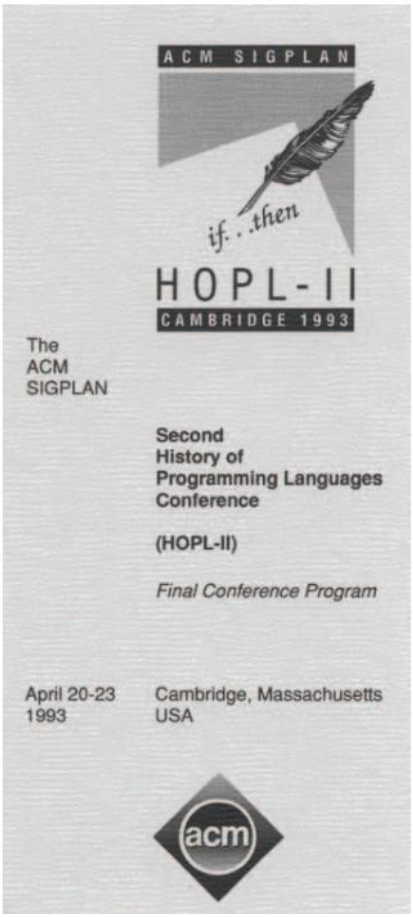
Деннис М. Ритчи

Разработка языка программирования С



27.05.2024
версия 1.0

Перевод: https://oscat.ru/

Оглавление

Предисловие

Этот документ содержит перевод статьи Денниса Ритчи «Разработка языка программирования C» и стенограммы его доклада, прочитанного на конференции History of Programming Languages II. Конференция состоялась 20-23 апреля 1993 года в Кембридже; она была организована группой SIGPLAN, входящей в состав Ассоциации вычислительной техники. Первоначальные версии материалов, посвящённые предстоящей конференции, были опубликованы в журнале SIGPLAN Notices (том 28, выпуск 3, март 1993). Финальные версии (включая стенограммы докладов и сессий вопросов-ответов) включены в книгу History of Programming Languages II, которая была выпущена в феврале 1996 года.

Если рассматриваемая тема вам интересна – то рекомендую также ознакомиться с книгой Брайана Кернигана «Время UNIX. A History and a Memoir».

В переводе активно используются гиперссылки. Для общеизвестных (в рамках темы статьи) сущностей и персоналий – в частности, ОС Unix и Кена Томпсона – гиперссылка используется только при первом упоминании в тексте. В остальных случаях гиперссылка используется при первом упоминании на странице.

Разработка языка программирования С

Деннис М. Ритчи

AT&T Bell Laboratories
Murray Hill, NJ 07974 USA

Аннотация

Язык программирования С был разработан в начале 1970-х годов в качестве инструмента реализации зарождающейся в то время операционной системы Unix и развил идеи языка BCPL, добавив в него систему типов. Он создавался на ЭВМ со скромнейшими аппаратными ресурсами для улучшения скудного набора инструментов программистов и в итоге к настоящему времени стал одним из доминирующих языков программирования. В данной статье рассматривается история создания и развития этого языка.

Вступление

Эта статья посвящена развитию языка программирования С – в том числе тому, что на него повлияло, и условиям, в которых он был создан. Для экономии места я опускаю полное описание самого языка С, его родителя – языка B [Johnson 1973] и его прародителя – языка BCPL [Richards 1979], и вместо этого концентрируюсь на характерных элементах каждого языка и том, как они развивались.

Язык С появился в 1969–1973 годах, на ранних стадиях разработки операционной системы Unix; наиболее продуктивным в плане развития языка был 1972 год. Ещё одна волна изменений достигла пика между 1977 и 1979 годами, когда была продемонстрирована переносимость ОС Unix. В середине этого второго периода появилось первое широко доступное описание языка: книга «Язык программирования С», часто называемая «белой книгой» и «K&R» [Kernighan 1978]. Наконец, в середине 1980-х годов С был официально стандартизирован комитетом ANSI X3J11, который взял на себя ответственность за внесение в язык дальнейших изменений. До начала 1980-х годов язык ассоциировался практически лишь с одной Unix, хотя уже тогда существовали его компиляторы для различных машинных архитектур и операционных систем. В последнее время область его применения существенно расширилась, и сегодня он входит в число наиболее часто используемых языков компьютерной индустрии.

История: место действия

Для Bell Telephone Laboratories конец 1960-х был бурной эпохой исследования компьютерных систем [Ritchie 1978, 1984]. Компания вышла из проекта Multics [Organick 1975], который начинался как совместная разработка Массачусетского технологического института, General Electric и Bell Labs; к 1969 году руководство Bell Labs (и даже сами учёные, участвующие в проекте) пришло к выводу, что достижение целей, сформулированных перед началом работ по Multics, потребует слишком большого количества времени и денег. Ещё до того, как ЭВМ GE-645, на который разрабатывалась Multics, была вывезена из офиса, неофициальная рабочая группа, возглавляемая главным образом Кеном Томпсоном, начала проработку альтернативной операционной системы.

Томпсон хотел создать комфортную вычислительную среду, которая будет соответствовать его видению, используя для этого любые доступные средства. Оглядываясь назад, становится очевидным, что в его планы входила реализация многих инновационных идей из Multics – в том числе явное представление процессов как обособленных объектов управления, файловой системы с древовидной иерархической структурой, интерпретатора команд (командной оболочки), являющегося программой пользовательского уровня, простое представление текстовых файлов и общий доступ к устройствам. Некоторые идеи Multics были отброшены – например, унифицированный доступ к памяти и файлам. Стоит сказать, что с самого начала и Томпсон, и я, и остальные участники рабочей группы вдохновлялись ещё одной новаторской концепцией (хотя впервые она была применена не в Multics) – написанием операционной системы на языке высокого уровня. Язык PL/I, на котором разрабатывалась в Multics, нам не особо нравился, и мы использовали другие языки, включая BCPL, одновременно с этим сожалея о потере преимуществ создания программ на более высокоуровневых (по сравнению с языком ассемблера) – таких, как простота написания и лёгкость понимания получившегося кода. В то время мы не придавили большого значения вопросам переносимости; интерес к этой теме возник позже.

Томпсон столкнулся с по-спартански ограниченными даже для тех времён аппаратными ресурсами: ЭВМ DEC PDP-7, на которой он начал работу в 1968 году, обладала памятью размером в 8 000 машинных слов (размер слова – 18 бит) и не включала никаких полезных для задач Томпсона утилит. Для удовлетворения своего желания – использовать язык высокого уровня – ему пришлось написать исходную версию Unix на языке ассемблера PDP-7. Вначале он даже не программировал на самом PDP-7, а вместо этого использовал набор макросов для ассемблера GEMAP на машине GE-635. Постпроцессор выдавал результат на перфоленту, которую мог «прочитать» PDP-7.

Код с перфолент загружался с GE-635 на PDP-7 и тестировался. Это повторялось до тех пор, пока не были созданы примитивное ядро Unix, текстовый редактор, ассемблер, простая командная оболочка-интерпретатор и несколько утилит (таких, как rm, cat и cp). После этого операционная система стала самодостаточной – программы можно было писать и тестировать, не прибегая к использованию перфоленты, и разработка продолжилась на самом PDP-7.

Ассемблер PDP-7 Томпсона превзошёл по простоте даже ассемблер DEC; он лишь оценивал выражения и выдавал соответствующие биты. Не было ни библиотек, на загрузчика, ни линковщика – весь исходный код программы передавался ассемблеру, который формировал исполняемый файл (отсюда и появилось название файла a.out; это результат работы ассемблера. Даже после того,

как был разработан линковщик и появилась возможность указания произвольного имени для файла, создаваемого в результате компиляции, по умолчанию использовалось именно **a.out**).

В 1969 году, вскоре после того, как Unix запустили на PDP-7, Дуг Макилрой создал для этой системы первый язык высокого уровня: реализацию TMG Макклюра [McClure 1965]. TMG (TransMoGrifier) – язык для создания компиляторов, основанный на методе рекурсивного спуска и сочетающий в себе контекстно-свободную синтаксическую нотацию с процедурными элементами. Макилрой и Боб Моррис использовали TMG для написания ранней версии компилятора языка PL/I для Multics.

Вдохновившись подвигом Макилроя в реализации TMG, Томпсон воспринял это как вызов и решил, что Unix – возможно, тогда у неё ещё даже не было названия – нужен язык системного программирования. После быстро провалившейся попытки освоить FORTRAN он решил вместо этого создать собственный язык, который назвал B. Этот язык можно рассматривать как C без типов; точнее, это вариант языка BCPL, втиснутый в ограничения на 8 000 слов памяти PDP-7 и профильтрованный мозгом Томпсона. Его название (B), скорее всего, представляет собой сокращение от BCPL, хотя альтернативная версия утверждает, что это сокращение от Bon [Thompson 1969] – совсем не похожего на B языка программирования, созданного Томпсоном во времена работы над Multics. Язык Bon, в свою очередь, Томпсон назвал либо в честь своей жены Бонни, либо (согласно энциклопедической цитате из руководства по языку) в честь религии, ритуалы которой включают в себя бормотание «слов силы».

Истоки: языки

Язык BCPL был разработан Мартином Ричардсом в середине 1960-х во время его пребывания в Массачусетском технологическом институте, и в начале 1970-х годов использовался в нескольких интересных проектах, среди которых операционная система OS6, созданная в Оксфорде [Stoy 1972] и программное обеспечение компьютера Alto, созданного в Xerox PARC [Thacker, 1979]. Мы были знакомы с этим языком, потому что для разработки Multics использовалась система MIT CTSS [Corbato 1962], над которой работал Ричардс. Исходный компилятор BCPL был перенесён как в Multics, так и в систему GE-635 GECOS Раддом Канадаем при участии других сотрудников из Bell Labs [Canaday 1969]. Во время последнего (близкого к агонии) этапа работы над Multics в Bell Labs этот язык стал любимым для группы людей, которая позже начала заниматься Unix.

BCPL, B и C являются неотъемлемой частью семейства традиционных процедурных языков, к типичным примерам которых можно отнести FORTRAN и ALGOL 60. Они ориентированы на системное программирование, имеют лаконичный синтаксис и могут транслироваться в машинный код с помощью простых компиляторов. Они «близки к железу» в том смысле, что имеющиеся в них абстракции основаны на типах данных и операциях, предоставляемых обычными (не специфическими) процессорами, и используют библиотечные подпрограммы для организации ввода/вывода и другого взаимодействия с операционной системой. С меньшим успехом они используют библиотечные процедуры для создания интересных управляющих конструкций, таких как сопрограммы и замыкания процедур. В то же время абстракции этих языков относятся к

достаточно высокому уровню, что при осторожном подходе позволяет достичь переносимости кода между различными платформами.

BCPL, B и C имеют множество синтаксических отличий, но в целом они схожи. Программы состоят из последовательности глобальных объявлений и объявлений функций (процедур). В BCPL процедуры могут быть вложенными, но в коде процедуры нельзя использовать нестатические объекты, определённые в её вложенных процедурах. Языки B и C избегают такого ограничения, вводя более строгое: в них вложенные процедуры отсутствуют в принципе. Каждый из языков (за исключением самых ранних версий B) поддерживает раздельную компиляцию и предоставляет средства для подключения к файлу исходного кода текста из именованных файлов.

Некоторые синтаксические и лексические механизмы BCPL более элегантны и формализованы по сравнению с B и C. Например, объявления процедур и данных в BCPL имеют более единообразную структуру и предоставляют более полный набор операторов цикла. Хотя программы на языке BCPL теоретически представляют собой поток символов, в котором нет «разделителей», есть разумная возможность в большинстве случаев опускать точки с запятой после инструкций, расположенных в конце строк; в языках B и C такого удобства нет, и инструкции практически всегда должны заканчиваться точкой с запятой. Несмотря на различия языков, существенная часть инструкций и операторов BCPL имеет точные аналоги в B и C.

Некоторые структурные различия между BCPL и B возникли из-за ограничений промежуточной памяти. Например, объявления в BCPL могут иметь следующую форму:

```
let P1 be command
and P2 be command
and P3 be command
...
```

где текст программ, представленных в виде *команд*, включает в себя целые процедуры. Такие объявления связаны (за счёт оператора **and**) и компилируются «одновременно», поэтому имя P3 известно внутри процедуры P1. Аналогичным образом BCPL позволяет «упаковать» группу объявлений и инструкций в выражение, которое возвращает значение, например[1]:

```
E1 := valof $( declarations ; commands ; resultis E2 $) + 1
```

Компилятор BCPL легко обрабатывал такие конструкции, сохраняя и анализируя распарсенное представление всей программы в памяти. Ограниченный объём памяти компьютеров требовал от компилятора быть однопроходным и как можно быстрее формировать исполняемый файл. Доработки синтаксиса, сделавшие это возможным, были в дальнейшем перенесены в C.

[1] Из спецификации языка BCPL: «valof – это оператор результата выполнения блока (формы записи выражения). Вычисление выражения производится путём выполнения блока до тех пор, пока не встретится оператор resultis, что приводит к остановке выполнения блока и возврату значения, заданного в операторе resultis» (прим. переводчика)

Некоторые менее приятные особенности BCPL возникли из-за его собственных технологических проблем, и их сознательно избегали при разработке языка B. Например, BCPL использует механизм «глобального вектора» для передачи данных между раздельно скомпилированными программами. При таком подходе программист явно связывает имя каждой видимой «извне» процедуры и переменной с числовым смещением в глобальном векторе; эти смещения используются на этапе линковки. В первых версиях языка B этого неудобства избежали за счёт добавления требования передавать компилятору сразу весь текст программы. В более поздних реализациях B и всех реализациях C обработкой имён «внешних» (external) объектов, встречающихся в раздельно скомпилированных файлах, занимается обычный линковщик, снимая с программиста бремя назначения смещений.

Другие изменения, произведённые при переходе от BCPL к B, являлись «вкусовщиной», и некоторые из них были довольно спорными – например, решение использовать для оператора присваивания одиночный символ «=» вместо «:=». В языке B комментарии обрамлялись символами /* … */, а в BCPL используются однострочные комментарии, начинающиеся с символов //. Наследие PL/I здесь очевидно (в C++ были «воскрешены» символы комментариев из BCPL). На синтаксис объявлений повлиял FORTRAN: в языке B объявления начинаются со спецификатора (такого, как auto или static), за которым следует список имён объектов, а язык C не только сохранил этот стиль, но и развил его, добавив требование указания типа перед названиями объектов.

Не все различия между языком BCPL, описанным в книге Ричардса [Richards 1979] и языком B, были преднамеренными; мы начинали с более ранней версии BCPL [Richards 1967]. Например, оператор **switchcon** ещё не заканчивался ключевым словом **endcase**, и поэтому использование ключевого слова **break** для «выхода» из оператора **switch** в языках B и C не было сознательным изменением по сравнению с BCPL – просто языки развивались разными путями.

В отличие от повсеместных изменений синтаксиса, которые произошли во время создания B, основное семантическое содержание BCPL – его структура типов и правила вычисления выражений — осталось нетронутым. Оба языка являются бестиповыми или, если точнее, имеют один тип данных, называемым «словом» или «ячейкой» (набор бит фиксированной длины). Память в этих языках представлена линейным массивом таких «ячеек», а то, как интерпретируется их содержимое, зависит от выполняемой операции. Например, оператор «+» просто вычисляет сумму своих операндов, используя машинную инструкцию сложения целых чисел; другие арифметические операции тоже «не осознают» фактического содержания своих операндов. Поскольку память представляет собой линейный массив – значение в ячейке можно интерпретировать как индекс этого массива, и для этой цели BCPL предоставляет специальный оператор. Сначала он обозначался как **rv**, а позже как **!**; в языке B для этой же цели используется унарный оператор *****. То есть если **p** – это ячейка, содержащая индекс (или содержащая адрес, или являющаяся указателем) другой ячейки, то ***p** относится к содержимому ячейки **p** и может быть использовано как значение в выражении или место, в которое будет что-то записано с помощью оператора присваивания.

Поскольку указатели в BCPL и B — это просто целочисленные индексы массива, размещённого в памяти, то арифметические операции над ними имеют смысл: если **p** – это адрес ячейки, то **p+1** – адрес следующей ячейки. Это соглашение является основой семантики операций с массивами для обоих языков

Когда на BCPL пишут

```
let V = vec 10
```

или на B пишут

```
auto V[10];
```

то эффект один и тот же: выделяется память под ячейку с именем **V**, затем выделяется память под 10 последовательно размещённых ячеек, и адрес (индекс) памяти первой из них помещается в **V**.

По общему правилу языка B, в выражении

```
*(V+i)
```

выполняется суммирование переменных **V** и **i**, и осуществляется доступ к ячейке с соответствующим индексом. И в BCPL, и в B добавлены специальные обозначения для облегчения такого доступа к массивам. В языке B его эквивалентом является выражение:

```
V[i]
```

а в BCPL:

```
V!i
```

Такой подход к индексации массивов был необычен даже для того времени. Позднее он стал частью языка C, в котором принял ещё более нетрадиционную форму.

Ни в BCPL, ни в B, ни в C нет встроенной поддержки строковых типов данных; в каждом из языков строки рассматриваются как массивы целых чисел, правила обработки которых дополнены несколькими соглашениями. В BCPL и B строковые литералы содержат адрес статической области памяти, ячейки которой инициализированы символами строки. В BCPL в первом байте строки записывалось число её символов; в языке B такого счётчика нет, и строка завершается специальным символом-терминатором ('*e'). Это изменение было сделано намеренно: во-первых, чтобы избежать ограничения на длину строки, связанное с хранением числа её символов в одном байте (который состоял из 8 или 9 бит), а во-вторых – по нашему мнению, использование счётчика символов казалось менее удобным, чем использование терминатора.

Обработка отдельных символов строки в BCPL обычно выполнялась путём копирования содержимого строки в другой массив (по одному символу в ячейку) с последующей повторной «упаковкой». В языке B были поддержаны соответствующие подпрограммы, но люди чаще использовали функции из дополнительных библиотек, которые позволяли обратиться к отдельным символам строки или заменить их.

Продолжение истории

После появления компилятора B, написанного на языке TMG, Томпсон переписал B на самом себе (используя метод раскрутки). Во время разработки языка он постоянно сражался с ограниченным объёмом памяти своей ЭВМ: добавление в язык очередной новой возможности раздувало компилятор до такой степени, что он едва помещался в память, но в результате каждого переписывания размер уменьшался. Например, в B появились обобщённые операторы присваивания: выражение «x=+y» приводило к записи в переменную **x** суммы **x** и **y**. Такая нотация пришла из ALGOL 68 [Wijngaarden 1975]; мы узнали о ней от Макилроя, который добавил её в свою версию TMG (в языке B и ранних версиях C этот оператор записывался как =+ вместо +=; это неудачное решение, исправленное в 1976 году, было вызвано соблазнительно простым способом обработки первого варианта записи в лексическом анализаторе языка B).

Томпсон пошёл ещё дальше, изобретя операторы ++ и --, которые, соответственно, выполняют инкремент и декремент значения операнда; их расположение по отношению к операнду (перед или после) определяет, будет ли соответствующая операция выполнена до или после записи значения в результирующую переменную. Этих операторов не было в самых ранних версиях языка B; они появились в процессе его развития. Люди часто предполагают, что они были созданы для использования режимов автоинкремента и автодекремента адресов, которые поддерживал DEC PDP-11 (это была первая ЭВМ, на которой язык C и ОС Unix обрели популярность). Но это невозможно по историческим причинам – на момент разработки языка B у нас ещё не было PDP-11. Однако у используемого нами PDP-7 было несколько ячеек памяти с «автоинкрементом»; косвенное обращение к этим ячейкам увеличивало значение в них на единицу. Вероятно, эта возможность и подсказала Томпсону идею для его операторов; решение сделать их одновременно префиксными и постфиксными было его собственной находкой. Конечно, он не использовал эти ячейки для непосредственной реализации операторов, и более сильным стимулом для их добавления, скорее всего, было его наблюдение о том, что результат трансляции является более компактным, чем для «x=x+1».

Компилятор B на PDP-7 генерировал не машинные инструкции, а «шитый код» [Bell 1972] – последовательность адресов фрагментов кода, выполняющих элементарные операции. Такие операции в большинстве случаев – что особенно характерно для языка B – выполняются на простой стековой машине.

В версии Unix для PDP-7 на языке B было написано всего несколько вещей (и ещё компилятор самого B), потому что сама ЭВМ обладала такими скромными аппаратными ресурсами, что годилась только для экспериментов; полностью переписать операционную систему и утилиты на B было слишком затратным решением, чтобы браться за его осуществление. В какой-то момент Томпсон облегчил работу с адресным пространством, разработав компилятор «виртуального B», который позволял интерпретируемой программе занимать более 8 000 байт памяти за счёт постепенной «подкачки» кода и данных внутри интерпретатора, но был слишком медленным для практического использования при написании большинства основных утилит. Тем не менее, на B всё же были написаны некоторые утилиты, включая раннюю версию калькулятора с произвольной точностью dc, знакомого пользователям Unix [McIlroy 1979]. Самым амбициозным проектом, который я затеял, было создание настоящего кросс-компилятора, который транслировал B в машинные инструкции GE-635, а не в «шитый код». Это был небольшой подвиг: полноценный

компилятор языка B, написанный на нём самом же, и генерирующий код для 36-битного мэйнфрейма, причём запускался этот компилятор на 18-битной машине с 4 000 машинных слов пользовательского адресного пространства. Этот проект стал возможен только благодаря простоте языка B и его системе исполнения (runtime).

Хотя время от времени у нас возникали мысли о поддержке в Unix одного из основных языков того времени, таких как FORTRAN, PL/I или ALGOL 68, подобный проект казался безнадёжно большим для наших ограниченных ресурсов: требовались гораздо более простые и компактные инструменты. Все эти языки повлияли на нашу работу, но веселее было создать свой собственный.

К 1970 году проект Unix показал себя достаточно многообещающим, и мы смогли приобрести новую ЭВМ – DEC PDP-11. Процессор был одним из первых в этой линейке, выпущенной DEC, и ещё три месяца мы ждали диск. Чтобы заставить запускаться «шитый код» программ, написанных на B, потребовалось лишь добавить фрагменты кода, реализующие работу операторов, и простой ассемблер, который я тоже написал на B. Первой интересной программой, запущенной и протестированной на нашей PDP-11, был калькулятор dc – никакой операционной системы в тот момент у нас ещё не было. Пока диск был в пути, Томпсон быстро переписал ядро Unix и некоторые основные команды на языке ассемблера PDP-11. Из доступных 24 Кб памяти самая ранняя версия Unix использовала 12 Кб для операционной системы; крошечный сегмент выделялся для пользовательских программ, а всё остальное использовалось как RAM. Эта версия предназначалась только для тестирования, а не для реальной работы; мы использовали машинное время для решения задач о ходе коня на шахматных досках разного размера. Как только приехал диск – мы быстро перешли на него, переписав команды ассемблера на языке ассемблера PDP-11 и перенеся то, что уже было написано на B.

К 1971 году у нашего миниатюрного компьютерного центра начали появляться пользователи. Все мы хотели упростить создание интересного программного обеспечения. Программирование на языке ассемблера было настолько утомительным, что для B, несмотря на его проблемы с производительностью, была создана небольшая библиотека полезных сервисных функций, и он использовался для создания всё новых и новых программ. Одним из наиболее заметных достижений этого периода являлась первая версия генератора синтаксических анализаторов yacc, разработанная Стивом Джонсоном [Johnson 1979a].

Проблемы языка B

Адресация памяти ЭВМ, на которых мы использовали BCPL, а потом B, основывалась на «словах», и единственный тип данных этих языков – «ячейка» – соответствовал аппаратному машинному слову. Появление PDP-11 выявило некоторые недостатки семантической модели языка B. Во-первых, его механизмы обработки символов, унаследованные (с небольшими изменениями) от BCPL, были довольно неуклюжими: использование библиотечных процедур для распределения символов строк по отдельным ячейкам с их последующей «упаковкой» для доступа к отдельным символам или их замене стало казаться неудобным и даже глупым на ЭВМ с побайтовой адресацией памяти.

Во-вторых, хотя изначально PDP-11 не поддерживала арифметические операции с плавающей точкой, производитель обещал, что эта возможность скоро будет реализована. В наших компиляторах BCPL для Multics и GECOS она была поддержана с помощью добавления специальных операторов; механизм их реализации базировался на том, что на используемых нами ЭВМ размер машинного слова был достаточен, чтобы хранить в нём число с плавающей точкой. На 16-битной PDP-11 это бы не сработало

Наконец, семантические модели B и BCPL подразумевали накладные расходы при работе с указателями: по правилам этих языков указатель определялся как индекс массива машинных слов. Каждое обращение по указателю генерировало вызываемое на этапе выполнения преобразование его значения в байтовый адрес, ожидаемый аппаратным обеспечением.

Все перечисленные моменты наталкивали на мысль, что необходима система типизации, которая поможет справиться с адресацией символов и байтов, а также подготовиться к появлению аппаратной поддержки чисел с плавающей точкой. Другие связанные с ней аспекты – в частности, типобезопасность и проверка интерфейса – тогда не казались такими важным, какими они станут в будущем.

Помимо проблем с самим языком, метод «шитого кода» компилятора B создавал на выходе настолько медленные (по сравнению с их аналогами на языке ассемблера) программы, что мы отказались от идеи переписать операционную систему и её основные утилиты на B.

В 1971 году я начал расширять язык B, добавив в него символьный тип данных, а также переписал компилятор таким образом, чтобы он генерировал машинные инструкции PDP-11 вместо «шитого кода». Таким образом, переход от B к C совпал с созданием компилятора, способного создавать быстрые и достаточно компактные программы, чтобы конкурировать с языком ассемблера. Свою слегка расширенную версию языка B я назвал NB, что означает «new B».

Язык C: эмбриональный период

Язык NB просуществовал так недолго, что я даже не создал его полного описания. В нём были типы **int** и **char**, а также массивы и указатели. Объявление переменных производилось в таком стиле:

```
int i, j;
char c, d;
int iarray[10];
int ipointer[];
char carray[10];
char cpointer[];
```

Семантика массивов осталась точно такой же, какой была в B и BCPL: объявление **iarray** (и **carray**) создает ячейку, динамически инициализируемую значением, являющимся начальным адресом последовательности из 10 целых чисел (для **iarray**) и символов (для **carray**) соответственно. В объявлениях **ipointer** и **cpointer** размерность массива опущена, чтобы указать компилятору, что в данном случае не нужно производить автоматическе выделение памяти. Внутри процедур интерпретация указателей в NB была идентична интерпретации переменных массива: объявление указателя создавало ячейку, отличающуюся от ячейки массива только тем, что программист должен был сам записать в неё адрес памяти вместо того, чтобы позволить компилятору выделить сегмент памяти и инициализировать ячейку значением его начального адреса.

В ячейках, связанных с именами массивов и указателей, хранились машинные адреса (байтовые смещения) соответствующих областей памяти. Таким образом, обращение по указателю не требовало дополнительных накладных расходов на этапе выполнения для пересчёта значения указателя из слов в байты. С другой стороны, машинный код для индексирования массивов и арифметических операций с указателями теперь зависел от типа массива или указателя: в процессе вычисления выражений «iarray[i]» и «ipointer+i**»** при доступе к памяти значение **i** умножалось на размер базового типа массива/указателя, выраженный в байтах.

Эта семантика позволила организовать простой переход от языка B к NB, и я экспериментировал с ней несколько месяцев. Её проблемы стали очевидны, когда я попытался расширить число типов данных, поддерживаемых языком – в частности, добавить структуры (записи). Структуры должны были размещаться в памяти интуитивно понятным образом, но в структуре, содержащей массив, не было ни подходящего места для размещения указателя, содержащего начальный адрес массива, ни удобного способа инициализировать его. Например, записи каталогов[2] в ранних версиях Unix описывалась так:

```
struct {
   int inumber;
   char name[14];
};
```

[2] Запись каталога (directory entry) содержит имя объекта (файла или каталога) и набор его атрибутов (прим. переводчика)

Я хотел, чтобы структура не просто описывала абстрактный объект, но и позволяла работать с его данными. Где компилятор мог спрятать указатель на имя, которого требовала семантика? Даже если воспринимать структуры более абстрактно и каким-то образом скрыть область памяти, в которой будет храниться значение указателя, то как бы я мог реализовать корректную инициализацию этих указателей при создании сложных объектов – например, структур с полями-массивами других структур, содержащих другие массивы и т. д. – с произвольным уровнем вложенности?

Решение этой технической проблемы обеспечило ключевой скачок в эволюционном переходе от бестипового BCPL к типизированному C. Оно заключалось в отказе от идеи записывать значение указателя начального адреса массива в какое-то «хранилище»; вместо этого создание указателя происходило в тот момент, когда имя массива использовалось в каком-либо выражении. Правило, сохранившееся в языке C и поныне, заключается в том, что переменная типа «массив» при её использовании в выражении на этапе компиляции преобразуется в начальный адрес области памяти, в которой расположены данные этого массива.

Это изобретение позволило сохранить работоспособность большей части существовавшего кода (написанного на «чистом» B), несмотря на существенное изменение в семантике языка. Те немногие программы, в которых в переменную с именем массива записывалось новое значение адреса (такое было возможно в B и BCPL, но бессмысленно в C) для наполнения массива новыми данными, легко было исправить. Что ещё важнее – новый язык сохранил согласованную и работоспособную (хотя и необычную) семантику массивов, одновременно открывая путь к более полному набору типов данных.

Этот набор типов – второе нововведение, которое наиболее явно отличает язык C от его предшественников; особенно ярко оно проявляется в синтаксисе объявления переменных. Язык NB поддерживал типы **int** и **char**, а также массивы этих типов и указатели на них, но не предлагал никаких дополнительных способов композиции для создания составных типов данных. Нужно было обобщение: для объекта произвольного типа требовалась возможность объявить массив таких объектов, вернуть такой объект в результате вызова функции и получить указатель на него.

Для каждого описанного случая уже существовал способ обращения к объекту «базового» типа – доступ к массиву по индексу, вызов функции и разыменование указателя соответственно. Путем проведения аналогий был создан синтаксис объявления объектов, отражающий синтаксис выражений, в которых обычно эти объекты используются. Таким образом

```
int i, *pi, **ppi;
```

представляют собой переменную типа **int**, указатель на **int** и указатель на указатель на **int** соответственно. Синтаксис этих объявления отражает тот факт, что i, *pi и **ppi в конечном итоге используются в выражениях для работы со значением типа **int**.

Аналогично

```
int f(), *f(), (*f)();
```

являются объявлениями функции, возвращающей **int**, функции, возвращающей указатель на **int** и указателя на функцию, которая возвращает **int**. Если говорить о

```
int *api[10], (*pai)[10];
```

то это объявление массива указателей на **int** и указателя на массив типа **int**. Во всех приведённых выше примерах объявление переменной похоже на её использование в выражении при работе с типом, указанным в начале объявления.

Схема композиции типов, использованная в языке C, во многом обязана ALGOL 68, хотя, возможно, приверженцы ALGOL не одобрили бы то, какой она получилась. Ключевым элементом, который я позаимствовал из ALGOL, была система типов, основанная на элементарных типах (к которым относились структуры), на базе которых можно создавать массивы, указатели (ссылки) и функции (процедуры). Концепции объединений (union) и приведения типов данных (type cast) из ALGOL 68 в будущем также оказали влияние на язык C.

После создания системы типов, соответствующих правок синтаксиса и разработки компилятора для нового языка я почувствовал, что он заслуживает нового имени; мне казалось, что «NB» не выражает весь объём изменений, внесённых мной в B. Я решил следовать «однобуквенному» стилю и назвал его C, оставив открытым вопрос, является ли это следующей после B буквой алфавита или следующим символом в аббревиатуре BCPL.

Язык C: неонатальный период

Язык продолжал активно развиваться, и вскоре после того, как я дал ему имя, появились операторы && и ||. В языках BCPL и B вычисление выражений зависит от контекста: внутри **if** и других условных операторов, которые сравнивают значение выражения с нулем, операторы **and** (&) и **or** (|) интерпретируются особым образом. В обычном контексте они являются побитовыми, но в языке B при вычислении выражения

```
if (e1 & e2) ...
```

компилятор проверяет значение переменной **e1**, и если оно не равно нулю – то проверяет значение переменной **e2**, а если и оно не равно нулю – вычисляет выражение внутри блока **if**. Этот подход рекурсивно применяется ко всем остальным операторам & и | в условии оператора **if**. Семантика «вычислений по короткой схеме» в таком контексте казалась весьма желательной, но перегрузка операторов & и | вызывала непонимание у пользователей. По предложению Алана

Снайдера я добавил отдельные операторы && и ||, чтобы сделать механизм использования логических операторов более понятным.

Их запоздалое введение объясняет неудачность правил приоритета вычисления операторов языка C. В языке B пишут

```
if (a==b & c) ...
```

чтобы проверить, что значения переменных **a** и **b** равны друг другу, и при этом значение переменной **c** не равно нулю; в таком выражении нужно, чтобы оператор & имел более низкий приоритет, чем ==. При переносе кода из B в C нужно заменить & на &&; чтобы сделать переход на новый язык менее болезненным, мы решили оставить у операторов & и == те приоритеты, которые у них были, а оператору && назначить приоритет чуть ниже. Сегодня уже кажется, что было бы предпочтительнее изменить приоритеты операторов & и ==, тем самым упростив распространённую идиому языка C: чтобы сравнить набор конкретных бит битовой маски с известным значением, нужно написать

```
if ((a & mask) == b) ...
```

где вложенные скобки обязательны для корректного вычисления выражения, но легко забыть их поставить.

В 1972-1973 годах в языке произошли и многие другие изменения, но наиболее важным было введение препроцессора – частично по настоянию Алана Снайдера [Snyder 1974], но также и в знак признания полезности механизма подключения файлов, доступного в BCPL и PL/I. Его первоначальная версия была чрезвычайно простой и содержала только подключение файлов (#include) и простые замены строк для макросов без параметров (#define). Вскоре после этого препроцессор был расширен – в основном Майком Леском, а затем Джоном Райзером – для включения макросов с аргументами и директив условной компиляции. Изначально препроцессор считался необязательным дополнением к самому языку. Более того, в течение нескольких лет он даже не вызывался, если исходная программа не содержала в начале специального сигнала. Такое отношение сохранилось и в настоящее время; это объясняет как неполную согласованность синтаксиса препроцессора с остальными элементами языка, так и неточность его описания в ранних справочных руководствах.

Переносимость

К началу 1973 года основа современной версии C была готова. Язык и компилятор были достаточно мощными, чтобы позволить нам тем же летом переписать с их помощью ядро Unix для PDP-11 (в 1972 году Томпсон предпринял попытку написать систему на ранней версии C – когда в нём ещё не было структур – но быстро отказался от этой идеи). Также в этот период компилятор был перенесён на другие имеющиеся у нас ЭВМ – в частности, на Honeywell 635 и IBM 360/370. Поскольку язык не мог существовать в «вакууме» – были разработаны прототипы современных

библиотек. В частности, Леск написал «переносимый пакет ввода-вывода» [Lesk 1972], который позже был переработан и стал библиотекой «стандартного ввода-вывода» языка C. В 1978 году мы с Брайаном Керниганом опубликовали книгу «Язык программирования Си» (также часто называемую «K&R»). Хотя в ней не описывались некоторые дополнения к языку, которые вскоре стали общепринятыми, эта книга служила справочником по C в течение более 10 лет – до тех пор, пока не был разработан и принят его официальный стандарт. В процессе её создания мы с Керниганом плотно общались друг с другом, но при этом существовало чёткое разделение обязанностей: Керниган написал почти всю основную часть, а я отвечал за приложение, содержащее справочное руководство, и главу о взаимодействии с операционной системой Unix.

С 1973 по 1980 год в язык были добавлены спецификаторы типов **unsigned** и **long**, объединения (union) и перечисления, а структуры стали объектами первого класса (за исключением возможности присваивания в коде экземплярам структур «литеральных» значений). Не менее важные изменения произошли в рабочем окружении и связанным с ним технологиях. Написание ядра Unix на C вселило в нас достаточную уверенность в полезности и эффективности языка, и мы начали переписывать на нём системные утилиты и инструменты, а затем переносить наиболее интересные из них на другие платформы. Как описано в статье «Переносимость программ C и системы UNIX» [Johnson 1978a] – мы обнаружили, что самые сложные проблемы портирования инструментов Unix проявлялись не при поддержке языка C на новом оборудовании, а при переписывании на нём уже существующего программного обеспечения. Поэтому Стив Джонсон начал работать над pcc – компилятором C, который должен был легко переноситься на новые ЭВМ [Johnson 1978b]. Одновременно с этим он, Томпсон и я начали переносить саму Unix на ЭВМ Interdata 8/32.

Изменения в языке, сделанные в этот период (особенно в 1977 году), в основном были связаны с вопросами переносимости и типобезопасности. Они должны были помочь справиться с проблемами, которые мы предвидели и наблюдали при переносе значительного количества существующего кода на новую для нас платформу Interdata. В то время язык C всё ещё демонстрировал явные признаки своего бестипового происхождения. Например, в ранних руководствах и уже написанном коде работа с указателями практически не отличалась от работы с целочисленными переменными, хранящими адреса памяти. Видя сходство арифметических свойств указателей и целых беззнаковых чисел, мы не могли устоять перед искушением идентифицировать их отдельно. Беззнаковые (unsigned) типы данных были добавлены для того, чтобы отличать манипуляции с указателями от математических операций с беззнаковыми целыми. Также B, BCPL и ранние версии C допускали присвоение указателю целочисленного значения, но эта практика не поощрялась; семантика преобразования типов (называемая выше в примере с ALGOL 68 «приведением типов») была изобретена, чтобы сделать механизм конверсии типов более явным. Вдохновлённый примером языка PL/I, ранний вариант C не требовал жёсткого соответствия между указателем на структуру и самой структуры; это позволяло программистам писать в стиле

```
pointer->member
```

без учёта типа указателя. Такое выражение не считалось ошибкой; pointer использовался для получения адреса памяти, а название поля структуры определяло смещение в памяти относительно него и тип значения.

Хотя в первом издании K&R описывалось большинство правил, которые привели систему типов данных языка C к его нынешней форме, сохранялись и многие программы, написанные в старом, более «вольном» стиле, и компиляторы, допускавшие эти «вольности». Чтобы подтолкнуть людей уделять больше внимания соблюдению стандарта языка и выявлять синтаксически корректные, но подозрительные конструкции, а также помогать обнаруживать несоответствие интерфейсов, Стив Джонсон внёс изменения в свой компилятор pcc и разработал статический анализатор lint [Johnson 1979b], который сканировал файлы приложения и выводил замечания по сомнительным конструкциям, найденным в коде.

Рост популярности

Успех нашего эксперимента по переносу Unix на Interdata 8/32 вдохновил Тома Лондона и Джона Райзера на повторение этой процедуры для DEC VAX 11/780. Эта ЭВМ стала гораздо более популярной, чем Interdata, и Unix вместе с языком C начали быстро распространяться как внутри AT&T, так и за её пределами. Хотя к середине 1970-х годов Unix использовалась в различных проектах внутри Bell System, а также в небольшой группе исследовательских лабораторий промышленных, образовательных и правительственных организаций (не относящихся к нашей компании), реальный рост её популярности начался только после того, как была обеспечена переносимость. Отдельно стоит отметить её версии System III и System V, которые своими силами разработало новое подразделение компании AT&T, называющееся Computer Systems, и BSD, выпущенную Калифорнийским университетом в Беркли и основанную на исходном коде от Bell Laboratories.

В 1980-е годы язык C получил широкое распространение в компьютерной отрасли, и его компиляторы были разработаны практически для всех архитектур и операционных систем; в частности, он стал популярным инструментом разработки ПО для персональных компьютеров как среди производителей коммерческого программного обеспечения, так и среди обычных пользователей, интересующихся программированием. В начале десятилетия почти каждый компилятор был основан на **pcc** Джонсона; к 1985 году уже существовало много компиляторов, являющихся независимыми разработками.

Стандартизация

К 1982 году стало ясно, что язык C нуждается в официальном стандарте. Самое близкое к такому стандарту – первое издание K&R – больше не отражало реальное состояние языка; например, в ней не описывались типы void и enum. В этой книге упоминались структуры, но уже после её публикации в язык была добавлена возможность присваивания им значений на этапе инициализации, передачи их в функцию (и использование в качестве типа значения, возвращаемого функцией) , а также проверка наличия полей в структуре (или объединении) при доступе к ним по указателю. Хотя компиляторы, распространяемые AT&T, поддерживали эти нововведения, и большинство разработчиков компиляторов, не основанных на **pcc**, быстро их реализовали, не существовало полного и «официально-авторитетного» описания языка.

Также первое издание K&R не было достаточно точным в описании многих деталей, и становилось всё более непрактично рассматривать pcc как «эталонный компилятор»; он не полностью соответствовал даже K&R, не говоря уже о последующих расширениях языка. Наконец, язык C постепенно начинал использоваться в коммерческих и государственных проектах, для которых было важного наличие официально одобренного стандарта. По этим причинам (и по настоянию Макилроя) летом 1983 года ANSI учредил комитет X3J11 под руководством CBEMA[3] с целью разработки стандарта языка C. В конце 1989 года X3J11 подготовил свой отчёт [ANSI 1989], который впоследствии был принят как ISO/IEC 9899-1990.

С самого начала своей деятельности комитет X3J11 придерживался осторожного и консервативного взгляда на расширения языка. К моему большому удовлетворению, они серьёзно отнеслись к своей цели: «*разработать ясный, последовательный и недвусмысленный стандарт для языка программирования C, который систематизирует общее существующее определение и поспособствует переносимости пользовательских программ в среде языка C*» [ANSI 1989]. Комитет осознавал, что простая публикация стандарта не изменит мир.

X3J11 внёс в сам язык только одно действительно важное изменение: он включил типы формальных аргументов в сигнатуру функции, используя синтаксис, заимствованный из C++ [Stroustrup 1986]. В старом стиле внешние (external) функции объявлялись так:

```
double sin();
```

т. е. указывался лишь тип значения, возвращаемого функцией. В новом стиле подход был улучшен:

```
double sin(double);
```

т. е. типы аргументов указывались явно, что способствовало лучшей проверке типов и соответствующих преобразований. Даже это дополнение, которое являлось заметным улучшением языка, вызвало трудности. Комитет обоснованно посчитал, что просто объявить «вне закона» определения и объявления функций «в старом стиле» невозможно, но также согласился, что новый подход лучше. Неизбежный компромисс между этими двумя позициями был лучшим из возможных решений, но стандарт языка усложнился (потому что были разрешены оба варианта объявления), и авторам кроссплатформенного ПО приходилось бороться с компиляторами, ещё не соответствующими ему.

Комитет X3J11 также внёс множество мелких дополнений и корректировок – например, квалификаторы типов const и volatile, а также немного изменил правила неявного приведения типов. Тем не менее, процесс стандартизации не изменил «характер» языка. В частности, стандарт не пытался создать формализованную спецификацию семантики языка, поэтому некоторые его тонкости могут являться предметом споров. Тем не менее, в нём учтены изменения, появившиеся уже после выхода первого издания K&R, и он обеспечивает достаточный уровень точности, чтобы служить основой для реализации языка.

[3] Computer and Business Equipment Manufacturers' Association

Таким образом, суть C не пострадала от процесса стандартизации, а сам текст стандарта не включал в себя ничего принципиально нового – по большей части он лишь тщательно специфицировал уже устоявшееся состояние языка. Более важные изменения произошли в языковом окружении: препроцессоре и библиотеках. Исторически препроцессор C по синтаксису не согласован с самим языком. На тот момент не существовало хорошего описания, как именно препроцессор взаимодействует с компилятором, и комитет X3J11 попытался исправить ситуацию. Получившийся результат был заметно лучше, чем пояснения из первого издания K&R. Помимо создания исчерпывающего описания, в препроцессор были включены новые операции, ранее добавленные лишь в конкретных реализациях по инициативе их авторов – например, объединение токенов.

Комитет X3J11 обоснованно предположил, что полное и подробное описание стандартной библиотеки C так же важно, как и работа над спецификацией самого языка. Сам язык не предоставляет средства ввода/вывода информации или любого другого взаимодействия с внешним миром и, таким образом, зависит в этом плане от набора стандартных функций. На момент публикации K&R язык C считался главным образом языком системного программирования для Unix; хотя мы предоставили примеры библиотечных функций, которые можно было легко перенести на другие операционные системы, связанная с Unix история происхождения языка была неверно истолкована. Таким образом, комитет X3J11 потратил значительную часть своего времени на разработку и документирование набора библиотечных функций, которые должны быть доступны во всех реализациях языка.

По правилам процесса стандартизации текущая деятельность комитета X3J11 ограничивается выдачей комментариев по вопросам, связанным с интерпретацией существующего стандарта. Однако неформальная (изначально) рабочая группа NCEG (Numerical C Extensions Group), сформированная Рексом Яшке, впоследствии была официально преобразована в подгруппу X3J11.1, и её участники продолжают рассматривать возможность добавления в язык различных нововведений. Как следует из первоначального названия рабочей группы – многие из нововведений касаются работы с числами: например, многомерные динамические массивы, поддержка вычислений с плавающей точкой по стандарту IEEE и повышение эффективности языка на процессорах с векторными или иными расширенными инструкциями. Но нововведения касаются не только числовых типов – например, разрабатывается нотация для обозначения литералов структур.

Наследники C

C и (даже B) повлияли на несколько языков, ставших их прямыми наследниками – хотя они не соперничают с многочисленным «потомством», которое оставил Pascal. Одна из ветвей этого генеалогического древа начала развиваться довольно рано. Когда Стив Джонсон в 1972 году во время своего отпуска посетил университет Ватерлоо, он взял с собой B. Этот язык там получил популярность на ЭВМ Honeywell, а позже породил языки Eh и Zed (канадский ответ на вопрос «что идёт после B?»). Когда Джонсон вернулся в Bell Labs в 1973 году, он был сбит с толку, обнаружив, что язык, семена которого он привёз в Канаду, успел за это время эволюционировать; даже его собственная программа yacc была переписана на C Аланом Снайдером.

К более поздним потомкам, испытавшим влияние уже непосредственно языка C, относятся Concurrent C [Gehani 1989], Objective C [Cox 1986], C* [Thinking 1990] и, что особенно важно, C++ [Stroustrup 1986]. Язык C также широко используется в качестве промежуточного представления (по сути, в качестве «переносимого языка ассемблера») для самых разных компиляторов – как языков, являющих прямыми потомками C (например, C++), так и для не связанных с C языков – например, Modula 3 [Nelson 1991] и Eiffel [Meyer 1988].

Критика

От других процедурных языков C отличается двумя характерными особенностями: взаимосвязью между массивами и указателями и способом, которым синтаксис объявления объектов «подстроен» под синтаксис записи выражений. Они же входят в число наиболее регулярно критикуемых особенностей языка и часто служат камнем преткновения для новичков. В обоих случаях трудности, созданные этими особенностями, усугубились случайностями («так исторически сложилось) или ошибками при проектировании языка. Наиболее важным фактором являлась снисходительность компиляторов C к ошибкам, связанным с типами данных. Как уже упоминалось выше – C эволюционировал из бестиповых языков. Первым своим пользователям C не казался каким-то совершенно новым языком с уникальными особенностями; вместо этого нам приходилось постоянно адаптировать существующие программы под изменения, появлявшиеся по мере развития языка, и учитывать при внесении этих изменений их влияние на уже существующий код (позже комитет ANSI X3J11, занимавшейся стандартизацией C, столкнулся с этой же проблемой).

Компиляторы в 1977 году (и даже намного позже) не жаловались на такие моменты, как присвоение указателям значений целочисленного типа (и наоборот) или использование объектов «не того» типа для доступа к полям структуры. Хотя описание языка, представленное в первом издании K&R, было достаточно (пусть и не полностью) последовательным в отношении правил типизации, в книге признавалось, что существующие компиляторы не обеспечивают их полное соблюдение. Более того, некоторые правила, призванные облегчить переход от ранних версий языка к более поздним, только способствовали нарастанию путаницы. Например, пустые квадратные скобки в объявлении функции

```
int f(a) int a[]; ( ... }
```

являются атавизмом – так объявлялись указатели во времена NB; поэтому в C в этом конкретном особом случае **a** интерпретируется как указатель. Подобная нотация сохранилась отчасти ради обратной совместимости, а отчасти потому, что она позволяла программистам сообщать тем, кто будет читать их код, о намерении передать в функцию **f** указатель на массив, а не единственное значение типа **int**. К сожалению, помимо декларации намерений такая запись запутывает тех, кто только начинает изучать язык.

В K&R ответственность за типы аргументов, передаваемых функции, ложилась на программиста, и существующие в то время компиляторы не проверяли соответствие типов.

Отсутствие в исходной версии языка требования включения типов аргументов в сигнатуру функции являлось существенным недостатком, для исправления которого комитету X3J11 пришлось внедрить в язык самое смелое и болезненное нововведение. Могу объяснить (хотя, возможно, это будет лишь оправданием) свой ранний дизайн языка тем, что я избегал технических проблем (особенно перекрестной проверки между отдельно компилируемыми файлами исходного кода) и неполным пониманием последствий перехода от бестипового языка к типизированному. Статический анализатор lint, упоминавшийся ранее, пытается смягчить эту проблему: помимо других своих функций он проверяет согласованность и связность всей программы, сканируя набор файлов исходного кода и сравнивая типы аргументов, используемые в вызовах функций, с типами аргументов в их определениях.

Некоторые случайные решения в синтаксисе способствовали увеличению сложности изучения языка. Оператор разыменования, который в C записывается как *, синтаксически является унарным префиксным оператором, как в BCPL и B. Он хорошо работает в простых выражениях, но в более сложных случаях для передачи своих намерений синтаксическому анализатору разработчику требуется использовать круглые скобки. Например, чтобы отличить функцию, возвращающую указатель, от вызова функции по указателю, пишут *fp() и (*pf)() соответственно. Поскольку синтаксис записи выражений «подстроен» под синтаксис объявления объектов – приведённая выше запись может использоваться и при объявлении:

```
int *fp();
int (*pf)();
```

В более витиеватых (но всё же возможных в реальной жизни) случаях ситуация становится сложнее:

```
int *(*pfp)();
```

Так обозначается указатель на функцию, которая возвращает указатель на значение типа **int**. Здесь надо отметить два момента. Во-первых (и это главное) – язык C имеет довольно широкий набор способов описания типов данных (по сравнению, скажем, с Pascal). Объявления объектов в выразительных языках, подобных C (например, в ALGOL 68), зачастую трудно понять потому, что сами объекты могут являться концептуально сложными (как в примере выше). Во-вторых, нужно учитывать детали синтаксиса. Объявления на языке C необходимо читать «наизнанку», и многим людям трудно понять такой подход [Anderson 1980]. Сетхи [Sethi 1981] заметил, что многие из вложенных объявлений и выражений стали бы проще, если бы оператор * использовался как постфиксный, а не префиксный, но к тому моменту было уже слишком поздно что-то менять.

Несмотря на описанные сложности я считаю принцип объявления объектов в C оправданным, и он меня устраивает. Это хороший способ унификации языка.

Сомнения насчёт целесообразности другой характерной черты языка C – специфического синтаксиса работы с массивами – кажутся более оправданными, но и у него тоже есть свои достоинства. Хотя связь между указателями и массивами в языке необычна, её можно выучить и понять.

Более того, язык имеет широкие возможности для поддержки важных концепций – например, одномерных массивов (векторов), размер которых может меняться в процессе выполнения программы – с помощью всего лишь нескольких правил и соглашений. В частности, строки обрабатываются по тем же принципам, что и любой другой массив, с дополнительным соглашением о том, что признаком конца строки является символ с кодом 0. Интересно сравнить этот подход с подходами двух других языков, являющихся практически «современниками» С: ALGOL 68 и Pascal. Массивы в ALGOL 68 либо имеют фиксированный размер, либо являются динамическими (flexible). Как в спецификации языка, так и в его компиляторах потребовалось предпринять значительные усилия для работы этого механизма (и не во всех компиляторах он реализован полностью). В оригинальной версии Pascal поддерживались только массивы и строки фиксированного размера, и это ограничивало возможности языка [Kernighan 1981]. Позже это было частично исправлено, хотя новая версия языка пока ещё не получила широкого распространения.

В языке С строки рассматриваются как массивы символов, заканчивающиеся «терминатором» – символом с кодом 0. За исключением одного специального правила, связанного с инициализацией строковых переменных литералами, семантика строк полностью соответствует более общим правилам семантики массивов, и в результате специфицировать язык и реализовывать компилятор для него становится проще, чем язык, в котором строки являются отдельными и уникальными типами данных. Но выбранный в С подход приводит и к определённым издержкам: некоторые операции со строками являются менее эффективными, чем в других языках, поскольку код приложения или библиотечная функция должны тратить время на поиск конца строки; по этой же причине в языке мало встроенных функций для работы со строками. Также ответственность за выделение и контроль буферов, в которых размещаются данные строк, в основном ложится на разработчика. Тем не менее, такая реализация строк в языке хорошо себя зарекомендовала.

С другой стороны, даже не беря в расчёт строки, использованная в С реализация массивов в целом оказывает неудачное влияние как на возможность оптимизации генерируемого кода со стороны компилятора, так и на будущие расширения языка. Активное использование указателей в программах на С (как объявленных в явном виде, так и используемых для доступа к элементам массивов) означает, что компиляторы должны осторожно проводить оптимизацию и использовать методы, которые тщательно учитывают поток данных. Хорошо написанные компиляторы в существенном числе случаев могут оценить, в результате чего может произойти изменение указателей, но некоторые важные варианты работы с ними всё равно остаются сложными для анализа. Например, для функций с аргументами-указателями, являющимися элементами массивов, компилятору сложно сгенерировать эффективный код на ЭВМ с векторными процессорами, потому что редко можно определить, что один аргумент-указатель не «перекрывает» данные, на которые также ссылается другой аргумент, или которые доступны за пределами функции. Если смотреть на это более фундаментально, то спецификация С настолько конкретно описывает семантику массивов, что изменения или расширения, рассматривающие массивы как более примитивные объекты и разрешающие использовать их «в целом» (а не путём доступа к отдельным элементам), становится трудно вписать в рамки существующего языка. Даже добавление в язык расширения, позволяющего объявлять и использовать многомерные массивы, размер которых определяется динамически, является не такой уж простой задачей [MacDonald 1989; Ritchie 1990], хотя это значительно облегчило бы написание библиотек для работы с числовыми данными. Таким образом, С охватывает наиболее важные области применения строк и массивов, возникающие на

практике, с помощью единообразного и простого механизма, но это создает проблемы для реализации высокоэффективных компиляторов и внедрения в язык расширений.

Конечно, выше указаны далеко не все недостатки, присутствующие в реализации и описании языка C. Существует также критика более общего характера, выходящая за рамки обсуждения конкретных моментов. Главное из этих замечаний указывает на то, что язык и его типовое окружение мало применимы для создания очень крупных приложений. Область видимости объектов имеет только 2 уровня: «внешний» (доступ возможен из любого объекта) и «внутренний» (доступ возможен только в функции, в которой объявлен объект). Промежуточный уровень видимости (например, в пределах одного файла исходного кода) почти не учитывается в спецификации языка. Таким образом, встроенная поддержка модульности фактически отсутствует, и разработчики вынуждены создавать свои собственные соглашения для её реализации.

Аналогичным образом имеется только 2 модификатора объектов, управляющих их «временем жизни»: auto – такие объекты хранятся в памяти, пока выполняется функция, в которой они объявлены, и static – такие объекты хранятся в памяти, пока выполняется программа. Динамическое выделение памяти за пределами стека возможно только с помощью библиотечной функции, и бремя управления такой памятью ложится на разработчика: автоматическая сборка мусора противоречит духу C.

В чём причины успеха?

Язык C стал гораздо более успешным, чем могли предполагать его создатели. Какие же качества языка способствовали его широкому распространению?

Несомненно, самым важным фактором стал успех операционной системы Unix; это сделало язык доступным для сотен тысяч людей. Конечно же, верно и обратное – использование C в качестве языка системного программирования Unix позволило обеспечить сравнительно лёгкий перенос ОС на другие ЭВМ, что сыграло важную роль в её успехе. Но применение языка на других платформах предполагает, что у него есть более фундаментальные преимущества.

Несмотря на некоторые аспекты, которые кажутся новичкам (а иногда даже опытным разработчикам) загадочными, C остаётся простым и компактным языком, трансляцию которого могут осуществлять такие же простые и компактные компиляторы. Его типы данных и операторы хорошо соответствуют тому функционалу, который предоставляет аппаратное обеспечение, и для людей, привыкших к ЭВМ и понимающих, как они устроены, не составит труда изучить идиомы, позволяющие создавать эффективный и экономно использующий ресурсы код. В то же время язык достаточно абстрагирован от деталей реализации конкретной платформы, что позволяет обеспечить переносимость программ.

Не менее важно и то, что язык C и ядро его стандартных библиотек всегда оставались «в контакте» с реальными задачами. Они были разработаны не в изоляции, чтобы доказать какую-то точку зрения или служить академическим примером, а как инструмент написания полезных программ, и предназначались для взаимодействия с «серьёзной» операционной системой и рассматривались как средства создания крупных приложений. Экономия и прагматизм – вот, что

повлияло на C; язык закрывает потребности большинства разработчиков, но не пытается предоставить слишком много возможностей.

Наконец, несмотря на изменения, которые произошли в C со времён выпуска первого издания K&R (которое, по общему признанию, было неформальным и неполным), текущая версия языка, используемая миллионами разработчиков, остаётся удивительно стабильной и унифицированной по сравнению с другими широко используемыми языками – например, Pascal и FORTRAN. Существуют разные диалекты C – среди них наиболее известны тот, который был описан в первом издании K&R, и версия из свежего стандарта – но в целом C получил гораздо меньше расширений и дополнений, чем другие языки. Возможно, наиболее важным из таких расширений является концепция «ближних» и «дальних» указателей (near-pointer и far-pointer), предназначенная для учёта особенностей некоторых процессоров Intel. Хотя при разработке C переносимость изначально не являлась одной из основных целей, язык оказался востребованным для написания программ (включая операционные системы) на самых разных платформах – от маленьких персональных ЭВМ до мощнейших суперкомпьютеров.

C — необычный и несовершенный язык, пользующийся огромным успехом. Благодаря ряду исторических случайностей он стал языком системного программирования, достаточно эффективным, чтобы заменить язык ассемблера, но в то же время достаточно абстрактным и свободным для создания алгоритмов и использования в самых разных окружениях.

Благодарности

Стоит кратко подытожить список непосредственных разработчиков современной версии языка C. Кен Томпсон создал язык B в 1969–1970 годах на базе языка BCPL, придуманного Мартином Ричардсом. Деннис Ритчи превратил B в C в 1971–1973 годах, сохранив большую часть синтаксиса, добавив типы данных и внеся множество других изменений, а также написав первый компилятор. Ритчи, Алан Снайдер, Стивен К. Джонсон, Майк Леск и Томпсон развивали язык в 1972–1977 годах, а портативный компилятор Джонсона до сих пор широко используется. Благодаря им и многим другим сотрудникам Bell Laboratories за этот же период значительно выросла коллекция библиотек и программ. В 1978 году Брайан Керниган и Ритчи написали книгу, которая на несколько лет стала спецификацией языка. Начиная с 1983 года комитет ANSI X3J11 занимается стандартизацией языка. Особенно стоит отметить усилия его сотрудников Джима Броди, Тома Плама и П. Дж. Плаугера, удерживающих работу комитета в правильном русле, и редакторов черновых версий (драфтов) стандарта Ларри Розье и Дэйва Проссера.

Я благодарю Брайана Кернигана, Дуга Маклроя, Дэйва Проссера, Питера Нельсона, Роба Пайка, Кена Томпсона и рецензентов HOPL за советы при подготовке этой статьи.

Список использованной литературы

[ANSI, 1989] American National Standards Institute, American National Standard for lnformation Systems-Programming Language C, X3.159-1989.

[Anderson, 1980] B. Anderson, Type syntax in the language C: an object lesson in syntactic innovation, SIGPLAN Notices, Vol. 15, No. 3, Mar. 1980, pp. 21-27.

[Bell, 1972] J. R. Bell, Threaded Code, C. ACM, Vol. 16, No. 6, pp. 370--372.

[Canaday, 19 69] R. H. Canaday and D. M. Ritchie, Bell Laboratories BCPL, AT&T Bell Laboratories internal memorandum, May 1969.

[Corbato, 1962] E J. Corbato, M. Merwin-Dagget, and R. C. Daley, An Experimental Time-sharing System, AFIPS Conference Proceedings SJCC, 1962, pp. 335-344.

[Cox, 1986] B. J. Cox and A. J. Novobilski, Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley: Reading, MA, 1986, Second edition, 1991.

[Gehani 89] N. H. Gehani and W. D. Roome, Concurrent C, Silicon Press: Summit, NJ, 1989.

[Jensen 74] K. Jensen and N. Wirth, Pascal User Manual and Report, Springer-Verlag: New York, Heidelberg, Berlin. Second Edition, 1974.

[Johnson 73] S. C. Johnson and B. W. Kernighan, 'The Programming Language B,' Comp. Sci. Tech. Report #8, AT&T Bell Laboratories (January 1973).

[Johnson 78a] S. C. Johnson and D. M. Ritchie, 'Portability of C Programs and the UNIX System,' Bell Sys. Tech. J. 57 (6) (part 2), July-Aug, 1978.

[Johnson 78b] S. C. Johnson, 'A Portable Compiler: Theory and Practice,' Proc. 5th ACM POPL Symposium (January 1978).

[Johnson 79a] S. C. Johnson, 'Yet another compiler-compiler,' in Unix Programmer's Manual, Seventh Edition, Vol. 2A, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Johnson 79b] S. C. Johnson, 'Lint, a Program Checker,' in Unix Programmer's Manual, Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Kernighan 78] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988.

[Kernighan 81] B. W. Kernighan, 'Why Pascal is not my favorite programming language,' Comp. Sci. Tech. Rep. #100, AT&T Bell Laboratories, 1981.

[Lesk 73] M. E. Lesk, 'A Portable I/O Package,' AT&T Bell Laboratories internal memorandum ca. 1973.

[MacDonald 89] T. MacDonald, 'Arrays of variable length,' J. C Lang. Trans 1 (3), Dec. 1989, pp. 215-233.

[McClure 65] R. M. McClure, 'TMG A Syntax Directed Compiler,' Proc. 20th ACM National Conf. (1965), pp. 262-274.

[McIlroy 60] M. D. McIlroy, 'Macro Instruction Extensions of Compiler Languages,' C. ACM 3 (4), pp. 214-220.

[McIlroy 79] M. D. McIlroy and B. W. Kernighan, eds, Unix Programmer's Manual, Seventh Edition, Vol. I, AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[Meyer 88] B. Meyer, Object-oriented Software Construction, Prentice-Hall: Englewood Cliffs, NJ, 1988.

[Nelson 91] G. Nelson, Systems Programming with Modula-3, Prentice-Hall: Englewood Cliffs, NJ, 1991.

[Organick 75] E. I. Organick, The Multics System: An Examination of its Structure, MIT Press: Cambridge, Mass., 1975.

[Richards 67] M. Richards, 'The BCPL Reference Manual,' MIT Project MAC Memorandum M352, July 1967.

[Richards 79] M. Richards and C. Whitbey-Strevens, BCPL: The Language and its Compiler, Cambridge Univ. Press: Cambridge, 1979.

[Ritchie 78] D. M. Ritchie, 'UNIX: A Retrospective,' Bell Sys. Tech. J. 57 (6) (part 2), July-Aug, 1978.

[Ritchie 84] D. M. Ritchie, 'The Evolution of the UNIX Time-sharing System,' AT&T Bell Labs. Tech. J. 63 (8) (part 2), Oct. 1984.

[Ritchie 90] D. M. Ritchie, 'Variable-size arrays in C,' J. C Lang. Trans. 2 (2), Sept. 1990, pp. 81-86.

[Sethi 81] R. Sethi, 'Uniform syntax for type expressions and declarators,' Softw. Prac. and Exp. 11 (6), June 1981, pp. 623-628.

[Snyder 74] A. Snyder, A Portable Compiler for the Language C, MIT: Cambridge, Mass., 1974.

[Stoy 72] J. E. Stoy and C. Strachey, 'OS6 An experimental operating system for a small computer. Part I: General principles and structure,' Comp J. 15, (Aug. 1972), pp. 117-124

[Stroustrup 86] B. Stroustrup, The C++ Programming Language, Addison-Wesley: Reading, Mass., 1986. Second edition, 1991.

[Thacker 79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, D. R. Boggs, 'Alto: A Personal Computer,' in Computer Structures: Principles and Examples, D. Sieworek, C. G. Bell, A. Newell, McGraw-Hill: New York, 1982.

[Thinking 90] C* Programming Guide, Thinking Machines Corp.: Cambridge Mass., 1990.

[Thompson 69] K. Thompson, 'Bon an Interactive Language,' undated AT&T Bell Laboratories internal memorandum (ca. 1969).

[Wijngaarden 75] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. Koster, M. Sintzoff, C. Lindsey, L. G. Meertens, R. G. Fisker, 'Revised report on the algorithmic language Algol 68,' Acta Informatica 5, pp. 1-236.

Стенограмма доклада Денниса Ритчи

Спасибо. Перед началом своего выступления я хотел бы поделиться мыслью, которую обдумываю последние несколько дней. Она касается проблемы программирования, связанной с теорией графов. Пусть у нас есть граф, узлы которого содержат записи, обозначающие язык программирования и связанного с ним человека. Рассмотрим конкретный пример со следующими узлами:

- (C, Ритчи)
- (Ada, Ишбиа)
- (Pascal, Вирт)
- (Parallel Pascal[4], Бринч Хансен)

Можно добавить к ним (Lisp[5], Стил) и (C++, Страуструп).
Есть связь от вершины **X** до вершины **Y**, в рамках которой **X.Person** публично озвучивает колкости в адрес **Y.Language**. Вопросы, которые меня интересуют:

- является ли этот граф полным?
- имеет ли он петли?
- если он неполный, то какие существуют клики?

Можно сформулировать и другие вопросы. Если бы это был конечный автомат, то можно было бы задать вопрос о его диагностируемости. Есть ли способ надавить на людей и заставить их перестать обмениваться колкостями?

Разработка языка программирования C или Как росли пять маленьких языков[6] Деннис М. Ритчи AT&T Bell Laboratories dmr@research.att.com			**Пять маленьких языков** Bliss Pascal Algol 68 BCPL C		
Ритчи	HOPL-II	1	Ритчи	HOPL-II	2

(**Слайд 1**) История языка C приведена в статье, так что я не буду её повторять. Вместо этого я хочу провести небольшое сравнение некоторых языков программирования; хотя даже не совсем так. Я планирую поговорить о нескольких языках, которым уже 20 лет. Другие люди могут обсуждать содержание этих языков. Но они были продуктами своего времени, и я собираюсь сравнить их по некоторым параметрам просто чтобы показать ход наших (разработчиков языка C) мыслей и,

[4] Возможно, имеется в виду Concurrent Pascal (прим. переводчика)
[5] Речь идёт о диалекте Common Lisp, в создании которого Гай Стил сыграл одну из ключевых ролей (прим. переводчика)

возможно, объяснить некоторые аспекты языка C. С помощью них я хочу показать, почему C такой, какой он есть.

Таким образом, фактическое название моего доклада, в отличие от статьи, – «Как росли пять маленьких языков».[7]

(**Слайд 2**) Рассмотрим 5 языков: Bliss, Pascal, ALGOL 68, BCPL и C. Все они были разработаны примерно в один и тот же период времени. Я собираюсь доказать, что во многом они схожи. И каждый преуспел по-своему – кто-то на поприще реального использования, а кто-то путём влияния на другие языки. Язык C добился успеха без «политики и маркетинга» – в том смысле, что мы вообще не занимались его продвижением; значит, у людей была потребность в таком языке. А что насчёт остальных? Почему эти языки похожи друг на друга?

(**Слайд 3**) Во-первых, сущности, которыми они манипулируют, их элементарные типы и низкоуровневые объекты по своей природе идентичны. Это просто машинные слова, интерпретируемые разными способами. Набор доступных над объектами операций в этих языках на самом деле очень похож. В этом, например, их отличие от SNOBOL, ориентированного на работу со строками, и Lisp, ориентированного на работу со списками. Языки, о которых я говорю в своём докладе – это просто хитроумно разработанные способы перетасовки битов; все разработчики начинают понимать принцип их работы, как только немного ознакомятся с архитектурой ЭВМ. Я называю их «низкоуровневыми». Все они процедурные, т. е. императивные. Во всех нет сложных управляющих конструкций, и они рассчитаны на выполнение операций присваивания. Они основаны на очень-очень старой модели выполнения: берём данные, производим над ними операции и записываем результат куда-то ещё. На них оказали сильное влияние ALGOL 60, FORTRAN и другие языки, которые обсуждались на первой конференции HOPL.

В основном эти языки были разработаны для «системного программирования» (в широком смысле этого термина). Действительно, BCPL, C и Bliss разрабатывались именно для этой цели; для неё же применялся и Pascal. ALGOL 68, на самом деле, задумывался для другого, но его тоже можно использовать в качестве языка системного программирования; когда Стив Борн пришёл в Bell Labs с компилятором ALGOL 68C, то он использовал его для тех же целей, для которых в будущем применялся C – у него была поддержка интерфейса системных вызовов Unix и т. д. Надо сказать, что под «системами» можно подразумевать не только операционные системы, но и более широкий круг понятий.

Элементы сходства

- Схожий набор элементарных типов
- Низкоуровневые
- Процедурные
- Основаны на традициях ALGOL 60 и FORTRAN
- Ориентированы на системное программирование

Ритчи	HOPL-II	3

[7] Вероятно, название доклада – отсылка к фильму Five Little Peppers and How They Grew (прим. переводчика)

Позвольте мне дать очень краткую характеристику каждому из рассматриваемых языков.

Bliss – это язык, о котором ранее на этой конференции не упоминалось. Его разработал Билл Вульф и его студенты из Университета Карнеги-Меллона. Я предполагаю, что Bliss задумывался как язык системного программирования PDP-10, а затем использовался для системного программирования PDP-11 и широкого круга связанных с ним проектов. Но потом язык вышел за рамки этих ЭВМ – в частности, его стала использовать компания Digital Equipment Corp. (DEC) – как я думаю, отчасти из-за тесных связей с университетом, которые она в то время имела. Стоит отметить, что он использовался (а может, используется до сих пор) для написания некоторых частей VMS. В Bliss имеется только один тип данных – машинное слово. В случае PDP-10 его размер составляет 36 бит, а для PDP-11 – 16 бит. Доступ к отдельным байтам и символам осуществляется с помощью специального оператора, в вызове которого указывается адрес начального бита и размер блока памяти. Таким образом, доступ к байтам/символам представляет собой частный случай доступа к набору бит заданного размера.

Во многих отношениях Bliss был инновационным языком. В нём нет оператора **goto**; он основан на выражениях. Многие объекты, которые для других языков являются «встроенными», в Bliss представлены синтаксическими макросами – особенно это касается массивов (в частности, многомерных) и структур. Массивы и структуры были реализованы как своего рода встроенные в язык функции для доступа к элементам данных, а не представляли собой тип данных.

(**Слайд 5**) Но больше всего людям запомнились в Bliss именно эти точки. Язык необычен тем, что имя переменной в выражении соответствует её адресу, а для получения значения, хранящегося по этому адресу, нужно перед именем использовать оператор «.». Например, для вычисления суммы значений переменных **A** и **B** используется следующее выражение:

```
C <- .A + .B;
```

А если вы опустите точки, то в переменную **C** будет записана сумма адресов памяти. С другой стороны, оператор присваивания реализован таким образом, что слева от него должен располагаться адрес памяти, а справа – значение, которое будет по нему записано; вот почему перед **C** нет точки.

Bliss • Разработан Вульфом и др. в университете Карнеги — Меллона • Язык системного программирования для PDP-10 и PDP-11 • Использован DEC в их продуктах • Ориентирован на работу с машинными словами; для доступа к байтам используется специальный оператор • Поддержка массивов и структур основана на макросах			**Ох уж эти точки!** Имя переменной соответствует её адресу, а для доступа к значению нужно использовать оператор «.» `C <- .A + .B;`		
Ритчи	HOPL-II	4	Ритчи	HOPL-II	5

Проблемы Bliss Никогда не использовался за пределами своей исходной платформы: отсутствовала переносимость программ и компилятора			**Pascal** Во многом похож на C, хотя, как ни странно: • взаимного влияния не было • разные намерения: дидактический подход (Pascal) против прагматичного (C) Даже проблемы этих двух языков во многом схожи (например, массивы с изменяющимся размером)		
Ритчи	HOPL-II	6	Ритчи	HOPL-II	7

Какие проблемы были у Bliss? По правде говоря, он никогда не использовался за пределами своей исходной платформы. Программы, как правило, были непереносимыми. Существовал оператор для извлечения набора бит (для работы с байтами и символами), но также применялись специфичные для PDP-10 указатели на байты, потому что разработчики языка не смогли устоять перед соблазном их использовать. Это (и другие моменты) привело к тому, что программы стали непереносимыми, потому что были привязаны к архитектуре PDP-10 или PDP-11. И, что не менее важно, непереносимым был и компилятор – он запускался только на PDP-10 и не был перенесён на PDP-11.

(**Слайд 6**) Какой бы ни была мотивация создания Bliss – основной интерес к нему на самом деле возник из-за многофазного оптимизирующего компилятора для PDP-10, созданного группой студентов. Именно он и является наследием этого языка. Проект разработки компилятора был разбит на этапы: каждый студент брался за одну из фаз компиляции, и писал работу, посвящённую конкретному типу оптимизаций. В целом, это был типичный подход Университета Карнеги — Меллона – серия отдельных программ, совместно используемых для получения нужного результата. По-моему, это хороший подход к проектированию (он также использовался при создании C.mmp и Mach).

(**Слайд 7**) Позвольте мне теперь перейти к Pascal. Я хочу привести доводы о том, что Pascal очень похож на C; кого-то это может удивить, кого-то нет. C и Pascal были разработаны в одно и то же время. Может показаться, что эти языки имели взаимное влияние друг на друга, но на самом деле его не было. От этого их схожесть кажется ещё более интересной. Языки имеют значительные отличия в деталях, но по своей сути они одинаковы; если вы посмотрите на типы данных и операторы, которые с ними работают – то увидите существенное сходство. Некоторые моменты отличаются – например, множества в Pascal в некоторым смысле являются более интересной абстракцией, чем беззнаковые целые в C, но так или иначе они представляют собой битовые поля.

И всё это несмотря на то, что намерения Вирта при создании Pascal сильно отличались от наших намерений при создании C. Вирт разрабатывал язык для обучения студентов, так что у него была дидактическая цель. В частности, из Pascal и более поздних языков Вирта я понял, что он ищет грань между безопасностью (проистекающей из ограничений языка) и выразительностью. Я думаю, что он действительно был «экономен» при обдумывании возможностей своего языка – как и я.

Даже некоторые характерные проблемы Pascal и C очень похожи. В частности, это касается обработки массивов с изменяющимся в процессе работы программы размером: стоит обсудить это отдельно.

В языке C всегда поддерживались массивы с изменяющимся размером[8] (одномерные массивы или, иначе говоря, векторы). Также обработка строк в C выполняется по тем же общим правилам, принятым для массивов (то есть строки в целом интерпретируются как массивы). Pascal, если говорить об его исходной версии, не позволял даже этого. Другими словами, в этом языке даже для одномерных массивов размер должен был быть известен на этапе компиляции. В некоторых версиях языка существует понятие «согласованных» (conformant) массивов, позволяющих процедурам принимать в качестве входных данных массивы разных размеров. Но всё же проблема до сих пор полностью не решена, и её текущий статус неясен.

То, как она решена в C – далеко не идеально, и точка зрения людей, указывающих на возникающие из-за этого проблемы с безопасностью, безусловно имеет право на существование. Тем не менее, этот подход за годы не изменился, и он действительно работает. В случае с Pascal – и его исходная версия и, вероятно, даже некоторые последующие нуждаются в расширениях, чтобы стать действительно полезными. Нельзя взять «чистый» Pascal и использовать его, например, как язык системного программирования – для этой задачи требуются вещи, которых в нём нет.

Хочу озвучить афоризм. Я придумал его не для этой конференции, а ещё пару лет назад. Он кажется очень уместным, особенно с учётом присутствующих [Стью Фельдмана и Никлауса Вирта]:

«*Make и Pascal похожи. Всем они нравятся, и все хотят их изменить*».

В обоих случаях очень хорошая идея не подразумевала, что из-за неё сразу начнутся бурные сражения.

Вот ещё один анекдот, основанный на событиях вчерашнего дня. Во время перерыва на кофе Вирт сказал мне: «Иногда ты можешь быть слишком строгим…». Интересно, что он говорил не о проектировании и реализации языка, а «проверке типов и границ», произошедшей в ходе конференции (я требовал, чтобы задаваемые выступающим вопросы были заранее записаны, а ограничение по времени на сессию вопрос-ответов строго соблюдалось).

Pascal – очень элегантный язык. Безусловно, он ещё жив. Он оказал существенное влияние на ряд других языков (являющихся его прямыми наследниками) и на разработку языков программирования в целом.

[8] Массивы переменной длины появились только в C99. Вероятно, Ритчи имеет в виду индексный доступ по указателю к динамически выделяемой памяти – поскольку синтаксис работы с массивами и указателями в C крайне близок (о чём упоминается в статье), то в широком смысле можно трактовать его как «поддержку массивов с изменяющимся размером» (прим. переводчика)

Pascal • Элегантный • Ещё живой • Имеет «наследников»			**ALGOL 68** Странно выглядит в моём списке: • «большой проект», разрабатываемый целым комитетом • имеет формализованную спецификацию • включает «сюрпризы» (например, идентификацию операторов[9]) • «сложный» язык (массивы с «гибкими» границами, срезы, параллелизм, расширяемость)		
Ритчи	HOPL-II	8	Ритчи	HOPL-II	9

(**Слайды 8 и 9**) ALGOL 68 определённо является самым странным участником моего списка. На слайде я написал, что он «разработан комитетом»[10], и уже начал это вычёркивать после недавних слов Чарльза Линдсея, но всё же оставил, потому что хочу подчеркнуть не столько сам факт участия комитета, а то, что всё было «официально». Другими словами, существовала международная организация по стандартизации, которая поддерживала работу над языком и ожидала какого-то результата. И каким бы ни был этот результат – определённо, это был крупный проект. Конечно же, сначала было разработано формализованное описание языка – ещё до написания первого компилятора. Как и в случае других интересных языков, разработанных подобным образом, это решение в будущем привело к ряду сюрпризов, проявившихся на этапе реализации; такие сюрпризы таил в себе даже ALGOL 60. На бумаге заложенный в нём механизм «вызова по имени» выглядел красиво, а затем люди начали его реализовать и поняли, что он может приводить к неожиданным последствиям. Были и другие подобные ситуации. Аналогично и в ALGOL 68 есть вещи, которые были добавлены потому, что на этапе проектирования они выглядели естественными и ортогональными, а затем, когда дело дошло до кодирования, стало ясно, что, хоть реализовать их и возможно, но довольно затруднительно. Это сложный и большой язык. В нём больше концепций, чем в других языках, пусть ряд из них и ортогонален[11]. Упомяну массивы с «гибкими» границами, срезы, параллелизм, расширяемость (особенно механизм идентификации операторов) и т. д.

Несмотря на усилия Чарльза Линдсея, я думаю, что язык действительно пострадал от процесса разработки, начавшегося с написания и согласования формализованной спецификации. Тем не менее, выбранный подход действительно был весьма практичным.

[9] ALGOL 68 позволяет разработчику создавать свои операторы или расширять поведение существующих, что приводит к вопросу о принципе идентификации операторов в исходном коде. Подробнее см. здесь в главе VI (прим. пер).

[10] Подробнее история разработки ALGOL описана в этой статье (прим. переводчика)

[11] Набор функциональных возможностей языка является ортогональным при отсутствии ограничений на совместное использование этих возможностей. Характерный пример ортогональности ALGOL 68 – совмещение конструкции IF с оператором присваивания:
X := IF a < 2 THEN 3.2 ELSE 5.2 FI ; (прим. переводчика)

ALGOL 68 • В некотором смысле очень элегантный • Определённо самый амбициозный • Тем не менее, вполне подходящий для практического применения (см. ALGOL 68C)			**BCPL** • Разработан Мартином Ричардсом • Бестиповый • Переносимый • Компактный и прагматичный		
Ритчи	HOPL-II	10	Ритчи	HOPL-II	11

(**Слайд 10**) В некотором смысле, ALGOL 68 – самый элегантный из языков, которые я сейчас обсуждал. Я думаю, что в какой-то степени он даже оказал наибольшее влияние на другие языки, хотя сам уже практически не используется. Но на этой конференции удивительно много людей произносили фразу «…на это повлиял ALGOL 68». Как описывается в моей статье, он повлиял и на язык C. Упоминание на слайде ALGOL 68C является отсылкой к тому факту, что в ранних версиях Unix (в 1970-х) у нас был компилятор A68C, который Стив Борн привёз с собой из Кембриджа. Этот компилятор не реализовывал все возможности языка, но его было вполне достаточно, чтобы уловить суть (компилятор был настолько любезен, что предупреждал меня каждый раз, когда я делал что-то не то. Самым частым сообщением было «Предупреждение! Результат логического выражения не используется», что всегда казалось мне забавным[12]. Конечно же, это означало, что я написал «A=B» вместо «A:=B»).

(**Слайд 11**) BCPL был прямым предшественником C. Он очень похож на Bliss – тоже бестиповый и создавался для системного программирования. В отличие от Bliss, он разрабатывался с расчётом на переносимость. Сам компилятор тоже был написан как переносимая программа; он генерировал читабельный промежуточный код. И несмотря на то, что единственным типом данных было машинное слово – язык можно было использовать на ЭВМ с разными размерами слов. Это был компактный язык, созданный с прагматичным подходом. Первоначальная цель его разработки заключалась в создании языка реализации CPL, более амбициозной инициативы Стрейчи и его студентов, которая так и не была осуществлена. BCPL применялся во многих местах. Например, это был один из первых языков, использованных в Xerox PARC на ЭВМ Alto.

(**Слайд 12**) Позвольте теперь мне провести некоторые сравнения (это будет единственный технический фрагмент моего доклада). Что представляет собой имя переменной, когда оно используется в выражении? В рассмотренных мной языках существует три совершенно разные интерпретации.

[12] В оригинале текст предупреждения звучит как «Voiding a Boolean», а одно из значений слова voiding – «мочеиспускание» (прим. переводчика)

Сравнение: значение имён переменных

В Bliss имя переменной обозначает адрес памяти. Для разыменования используется оператор «.»

```
A <- 1;
B <- A;  ! в B теперь адрес A
C <- .A; ! в C теперь 1
```

Ритчи	HOPL-II	12

Сравнение: значение имён переменных

В ALGOL 68 имя переменной в большинстве случае обозначает адрес памяти. Получение значения происходит путём неявного разыменования

```
int A; C ref int A = loc int C
ref int B;
int C;

A := 1;
B := A; C в B теперь адрес A C
C := A; C в C теперь 1 C
C := B; C в C повторно записано 1 C
```

Ритчи	HOPL-II	13

Во-первых, ещё один пример особенностей языка Bliss. В первом выражении вы просто присвоили переменной **A** значение **1**. Во втором выражении, когда вы говорите «пусть **B** станет **A**» – копируется адрес переменной **A**. Если вы посмотрите на значение **B** с помощью оператора печати – то увидите число, означающее некий адрес памяти. А вот если вы используете оператор «.», как в третьем выражении, то присвоите в переменную **C** значение переменной **A** (т.е. **1** – см. первое выражение). Это означает, что операция «разыменования» (этот термин пришёл из ALGOL 68) всегда должна явно указываться разработчиком, потому что в Bliss имя переменной соответствует ссылке, а не значению.

В ALGOL 68 ситуация более интересная. В большинстве случае имя переменной, как и в Bliss, фактически указывает на её расположение в памяти. На слайде 13 в первой строке при объявлении переменной **A** указывается, что она представляет собой ссылку на значение типа **int**, хранящееся в локальной ячейке памяти, выделенной для значения типа **int**. Запись «**int A**» является более короткой версией этой формулировки (а полная версия приведена в комментарии).

Позже в строке 4 мы пишем: «**A** получает значение **1**». Компилятор видит, что слева от оператора присваивания находится ссылка на **int**, а справа – значение типа **int**, и выполняет соответствующую магию, называемую «неявным разыменованием» (coercion), помещая значение **1** в ячейку памяти, связанную с переменной **A**. В строке 5 в переменную **B** записывается адрес переменной **A**, потому что **B** объявлена как ссылка на **int**; в строке 6 в переменную **C** записывается значение переменной **A** (т. е. **1**). В последней строке переменная **C** опять получит значение **1**, но теперь другим образом – путём неявного разыменования в неё переменной **B**.

Таким образом, имя переменной **A** в строках 5 и 6 интерпретируется по-разному в зависимости от контекста. «Разыменование» (то есть преобразование адреса в хранимое по этому адресу значение) происходит автоматически в тех ситуациях, когда это необходимо, и по понятным

правилам; несмотря на то, что основная семантика напоминает семантику Bliss, использовать «точки» не требуется.

Сравнение: значение имён переменных

В Pascal, BCPL и C имя переменной обозначает её значение. Для работы со ссылками применяются специальные операторы

```
int A;
int *B;
int C;

A = 1;
B = &A; /* в B теперь адрес A */
C = *B; /* в C теперь 1 */
```

Ритчи	HOPL-II	14

Влияние на компьютерную отрасль

- Bliss мёртв
- Pascal жив и имеет «наследников»
- ALGOL 68 и BCPL умирают, но продолжают оказывать влияние на новые языки
- C жив, а его потомки, вероятно, переживут и нас

Ритчи	HOPL-II	15

(**Слайд 14**) На следующем слайде показано, как всё ранее описанное реализовано в BCPL и его потомках, а также в Pascal. Мы рассматриваем переменные тех же типов: **A** представляет собой **int**, а **B** – ссылку (указатель) на **int**. Однако в этих языках имена переменных в выражениях соответствуют их значениям, а для получения адресов используется специальный оператор. Аналогично, если у вас есть переменная **B**, которая представляет собой ссылку (указатель) на **int**, то для доступа к значению, хранящемуся по этой ссылке, используется другой специальный оператор. В строке 4, где **A** присваивается значение **1**, не происходит «неявного разыменования»; вместо этого компилятор определяет, что слева от оператора присваивания находится так называемое lvalue типа **int**, уместное в данной позиции. В строке 5 оператор **&** используется для получения адреса памяти, по которому хранится значение переменной **A**; далее этот адрес записывается в переменную **B**. В строке 7 с помощью оператора ***** производится разыменование адреса для получения хранящегося по нему значения.

Эти языки (Bliss, ALGOL 68 и BCPL/B/C) дают три разных ответа на вопрос: «Каков смысл имени переменной в выражении?» Bliss говорит: «Имя означает адрес памяти; для доступа к значению, хранящемуся по этому адресу, вы должны использовать специальный оператор». ALGOL 68 отвечает: «Имя означает адрес памяти; однако правила языка описывают, в каких случаях должно производится "неявное разыменование", так что в зависимости от контекста вы можете получить как адрес, так и значение, хранящееся по нему; если же вы допустили ошибку при использовании имени переменной в выражении, то она будет диагностирована компилятором». BCPL, B, C и Pascal поясняют: «Имя означает значение переменной. Если же вы хотите получить её адрес, то должны использовать специальный оператор; другой специальный оператор позволяет получить значение, хранящееся по адресу».

Естественно, я предпочитаю подход, использованный в C, но при этом ценю то, как к этому вопросу подошёл ALGOL 68.

(**Слайд 15**) Давайте поговорим о влиянии, которое рассматриваемые языки оказали на компьютерную отрасль. Пока вы обдумываете содержимое слайда, я отвлекусь и охарактеризую их в духе доклада Фреда Брукса – одного из главных докладов этой конференции. В этом докладе «эмпирический» подход к созданию языков противопоставляется «рациональному», а прагматизм и «утилитарность» (нацеленность на практическое применение) – теоретическим изысканиям и «доктринам». Как же мы классифицируем эти языки? Создатели некоторых из них хотели решить какие-то свои проблемы, а другие – доказать некие постулаты. ALGOL 68 – беззастенчиво рационален и даже основан на «доктрине». Интересна ситуация с Pascal (возможно, он даже самый интересный в рамках данного сравнения) – в нём явно присутствует дух рационализма, но также и некоторая доля эмпирического подхода. BCPL и C, в целом, не продвигают какую-либо «идеологию» и принадлежат к лагерю «прагматичных» языков. Bliss – основанный на выражениях язык без оператора **goto**, с необычным подходом к значению имён переменных, во многом основан на рациональном подходе, как и Pascal – но, опять же, он был создан людьми, которые намеревались сами его использовать.

Я проведу ещё одно сравнение, которого нет на слайде. Из всех этих языков только в Pascal есть средства контроля точности арифметических вычислений. Авторы ALGOL 68 действительно задумывались о статической семантике имён и, в большинстве случаев, динамической семантики других сущностей. Но есть одна вещь, которая не рассмотрена в спецификации: какие числовые диапазоны имеют входные и выходные данные? Да, в языке есть типы **int**, **long int**, **long long int** и т. д., но нигде не сообщается, какие диапазоны значений у этих типов; у программиста нет контроля над ними. В языках B и BCPL вообще есть только «слова». Каков размер машинного слова? Это зависит от конкретной ЭВМ. Язык C в определённом смысле похож на ALGOL 68 – в нём тоже есть модификаторы типов, например, **long**. В стандарте языка C указывается: «Вот минимальный размер, который вы можете ожидать для **int**, для **long**, для **short**». Но это всё ещё не очень строгая формализация. В Pascal есть subrange-типы, поэтому вы можете в явном виде указать диапазон значений, который собираетесь хранить. Конечно, вы наткнётесь на ограничения, и использовать слишком большие числа у вас не получится.

Другие языки позволяют использовать очень большие числа. Различные предшественники этих языков, такие как PL/I, очень подробно специфицировали этот момент, а в их преемниках (например, в Ada) использован другой подход к этой проблеме. Часто поднимается вопрос: «Как язык может быть переносимым, если вы не знаете диапазоны целочисленных типов?» Интересный (и удивляющий меня) факт заключается в том, насколько на самом деле этот вопрос незначителен. Другими словами, хотя вам и приходится кое-что учитывать при проектировании программ, эта проблема довольно редко оказывается существенным источником ошибок – по крайней мере, так показывает мой опыт.

Позвольте мне вернуться к разговору о влиянии этих языков на компьютерную отрасль. Bliss практически исчез. Применённые в его компиляторе решения по оптимизации остаются полезными, и некоторые компании, которые использовали его, до сих пор живы. У Digital Equipment Corp. всё ещё есть много кода, написанного на Bliss, и они не знают, что с ним делать.

Pascal определённо жив. У него много прямых языков-наследников; он оказал сильное влияние и на другие языки. ALGOL 68 и BCPL умирают, но продолжают оказывать влияние на новые языки: ALGOL 68 делает это непосредственно, а BCPL – косвенно, через C. Очевидно, что C продолжает жить.

Как преуспеть в создании языка без намерения это делать Ни элегантности, ни формализации недостаточно (понимают ли это люди?) Объединить то, что люди хотят и могут получить: • доступность (технология компиляции, наличие реализаций, их распространение) • достаточная степень взаимодействия с имеющимся окружением (удобство использования языка как инструмента, возможность адаптации для решения неожиданных задач, варианты решения экстралингвистических проблем)			**Как преуспеть в создании языка без намерения это делать** Умение стареть изящно • Чем занимается ваш комитет по стандартизации? • Что войдёт в следующую версию стандарта? Лучше всего сделать всё правильно с первого раза		
Ритчи	HOPL-II	16	Ритчи	HOPL-II	17

Прямые потомки C (под ними я, главным образом, подразумеваю C++) вполне могут стать ещё более активными в ближайшие несколько лет. За исключением них, влияние C на семантический дизайн новых языков было довольно небольшим. С другой стороны, он оказал влияние на нотацию: в наши дни даже в псевдокоде часто можно увидеть фигурные скобки.

(**Слайд 16**) Позвольте мне закончить моё выступление рассуждением о том, как язык C получил такое широкое распространение по сравнению с другими языками, о которых я говорил. Я не могу дать точного ответа на этот вопрос, но попробую кое-что сказать.

По мнению некоторых людей, элегантность языка и наличие формализованной спецификации могут быть необходимы, но их явно недостаточно. Важно, чтобы люди понимали язык. Одна из проблем ALGOL 68 (несмотря на усилия Чарльза Линдсея и других) заключалась в том, что его спецификацию было тяжело читать. Ещё более важно то, что язык должен быть связан с реальными задачами, возникающими перед людьми, и помогать решать их. Для потенциальных пользователей должна быть доступна реализация языка.

Итак, нужен компилятор: должна быть возможность компилировать код на вашем языке в тех системах, которые есть у ваших пользователей.

Когда вы разрабатываете язык с новыми концепциями и начинаете с написания формализованной спецификации, то рискуете серьёзно продвинуть технологию создания компиляторов. Это может быть полезно для общества, но не принесёт никакой пользы вашему языку. Язык должен иметь реализацию, чтобы люди могли его опробовать, и нуждается в

распространении. Как я уже упоминал ранее, спецификация ALGOL 68 (как и предшествующего ему ALGOL 60) преподнесла немало сюрпризов разработчикам компилятора.

Кроме того, языки должны обеспечивать достаточную степень взаимодействия с имеющимся окружением. Языки программирования предназначены для выполнения прикладных задач, связанных с реальным миром, а не только для выражения алгоритмов, и поэтому их успех частично зависит от их полезности. Окружение, в котором используются языки, может быть различным. То, которое мы создали в ОС Unix, имеет свою специфику, и мы в полной мере воспользовались возможностями языка C для создания программных инструментов, подходящих для этой среды. Рассмотрим типичный пример: предположим, что вы хотите осуществить поиск по множеству файлов и найти строки, соответствующие регулярному выражению – примерно так, как это делает утилита grep в Unix.

На каких языках вы могли бы написать **grep**? Например, существуют очень изящные способы выражения алгоритма поиска по регулярным выражениям на языке APL. Однако традиционные системы с APL имеют «закрытое» окружение, и с их помощью крайне затруднительно создать утилиту, которую можно будет использовать на других платформах.

[Из-за недостатка времени Ритчи пропустил слайд 17]

(**Слайд 18**) Вот как добиться успеха: будьте удачливы. Ухватитесь за динамично развивающиеся проекты. Если вы оказались в нужном месте и в нужное время – то продолжайте путь.

Как преуспеть в создании языка без намерения это делать Будьте удачливы		
Ритчи	HOPL-II	18

Стенограмма сессии вопросов и ответов

Берни Галлер (Университет Мичигана): по поводу разыменования в ALGOL 68 – вы сказали, что оно происходит «по понятным правилам». На мой взгляд, неявное и автоматическое разыменование только затуманивает происходящее. В Pascal и C всё гораздо прозрачнее, разве нет?

Ритчи: я плохо сформулировал свою мысль. Я хотел сказать, что в ALGOL 68 неявное разыменование функционирует довольно понятно с точки зрения дизайна языка, но не имел в виду, что пользователям от этого легче понять происходящее. Вы совершенно правы. То, как работает неявное разыменование (и в каких случаях оно происходит), на самом деле является одним из камней преткновения для пользователей ALGOL 68. Но с точки зрения разработчика языка – если поразмыслить, то становится понятно, как всё устроено.

Энди Микель (MCAD Computer Center): вы знаете, что компилятор Apollo DOMAIN C был создан на базе компилятора Apollo Pascal (сохранив от последнего проверку типов) и что при его использовании для компиляции Unix он обнаружил неинициализированные переменные и «ведущие в никуда» указатели? (но не было случаев извлечения квадратных корней из указателей)

Ритчи: я не удивлён последнему факту! Да, C не является типобезопасным исключительно из-за бахвальства (или чего-то в этом роде) его авторов. Фактически, для проекта, которым мы сейчас занимаемся в нашей рабочей группе, используется гораздо более строгий компилятор, чем наш исходный компилятор C. Он обеспечивает соблюдение правил стандарта ANSI/ISO C и, в частности, требует прототипов функций, если только вы очень любезно не попросите не обращать внимания на их отсутствие. Таким образом, этот и другие современные компиляторы C фактически обнаруживают множество ошибок типизации.

Роберт Тау (MIT AI Lab): конструкция «valof … resultis», присутствующая в языке BCPL, затем была перенесена в B. Почему от неё отказались в C?

Ритчи: её не было в B; это упоминается в моей статье. Язык B в действительности был очень компактным. Стью Фельдман упоминал об этом вчера. Хочу напомнить, что первый компилятор B помещался в 8 000 байт. Первый компилятор C умещался примерно в 16 000. Может быть, тут дело как с гепардами; считается, что численность популяции африканских гепардов когда-то сокращалась до 17 особей или около того. Этот вид протиснулся в очень узкое «эволюционное окно». То же самое произошло с языком C.

Герберт Клерен (Тюбингенский университет): одобряете ли вы уточнения, сделанные в стандарте ANSI C, или согласны с той частью сообщества, которая считает, что его нужно называть «Anti-C»? Вы сами программируете на ANSI C?

Ритчи: определённо не согласен [Ритчи ответил сразу, не дослушав второй вопрос]. На слайде, который, к сожалению, я не успел показать, было написано, что один из способов добиться успеха — это передать язык в руки хорошего комитета по стандартизации, что я и сделал. Слушайте, там много деталей, о которых можно поспорить, но получился вариант языка, который действительно превосходит ранние версии. Конечно же, он не хуже их. И да, я программирую на ANSI C, и считаю, что комитет проделал хорошую работу.

Рич Миллер (SHL System House): книга K&R стала де-факто стандартом языка. Насколько это было важным и удачно ли сложились обстоятельства?

Ритчи: мне повезло. Отмечу один факт об языке C – долгое время он не нуждался в расширениях. И де-факто K&R была стандартом, хотя настоящего стандарта не было. Справочное руководство, конечно, не такое точное, как могло бы быть, и сейчас оно устарело.

Эндрю Блэк (DEC CRL[13]): вы как-то назвали синтаксис объявлений C «интересным экспериментом». На ваш взгляд, каковы его результаты?

Ритчи: не знаю; мне он до сих пор нравится. Я немного рассуждаю об этом в статье. Рави Сети заметил, что если бы оператор * использовался как постфиксный, а не префиксный, то синтаксис внезапно стал бы красивым, линейным и ясным. Думаю, семантика языка тоже до сих пор интересна. Полагаю, что выбранный при создании C подход жизнеспособен. Возможно, не всё сделано лучшим образом, но я считаю, что решения оправданы.

Гай Стил (Thinking Machines): если бы вы могли повторить процесс разработки C – сделали ли бы вы что-то иначе?

Ритчи: подался бы в монахи? Нет. Это очень сложный вопрос. Опять же, в моей статье есть некоторые мысли по этому поводу. Есть много мелких деталей, о которых сейчас можно сказать: «Ну и ну, лучше было бы сделать это по-другому». В широком смысле, мой ответ – нет; я вполне доволен результатами. Я думаю, что многие проблемы просто неизбежны; как только вы примите какое-то решение – вам придётся столкнуться с его последствиями и продолжить работу. Я не хочу вдаваться в подробности того, что бы я сделал иначе. Опять же, отсылаю вас к статье – если только вы не зададите какой-то конкретный вопрос.

Джефф Сазерленд (Object Databases): в своей статье вы пишите: «структуры (в C) стали объектами первого класса». Понятие объектов первого класса теперь используется в спецификации языка SQL3 (поскольку я поднял этот вопрос на совместном совещании ANSI X3H7/X3H2). Каково ваше точное определение объекта первого класса?

Ритчи: я просто предполагаю, что они (структуры) поддерживают все необходимые операции. В частности, изначально функция не могла вернуть структуру или получить её при вызове. Но такая возможность описывается даже в первом издании K&R. Если же им не хватает какой-то очевидно требуемой ортогональности – давайте считать их людьми второго сорта[14].

Никлаус Вирт (ETH): вы упомянули сходство C и Pascal. Какие различия этих языков вы считаете наиболее важными?

Ритчи: кажется, я уже говорил о некоторых из них. Важное различие заключается в том, что спецификация Pascal (хотя её первая версия и не была идеальной) до какого-то уровня полностью определяет семантику языка. Более того, что не менее (или даже более) важно – общепринятый стиль использования Pascal поощряет проверку правил семантики во время компиляции или на

[13] Cambridge Research Laboratory

[14] Понятие объектов первого и второго классов было предложено в 1967 г. Кристофером Стрейчи в статье «Understanding Programming Languages», где процедуры языка Algol, в противоположность действительным числам, он сравнил с подвергающимися социальной дискриминации «людьми второго сорта» (англ. second-class citizens).

этапе выполнения. Если вы прочитаете спецификацию C – то увидите, что там написано, какие аспекты, связанные с безопасностью языка, определены, а какие нет. В частности, совсем несложно реализовать проверку границ массивов. Проверка указателей и диапазонов типов тоже реализуема. Существует множество всяких возможностей, но большинство реализаций языка их не предоставляет. Кто-то очень легко может раскритиковать такой подход. Я думаю, существенная разница между языками заключается в стиле использования, а не отличиях в правилах семантики, синтаксиса и т. п.

Берни Галлер (Университет Мичигана): какой стиль получения адресов и разыменования ссылок проще выучить и использовать? Были ли какие-то эмпирические исследования на этот счёт?

Ритчи: мне точно неизвестно о каких-либо исследованиях. Я знаю, что когда обсуждаю с кем-то Bliss, то обычно слышу жалобы на «точки». Стиль, который остаётся широко используемым – это стиль языков семейства C; и стиль ALGOL 68 с его неявным разыменованием, и идея Bliss сделать всё предельно явным – очень необычные решения, которые не получили популярности.

Адам Рифкин (Калифорнийский технологический институт): вы ожидали, что язык C получит такое широкое распространение, какое мы видим сегодня? Если бы вы знали, что так произойдёт – изменили ли какие-то решения, принятые на этапе его проектирования?

Ритчи: нет, я не ожидал этого; язык C разрабатывался как полезный инструмент для моих коллег. Что касается решений: оглядываясь назад, я понимаю, что самым большим недостатком исходной версии языка была ограниченность прототипов функций (в них не указывались типы аргументов). Их добавление, безусловно, является самым важным изменением, внесённым комитетом ANSI, и его следовало сделать раньше.

Билл Маккиман (DEC): можно ли специфицировать язык C более чётко, чем это сделано в текущей версии его стандарта?

Ритчи: да, конечно. Некоторые фрагменты стандарта мне не очень нравятся – например, объяснение деталей того, как выполняется макрорасширение, а также понятие «линковки» и то, как оно определяет область видимости статических и внешних объектов. Это связано с тем, что ни я, ни комитет не смогли придумать чистую, унифицированную модель работы этих концепций с учётом наличия существующих различающихся реализаций. Тем не менее, я высоко ценю работу X3J11. Если кому-то не нравится язык C, то, вероятно, не нравится сам по себе, а не за формулировки его стандарта, и наоборот.

Биография Денниса М. Ритчи

Деннис М. Ритчи является руководителем отдела исследований в области вычислительной техники компании AT&T Bell Laboratories.

Он начал в ней работать в 1968 году после получения степеней бакалавра и магистра в Гарвардском университете. Ритчи помогал Кену Томпсону в создании операционной системы Unix и является основным разработчиком языка C, на котором написана Unix и многие другие системы. Он продолжает заниматься разработкой операционных систем и языков программирования.

Ритчи является членом Национальной инженерной академии США и научным сотрудником Bell Laboratories. Он является обладателем ряда наград, в том числе премии Тьюринга, премии Эмануэля Пиора, медали Ричарда Хэмминга, медали «Пионер компьютерной техники» и награды C&C Prize.