

Red = 64; // Purple, Yellow

Green = 128; // Yellow, Cyan

}

Blue = 255; // nothing gonna be auto-evaluated here

}

Очевидно, генерацию подобных классов-обёрток, единственным назначением которых является контроль времени жизни, можно переложить на компилятор. Jancy предоставляет упрощённый синтаксис специально для этого случая. Следующий пример полностью аналогичен приведённому выше.

autoev class CRaiiColorAutoEv

{

Purple (Red, Blue);

Yellow (Red, Green);

Cyan (Green, Blue);

}

main ()

{

{

stack new CRaiiColorAutoEv;

Red = 64; // Purple, Yellow

Green = 128; // Yellow, Cyan

}

Blue = 255; // nothing gonna be auto-evaluated here

}

На данный момент, самой вероятной областью применения парадигмы автоматического выполнения является пользовательский интерфейс. Пользователь пощёлкал какие-то свойства, надо автоматически обновить всё, что от этого зависит. Однако скорее всего, со временем найдутся и другие ситуации, в которых применение данного подхода может оказаться полезным.

# Пространства имён и модель контроля доступа

Как C++ и C#, Jancy поддерживает глобальные пространства имён; как и в C#, можно создавать и открывать сразу несколько пространств имён за раз. Одно и то же пространство имён можно открывать и закрывать многократно в пределах одного юнита компиляции для добавления в него новых элементов.

namespace a.b.c {

namespace d {

} // namespace a.b.c.d

} // namespace a.b.c

namespace a.b.c.d {

} // namespace a.b.c.d

Все именованные типы: перечисления (enum), структуры (struct), объединения (union), классы (class) в Jancy содержат неявно создаваемое пространство имён, в которое помещаются все члены данного типа.

Пространство имён любого класса, структуры или объединения может быть расширено (с ограничениями) после объявления типа (extension namespaces). Идея подсмотрена в C# и доведена до логического завершения.

class C1

{

int f;

foo ();

}

extend C1

{

bar ()

{

printf (“non-static type extension methods can access fields (f = %d)\n”, f);

}

}

main ()

{

С1 a = new C1;

a.foo ();

a.bar ();

}

Упомянутые выше ограничения призваны обеспечить инвариантность кода, работающего в присутствии расширения пространства имён и без оного. В расширении типа нельзя:

* ...создавать новые конструкторы, деструкторы или операторы
* ...перегружать уже объявленные методы
* ...создавать поля
* ...создавать виртуальные методы или свойства

Модель контроля доступа к элементам в пространствах имён в Jancy отличается от таковых в существующих объектно-ориентированных языках.

* Модификаторов доступа всего два
  + Публичный (public)
  + Защищённый (protected)
* По умолчанию модификатор доступа public
* Члены глобальных пространств имён могут иметь модификатор доступа, как и члены пространств имён именованых типов
* Модификатор доступа может быть указан непосредственно при объявлении элементы (Java-стиль), или же для группы элементов (C++-стиль)

Главным изменением, безусловно, является редукция количества модификаторов доступа до двух.

Для каждого отдельно взятого пространства имён весь остальной мир делится на две категории: «свои» и «чужие». К «своим» относятся:

* дочерние пространства имён
* пространства имён классов или структур, унаследованных от данного именованого типа
* расширения пространства имён (extension namespaces)
* пространства имён, объявленные как дружественные (friend)

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

С одной стороны, это предоставляет разработчикам значительно меньшую гибкость в определении того, кто и к кому имеет доступ. И это, конечно, минус. Но с другой стороны – и это стало решающим фактором, – упрощённая модель «свой-чужой» открывает возможность дуальных модификаторов: т.е. модификторов, имеющих разное значение для «своих» и «чужих».

#### Дуальный модификатор ‘readonly’

В Jancy ограничение доступа к полю класса только на чтение может быть решено без написания каких либо геттеров:

class C1

{

readonly int f;

foo ()

{

f = 10; // no problem

}

}

main ()

{

C1 a = stack new C1;

int x = a.f; // no problem

a.f = 20; // error: cannot assign to const-location

}

Для «своих» модификатор ‘readonly’ как бы невидим; для «чужих» поле, объявленное с модификатором ‘readonly’ трактуется как неизменяемое (см. const-корректность).

Это во-первых, делает код короче и элегантнее; во-вторых, упрощает работу оптимизатору backend-а, которому не нужно анализировать и выкидывать пустышки геттеров, единственным назначением которых был контроль доступа.

#### Дуальный модификатор ‘event’

Другим примером применения дуальных модификаторов будут события. Владелец события должен иметь полный над ним контроль, включая возможность вызвать всех подписчиков или очистить их список. Клиент события должен иметь возможность *только* добавить или удалить подписчика.

class C1

{

event OnProgress (int Percentage);

Work ()

{

for (int i = 0; i < 10; i++)

{

OnProgress (i \* 10); // no problem

}

OnProgress (100); // no problem

}

}

OnProgressHandler (int Percentage)

{

printf (“OnProgressHandler (%d)\n”, Percentage);

}

main ()

{

C1 a = stack new C1;

a.OnProgress += OnProgressHandler; // no problem

a.Work ();

a.OnProgress (101); // error: ‘Call’ is not accessible

}

Для «своих» поле с модификатором ‘event’ работает так же, как и мультикаст с соответствующей сигнатурой аргументов. Для «чужих» поле с модификтором ‘event’ ограничивает доступ к методам мультикаста: разрешены только Add () и Remove (), запрещены Call (), Set () и Clear ().

# Другое

## Curly-инициализаторы

Одна из моих любимых возможностей синтаксиса С/С++ это инициализация массива или структуры сразу после объявления. Лаконично, интуитивно понятно, и при этом предельно просто в реализации в компиляторе.

int a [10] = { 1, 2, 3 };

struct TPoint

{

int x;

int y;

int z;

};

TPoint Point = { 10, 20, 30 };

Point = { 100, 200, 300 }; // alas, no can do

В С# добавили возможность использования имён членов структур в момент создания с помощью оператора new, но зачем-то убрали возможность классической инициализации

TPoint Point = new TPoint { x = 100, y = 200, z = 300 }; // nice!

Point2 = new TPoint { 100, 200, 300 }; // no can do, but why??

В Jancy я решил довести curly инициализаторы до ума при использовании в самом общем случае.

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

TPoint Point = { , 20, z = 30 };

TPoint\* pPoint = new TPoint { , 200, z = 300 };

Point = { , 2000, z = 3000 };

// all of the above -- yes can do!

Как можно видеть из примера, curly иницализаторы в Jancy

* …можно использовать после произвольных присвоений (а не только в деклараторах)
* ...а также в в операторе new
* Можно пропускать поля / элементы массива (оставляя умолчальное 0-инициализированное значение)
* Для адресации полей можно использовать как имя (C# стиль) так и индекс (C/С++ стиль), 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 чтобы избежать возможных неочевидных эффектов и неоднозначности в случаях инициализации вложенных структур:

struct TNeighborhood

{

TPoint Point;

int r;

}

TNeighborhood n = { x = 10, y = 20, 30 }; // 30 goes to z? or to r?

## Единократное выполнение кода (once)

Jancy предоставляет встроенное средство для гарантированно единократного выполнения ленивой инициализации.

CThreadPool

GetIoThreadPool ()

{

static CThreadPool Pool;

once

{

Pool = new CThreadPool;

Pool.Configure (GetProcessorCount (), EThreadPriority.Normal);

Pool.Start ();

}

return Pool;

}

Блок кода (который, конечно, не обязан быть именно блоком – это может быть выражение или любой другой произвольный statement) будет заключён в сгенерированный компилятором аналог std::call\_once (). В Jancy 2 будет добавлена TLS-вариация ‘thread once’.

## Псеводонимы выражений (alias)

Typedef позволяет создать псевдоним для типа. А как быть, если мы хотим создать псевдоним для глобальной или статической функции класса? Или переменной? Что, если в коде, например, конструктора мы много-много раз обращаемся к глубоко вложенной структуре для инициализации её полей?

Старые добрые макросы С/С++, безусловно могут решить проблему сокращения многократно повторяемого выражения. Но они не без основания считаются опасным и слабо-отлаживаемым механизмом, так как позволяют «слишком много». Кроме того, объявленные макросы, существуя на уровне пре-процессора, остаются доступны и после выхода из области видимости, то есть на выходе нужно по-хорошему явно их всех разопределить. Плюс синтаксис определения макросов не столь красиво переплетается с собственно кодом функции. Одним словом, использование макросов С/С++ для данной задачи возможно, но далеко от элегантности.

Jancy предоставляет специальный синтаксис для создания псевдонимов выражений (alias).

main ()

{

alias s = a.b.c->d.e.f;

s.x = 10; // same as: a.b.c->d.e.f.x = 10;

s.y = 20; // same as: a.b.c->d.e.f.y = 20;

s.z = 30; // same as: a.b.c->d.e.f.z = 30;

}

Такие псевдонимы ведут себя во многом как макросы, но со следующими отличиями:

* Разрешается создать псевдоним лишь для законченного выражения
* Псевдонимы являются членами текущего пространства имён или области видимости

Первое свойство в определённой мере гарантирует безопасность и предсказуемость поведения псевдонима. Нельзя составить псевдоним для куска выражения, или куска объявления, порождая специальный синтаксис «только-для-людей-кто-в-теме» (такого, как BEGIN\_EVENT\_MAP)

Второе – обеспечивает локальность определения в случаях, где она необходима.

## Break-n/Continue-n

Помню, на RSDN была целая священная война с горячими дебатами – и даже комиксами (!), навеянная топиком «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 (необосновано дорого и громоздко для такого сценария). Между тем, с точки зрения компилятора нет никакой разницы, выйти из одного цикла, или из двух циклов, или из n циклов. Просто удивительно что проектировщики языков до сих пор не реализовали такую простую вещь, которая иногда может быть крайне полезной. Jancy предоставляет операторы break <n> и continue <n>, реализующие данную семантику.

main ()

{

int a [3] [3] =

{

{ 1, 2, 3 },

{ 4, -5, 6 },

{ 7, 8, 9 },

};

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

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

break 2;

printf (“%d\n” a [i] [j]); // -5

}

## Псевдонимы выражений

Помимо старого доброго typedef, который позволяет определить псевдоним для типа, Jancy поддерживает создание псевдонимов выражений. Данной возможность была реализована с двумя целями.

Во-первых, псевдонимы выражений в Jancy можно использовать как ограниченную, но безопасную альтернативу макросам C/C++:

* Разрешается создать псевдоним лишь для законченного выражения, что в определённой мере гарантирует законченность и предсказуемость поведения псевдонима – в частности, в отличие от макросов C/C++ нельзя составить псевдоним для куска выражения, или куска объявления, порождая специальный синтаксис «только-для-людей-кто-в-теме»
* Псевдонимы являются членами текущего пространства имён или области видимости, что обеспечивает локальность определения – опять таки, в отличие от макросов C/C++, которые утекают от объявления вниз до конца юнита компиляции или до момента явного удаления с помощью #undef

foo (Struct\* s)

{

alias d = s.m\_a.m\_b.m\_c.m\_d;

d.m\_x = 10;

d.m\_y = 20;

d.m\_z = 30;

// ...

}

Во-вторых, Но главной причиной создания псевдонимов выражений стало не это, а желание иметь возможность разделять события между несколькими связываемыми свойствами. Поясню на примере пользовательского интерфейса.

Согласитесь, достаточно типовыми являются композитные контролы с несколькими методами ввода данных. Всем известный комбобокс имеет текстовое поле ввода, а также выпадающий список опций. Каждое

. А как быть, если мы хотим создать псевдоним для глобальной или статической функции класса? Или переменной? Что, если в коде, например, конструктора мы много-много раз обращаемся к глубоко вложенной структуре для инициализации её полей?

Старые добрые макросы С/С++, безусловно могут решить проблему сокращения многократно повторяемого выражения. Но они не без основания считаются опасным и слабо-отлаживаемым механизмом, так как позволяют «слишком много». Кроме того, объявленные макросы, существуя на уровне пре-процессора, остаются доступны и после выхода из области видимости, то есть на выходе нужно по-хорошему явно их всех разопределить. Плюс синтаксис определения макросов не столь красиво переплетается с собственно кодом функции. Одним словом, использование макросов С/С++ для данной задачи возможно, но далеко от элегантности.

Jancy предоставляет специальный синтаксис для создания псевдонимов выражений (alias).

Такие псевдонимы ведут себя во многом как макросы, но со следующими отличиями:

Первое свойство (такого, как BEGIN\_EVENT\_MAP)

Второе –в случаях, где она необходима.

## Scopes-in-switch

Области видимости в C++ switch statement это моя больная мозоль. Вы пишете код внутри switch, натурально хотите объявить локальную переменную типа, скажем, std::string – так же, как делаете это везде! Жмакаете Compile и получаете ошибку: инициализация переменной может быть пропущена из-за переходов на метки case или default. Приходится либо выносить переменную за switch, нарушая принцип «объявление-перед-использованием», либо городить switch-уродец с явно указанными кое-где областями видимости и нарушенной структурой табуляции.

В Java и C# компилятор не ругается на инициализацию локальных переменных в switch, но не даёт их использовать из других меток case, и при этом не разрешает их переопределить! Что мешает компилятору неявно создавать области видимости для каждой метки case? Кто и когда использовал локальную переменную, объявленную в соседней метке case?

Создатель D Уолтер Брайт, возможно, первым предложил помещать код каждой метки case в неявно создаваемую область видимости. В Jancy switch работает так же (я честное пионерское не подсматривал данную фишку в D; просто всегда хотел чтобы С/С++ делал именно так)

main ()

{

switch (rand () % 3)

{

case 0:

int i = 10;

break;

case 1:

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

case 2:

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

break;

default: // true programmers expect the unexpected! just kidding, this ‘default’ is utterly pointless

int i = 40; // still ok. you’ve got the idea.

}

}