Питер Мойлан

Аргументы против Си

Peter J. Moylan (1993) The case against C // The ModulaTor, Vol. 3, No. 6, pp. 1-11.
Р. Богатырев, А. Китаев, перевод с англ.

Язык программирования Си используется повсеместно с начала 1970-х годов и, по всей видимости, на сегодняшний день он является наиболее распространенным языком среди профессионалов в области компьютерных наук. Цель данной статьи — доказать, что уже настало время переходить от него к другим более передовым языкам. Выбор языка программирования опирается, как правило, на эмоциональную основу, что отнюдь не способствует ведению беспристрастных дискуссий. Тем не менее, у меня все же есть надежда убедить вас в том, что существуют конкретные объективные причины, в силу которых язык Си является далеко не самым лучшим выбором при реализации больших программных проектов. Эти причины кроются в области восприятия программ и продуктивности работы самого программиста.

Введение

За эти заметки я взялся, поймав себя на том, что все время пишу и говорю разным людям одни и те же вещи. Чтобы не повторяться, я решил изложить свои мысли в одной статье.

Перед вами серьезный документ, хотя его заголовок и может показаться несколько легкомысленным. Широкое распространение языка Си в серьезном компьютерном программировании вызывает у меня глубокую озабоченность. Сфера применения этого языка гораздо шире, чем это предполагалось первоначально. Более того, похоже, что сторонники его использования в большинстве своем не имеют представления о прогрессе, достигнутом в области проектирования языков за последние 20 лет. Как мне кажется, достойная лучшего применения верность языку Си среди профессионалов представляет такую же серьезную проблему, как приверженность языку Basic среди любителей.

В мои намерения не входит обсуждать в этих заметках сравнительные достоинства языков процедурных (таких, как Паскаль или Си), функциональных (как Лисп) и декларативных (как Пролог). Это тема особого разговора. Моя цель скорее в том, чтобы побудить людей, работающих с процедурными языками, применять языки высокого уровня.

В последующем изложении в качестве примера современного языка программирования я буду использовать язык Modula-2. Причина проста: об этом языке я могу говорить со знанием дела. Не берусь утверждать, что Modula-2 совершенен; но с его помощью можно по крайней мере показать, что существуют языки, лишенные недостатков, свойственных языку Си.

Кстати говоря, язык C++ к числу таковых я не отношу. Об этом языке мы еще будем говорить, но немного позднее. Пока же достаточно отметить, что почти все критические замечания по отношению к языку Си, высказанные в моих заметках, в равной степени относятся и к C++. Конечно, язык C++ совершеннее, чем Си, но он не решает многих серьезных проблем, порожденных своим предшественником.

Немного истории

Первый компилятор языка Си для компьютера PDP-11, появился примерно в 1972 году. В то время PDP-11 был сравнительно новой машиной, и языков программирования для него было

реализовано не так много. Выбор в сущности ограничивался ассемблером, языками Basic и Fortran IV. (К тому времени были написаны компиляторы и интерпретаторы для некоторых других

языков, но эти языки были мало известны). Ввиду очевидных ограничений их использования для задач системного программирования, ощущалась острая потребность в новом языке.

К тому же в это время разработчики программного обеспечения стали приходить к мысли о том, что операционные системы совсем необязательно писать на языке ассемблера. Первая версия UNIX (1969-70) была написана именно на таком языке, но позднее ее почти полностью переписали на язык Си. Для этого нужен был язык, способный обойти некоторые жесткие правила, встроенные в большинство языков высокого уровня и обеспечивающие их надежность. Нужен был такой язык, который позволил бы делать то, что до него можно было реализовать только на языке ассемблера или на уровне машинного кода. Это привело к концепции машинно-ориентированных языков промежуточного уровня.

Надо сказать, что Си был не единственным таким языком, и уж никак не самым первым. В сущности, в это время машинно-ориентированные языки появлялись как грибы после дождя. (Мне довелось быть автором одного из таких языков — SGL, который в 1970-е годы наш департамент использовал в ряде проектов. В начале 1980-х годов от него отказались как от несколько старомодного). Все эти языки были похожи один на другой, как члены одной семьи, но не потому, что авторы списывали друг у друга (я, например, узнал о существовании Си, когда SGL был уже достаточно разработан), а потому, что все они были под влиянием одних и тех же идей, которые в то время витали в воздухе.

Почему язык Си стал популярным

История языка Си неразрывно связана с историей операционной системы UNIX. Эта система, как и большинство входящих в нее утилит, написана на Си; к тому же, насколько мне известно, каждый выпуск UNIX'а сопровождается компилятором на языке Си, тогда как получить компиляторы для других языков под UNIX гораздо сложнее. Итак, нам нужно рассмотреть причины быстрого распространения UNIX'а.

Очевидные причины — цена и доступность. UNIX распространялся практически бесплатно, и имелись исходные программы, позволявшие легко переносить его на другие системы. К UNIX'у прилагался целый ряд полезных утилит — написанных на Си, разумеется, — и обычно проще было оставлять их на Си, чем переписывать на другой язык. Так что для пользователя UNIX'а, желающего заниматься программированием, знание языка Си было почти обязательным.

С тех пор Си остается широко распространенным языком — по тем же причинам, что и Фортран: с расширением круга пользователей язык развивает сокрушительную инерцию. В ответ на вопрос: "Почему вы пользуетесь языком Си?", чаще всего можно услышать такие аргументы:

- доступность недорогих компиляторов;
- широкий набор библиотек подпрограмм и инструментальных средств;
- им пользуются все остальные.

Но ведь доступность компиляторов, библиотек и инструментов поддержки объясняется как раз большим числом пользователей. И, разумеется, каждое поколение программистов-педагогов обучает студентов своему любимому языку.

В числе причин популярности языка Си называют также переносимость, но это, на мой взгляд, отдает лукавством. Действительно, существует переносимая версия Си, однако убедить программистов придерживаться ее практически невозможно. Компилятор Си, который я использую, способен генерировать предупреждения о нарушении переносимости, но не составляет никакого труда написать непереносимую программу, не генерирующую предупреждений компилятора.

Почему Си остается популярным

С развитием технологии производства компиляторов потеряла актуальность та причина, что в свое время побудила специалистов взяться за разработку языков среднего уровня — я имею ввиду потребность в повышении эффективности объектного кода.

Большинство других машинных языков, появившихся примерно в то же время, что и Си, считают устаревшими. Почему же та же участь не постигла и язык Си? Бытует мнение, что Си апеллирует к "мужскому началу" программистов, которым нравится сражаться с малопонятными ошибками и находить невероятные и хитроумные решения проблем. Многим нравится и компактность кода Си. Похоже, сторонники этого языка считают, что возможность написать такой, скажем, оператор

```
**p++^=q++=*r---s
```

служит серьезным аргументом в пользу применения Си, поскольку экономит время. Скептик может возразить: экономию сведет на нет необходимость в дополнительных комментариях. Но достаточно просмотреть несколько типичных программ на языке Си, чтобы убедиться: на комментариях здесь тоже экономят — даже так называемые профессиональные программисты.

Еще один важный фактор состоит в том, что на языке Си, как полагают, можно быстрее, нежели на более структурированном языке, начать составление новой программы. (Я не согласен с этим утверждением и вернусь к его рассмотрению несколько позже.) Согласно общепринятому мнению, при работе с языком типа Modula-2 не обойтись без тщательного предварительного планирования, тогда как язык Си позволяет немедленно начать кодирование и быстрее получить результат.

Не правда ли, эти аргументы звучат знакомо? Действительно, они почти дословно повторяют доводы, выдвигавшиеся несколько лет назад в пользу языка Basic. А может быть, нынешние программисты на языке Си — это те же люди, что еще подростками возились с игрушечными компьютерами?

В то время мы утверждали, что использование Basic в качестве первого языка может создать дурные привычки, от которых трудно избавиться. Сегодня мы видим наглядное подтверждение этой мысли.

Прогресс в проектировании языков

Проследить и документировать историю разработки языков программирования — задача неподъемная, и я ее перед собой не ставлю. Многие ценные идеи вначале появлялись в малораспространенных языках и становились всеобщим достоянием лишь после того, как были реализованы в языках более известных.

Несомненно, самым важным шагом вперед была концепция языка высокого уровня, воплощенная в языках Фортран и Кобол. Они дали нам по меньшей мере три важных новых принципа: переносимость программ на разные машины, способность решать новые проблемы, которые были слишком объемны или трудны для языков ассемблера; и, наконец, расширение круга потенциальных программистов за пределы узкой группы энтузиастов, способных и желающих разгадывать тайны работы каждого конкретного процессора.

Нет нужды говорить, что были люди, по мнению которых "настоящие" программисты будут по-прежнему работать с машинными языками. Эти "настоящие" программисты все еще среди нас, и они все еще утверждают, что их особые навыки и высшие достоинства как-то компенсируют их низкую продуктивность.

Главные недостатки языка Фортран состояли в определенной бессистемности, в ряде неудобных и, как выяснилось позднее, ненужных ограничений, а также в некоторых особенностях, внесенных скорее для удобства машины, а не человека. (К примеру, имеющий три различных формы оператор IF в языке Fortran II был введен только потому, что хорошо вписывался в

машинный язык компьютера, которая сейчас уже не эксплуатируется). Часть этих недостатков была исправлена в языке Algol-60, породившем, в свою очередь, большое семейство "алголоподобных" языков. Главным достижением Algol'a в концептуальном плане было, пожалуй, введение вложенности в структуры управления, что, в свою очередь, привело к более наглядным управляющим структурам.

Иногда утверждают, что революция в структурном программировании восходит к известному письму Дейкстры (Dijkstra) "О вреде использования оператора GOTO". Это слишком упрощенное представление. И все же несомненно одно. Мысль о том, что конструкция GOTO не нужна — и даже нежелательна — стала важной ступенькой к открытию тесной взаимосвязи между продуктивностью программирования и наличием хорошо структурированных и понятных программ. Это открытие заставило разработчиков языков вновь заговорить о "сокращении концепций", т.е. о том, что язык нужно разрабатывать системно, избегая так называемых "особых случаев" и вычурных трудночитаемых конструкций.

Важным вкладом языка Паскаль было расширение сферы применения этих идей и перенесение их со структур управления на структуры данных. И хотя различные механизмы структурирования данных существовали и в более ранних языках — даже язык Си позволял описывать структуры записи — именно Паскаль органично свел их в единую систему.

Паскаль все еще можно считать перспективным языком, у которого много поклонников, но которому присущи по меньшей мере два явных недостатка.

Во-первых, его слишком рано стандартизировали. В итоге, некоторые досадные недостатки — например, непродуманный механизм ввода/вывода — так и не были исправлены в языковом стандарте. Они исправлены во многих диалектах языка Паскаля, но изменения выходят за пределы стандарта и, таким образом, не переносимы при переходе на другие платформы. Второй недостаток — программа на языке Паскаль должна (если придерживаться стандарта) существовать в виде одного файла, что делает этот язык не подходящим для действительно больших программ.

В последнее время в центре внимания специалистов оказались такие вопросы, как многократное вариативное использование программного обеспечения и эффективное управление большими программами. Главная идея здесь — модульность. Ее мы обсудим в следующем разделе.

И все же, какое же отношение имеет ко всему этому язык Си?

Ответ прост: язык Си создавали на базе опыта, извлеченного из языка Algol-60 и первых его преемников и в нем не учтены многие передовые идеи. А ведь за последние 20 лет нам стало известно много нового о проектировании языков, и теперь мы знаем, что некоторые идеи, казавшиеся плодотворными в то время, в сущности таковыми не являются. Так не пора ли сделать еще один шаг вперед — а, может быть, даже и два?

Модульность

В самом первом приближении модульность означает возможность разбиения большой программы на более мелкие, независимо компилируемые части. Язык Си предоставляет такую возможность. Ее давал даже Fortran II. И все же этого недостаточно.

Ибо на самом деле сущность модульности — это инкапсуляция данных и сокрытие информации. Суть в том, что каждый модуль должен заниматься своим конкретным типом данных, а доступ к последним возможен только через процедуры, предоставляемые этим модулем. Детали реализации структур данных должны быть скрыты. Возможность вызвать процедуру должна быть исключена, если только модуль явно не экспортирует эту процедуру. И что самое важное, при вызове модуля вам не нужно знать о нем ничего, кроме описаний и комментариев в его "видимой" части. Нужно иметь возможность создавать модуль, ничего не зная о внутренней структуре любого другого модуля.

Преимущества модульности очевидны. В любой момент времени программист может сосредоточиться на коротком фрагменте программы — обычно длиной в несколько страниц —

и не заботиться о побочных эффектах где-то в других ее частях. Можно работать со сложными структурами данных, не вникая в детали их строения. Можно заменять старый модуль новыми версиями — вплоть до изменения способа, которым реализуется структура данных, — и при этом нет нужды изменять или перепроверять другие модули. При совместной работе над программой группы специалистов проблема координации действий значительно упрощается.

Если аппаратура поддерживает сегментацию памяти, данные в каждом модуле защищены от случайного повреждения со стороны других модулей (если только указатели не передаются как параметры процедур). Поэтому легче выявлять и исправлять ошибки. Но даже без аппаратной защиты процент ошибок в программировании уменьшается. Ведь их число зависит от сложности программы, а модуль длиной в несколько страниц значительно уступает по сложности монолитной программе в сотню страниц.

Модульное программирование на языке Си возможно, но лишь в том случае, когда программист придерживается ряда довольно жестких правил.

- На каждый модуль должен приходиться ровно один header-файл. Он должен содержать лишь экспортируемые прототипы функций, описания и ничего другого (кроме комментариев).
- Внешней вызывающей процедуре об этом модуле должны быть известны только комментарии в header-файле. Программист никогда не должен испытывать потребности знать о модуле что-либо иное, кроме того, что содержится в header-файле.
- Для проверки целостности каждый модуль должен импортировать свой собственный header-файл.
- Для импорта любой информации из другого модуля каждый модуль должен содержать строки #include, а также комментарии, показывающие, что, собственно, импортируется. Комментарии должны постоянно модифицироваться. Нельзя прибегать к скрытому импорту, который возникает вследствие вложения строк #include, что обычно и происходит, когда нужно импортировать в header-файл определение типа или константу из другой части программы.
- Прототипы функций можно использовать только в header-файлах. (Это правило необходимо, поскольку язык Си не имеет механизма проверки того, что функция реализуется в том же модуле, что и ее прототип; так что использование прототипа может маскировать ошибку "отсутствия функции" — "missing function").
- Любая глобальная переменная в модуле, и любая функция, кроме той, что импортируется через header-файл, должны быть объявлены статическими.
- Следует предусмотреть предупреждение компилятора "вызов функции без прототипа" (function call without prototype); такое предупреждение всегда нужно рассматривать как ошибку.
- Программист должен удостовериться в том, что каждому прототипу, заданному в header-файле, соответствует реализованная под таким же именем в том же модуле неприватная (т.е. нестатическая в обычной терминологии Си) функция. (К сожалению, природа языка Си автоматическую проверку этого делает невозможной.)
- Следует с подозрением относиться к любому использованию утилиты grep. Если прототип расположен не на своем месте, то это, скорее всего, ошибка.
- В идеале программисты, работающие в одной команде, не должны иметь доступа к исходным файлам друг друга. Они должны совместно использовать лишь объектные модули и header-файлы.

Очевидная трудность в том, что мало кто будет следовать этим правилам, ибо компилятор не требует их неукоснительно соблюдать. Очень многие считают, что #include — это механизм сокрытия информации, а не ее экспорта. (Это подтверждается удручающе широко распространенной практикой написания header-файлов без комментариев.) Неинтуитивный смысл описания статических объектов не поощряет правильного их использования. Прототипы функций часто вводят в программу как придется, а не строго в header-файл. Программисты, полагающие, что комментарии следует писать после написания кода, едва ли будут исправно модифицировать комментарии к строкам #include. Очень многие программисты предпочитают при работе компилятора подавлять его предупреждения, поскольку те порождают слишком

много отвлекающих и "неважных" сообщений. Наконец, правило "ровно один header-файл на каждый модуль" противоречит традиции, сложившейся в среде пользователей языка Си.

Хуже того, в команде достаточно иметь одного плохого программиста, чтобы нарушить модульность всего проекта и заставить остальных тратить время на использование утилиты grep и на исправление таинственных ошибок, вызванных неожиданными побочными эффектами. По-моему, всем хорошо известно, что практически в каждой команде программистов есть хотя бы один плохой работник.

Модульный язык программирования по меньшей мере частично защищает хороших программистов от того хаоса, который создают плохие программисты. А язык Си этого сделать не в силах.

В довершение всего, даже хорошие программисты легко могут — конечно, без умысла — нарушить правила должной модульности. В языке Си нет механизма, поддерживающего правило, согласно которому каждому упомянутому в header-файле прототипу соответствует его реализация в том же модуле, или хотя бы проверяющего соответствие имен функций в модуле реализациям этих имен в header-файле. Легко забыть о необходимости реализовывать внутренние функции как приватные, поскольку по умолчанию поведение описывается от конца к началу.

Здесь все функции превращаются в экспортные, вне зависимости от того, используется ли прототип. Кроме того, легко запутаться в том, что и откуда импортируется, ибо основная информация укрыта в комментариях, а их компилятор не проверяет. Мне известен лишь один способ отслеживать, что же, собственно, импортируется: временно раскомментировать строки #include, чтобы понять, откуда поступают сообщения об ошибках.

Как правило, в модульных программах некоторые или даже все модули должны иметь части инициализации. (Структуры данных нельзя инициализировать извне соответствующего модуля, поскольку предполагается, что за его пределами детали данных не видны.) Это означает, что основная программа, написанная на языке Си, должна предусматривать корректный порядок инициализации процедур. Этот порядок задается снизу вверх: если модуль MA зависит от модуля MB, модуль MB должен быть проинициализирован до модуля MA. В любом языке, поддерживающем модульное программирование, это будет сделано без вашего участия, и вся инициализация пройдет корректно. (Вы также получите сообщение обо всех циклических ссылках, которые в большинстве случаев отражают ошибку в общем проекте программы.) Такие вещи в языке Си надо делать вручную, а если в программе с десяток или более модулей, то это начнет утомлять. Я заметил, что в большой программе на языке Си избежать циклических ссылок практически невозможно, тогда как при работе на Modula-2 они встречались мне крайне редко. Дело в том, Modula-2 дает возможность паре компилятор/компоновщик распознавать такие зависимости на ранней стадии, когда переработать программу еще не так сложно. В языке Си подобные ошибки всплывают лишь в виде таинственных сбоев при заключительном тестировании.

Коварство директивы #include

Мне часто приходилось слышать, что директива #include в языке Си выполняет в сущности те же функции, что и директива IMPORT в языке Modula-2. Но на самом деле они значительно различаются, что я сейчас и попытаюсь показать. Рассмотрим header-файл m2.h, который содержит следующие строки:

```
#include <m1.h>
/* FROM m1 IMPORT stInfo */
void AddToQueue
(stInfo* p);
```

и предположим, что директиву #include <m2.h> содержат несколько других модулей. Рассмотрим следующий вариант развития событий, весьма вероятный при работе над программой.

1. Компилируется несколько модулей, осуществляющих импорт из m2.
2. В результате изменений в проекте изменяется stInfo, задающий описание типа в m1.
3. Компилируются оставшиеся модули, осуществляющие импорт из m2.

К этому моменту программа находится уже в рассогласованном состоянии, поскольку некоторые ее модули были скомпилированы с устаревшими описаниями. Но эта ошибка, вероятно, не будет обнаружена ни компилятором, ни компоновщиком. Если вам приходилось ломать себе голову над вопросом, почему для устранения непонятных ошибок приходится постоянно использовать команду "откомпилировать все" ("Compile All"), ответ, возможно, состоит именно в этом.

Такая проблема возникает в языке Си и не встречается при работе с другими языками, предназначенными для модульного программирования, потому что header-файл в языке Си — это чисто текстовый файл. Он не позволяет отмечать время последней компиляции или вводить другой механизм проверки целостности. (Кстати, именно поэтому компиляторы языка Си работают раздражающе медленно по сравнению, скажем, с типичным компилятором Modula-2. Ведь чтение header-файла обычно занимает больше времени, чем считывание SYM-файла.)

Еще одно неприятное следствие дословного прочтения header-файла состоит в том, что содержащаяся в нем информация рассматривается, как если бы она была в файле, содержащем #include. Вокруг header-файла нет "защитного экрана". Все описанное в одном header-файле автоматически экспортируется, в сущности, во все header-файлы, упомянутые в последующих директивах #include. Что может привести к непонятным ошибкам, которые зависят от порядка строк #include. Это также означает, что поведение header-файла не полностью определяется человеком, его написавшим, а зависит от того, что предшествует этому файлу в импортирующем модуле.

Подобные проблемы возникают и с другими препроцессорными директивами, такими, как #define. Этот момент не всегда осознают до конца: воздействие #define затрагивает всю компиляцию, в том числе все импортируемые файлы. В языке Си нет возможности описать локальную текстовую константу.

Приходилось ли вам сталкиваться с такой ситуацией: компилятор сообщает об ошибке в библиотечной функции, которую вы даже не вызывали, а причина ошибки, как выясняется, — точка с запятой, которая поставлена не в том месте, да еще в файле, совершенно не имеющем к этому отношения? Подобные нелокальные явления — прямое игнорирование модульности.

С директивой #include связана еще одна проблема: эта директива функционирует по принципу "все или ничего". Проблему можно решить, помещая в каждом модуле многочисленные header-файлы; но тогда эти файлы будут поставлены под контроль импортера, а не экспортера, т.е. возникает опасность появления скрытых противоречий между модулем и его header-файлом (или файлами).

В любом случае, такая практика создает серьезные трудности в плане соглашений о формировании имен. Многие ли программисты прочитывают header-файл полностью перед тем, как решить, включать его или нет? Боюсь, что очень немногие. Более вероятна ситуация, когда #include импортирует имена, о которых импортер не имеет ни малейшего представления.

Это может стать катастрофой, если, как иногда бывает, две функции имеют одно и то же имя и те же типы параметров. (Если вы считаете это маловероятным, вспомните о тех устаревших версиях программного обеспечения, которые остаются, когда вы копируете файлы из одного места в другое.)

Компилятор не будет жаловаться; он просто предположит, что вы пребываете в благодушном настроении и решили написать прототип функции дважды. Возможно, пожалуется компоновщик, чего, впрочем, нельзя гарантировать. В результате вы можете получить совсем не ту функцию, какую импортировали, причем последует ли соответствующее предупреждение — это еще вопрос.

Проблема отчасти состоит в том, что механизм выхода компоновщика на требуемую функцию никак не связан с механизмом, с помощью которого компилятор проверяет прототипы функций.

То есть нет возможности удостовериться, принадлежит ли данному модулю данный конкретный header-файл.

Следует отметить также, что неиспользование прототипа никогда не рассматривается как ошибка. (Вот вам еще один довод в пользу того, чтобы помещать прототипы только в header-файлы и никуда больше. Проблему это не решает, зато уменьшается объем необходимых проверок, производимых вручную.) Такой порядок не вызывает сбоев в работе программы, однако того, кому в будущем предстоит ее эксплуатировать, ждет масса неприятных сюрпризов.

Скорость разработки программ

Часто приходится слышать, будто первоначальный этап разработки программы на языке Си проходит быстро, поскольку легко достижима стадия "первой чистой компиляции". (Подобное утверждение популярно и среди сторонников языка Basic). А в языках типа Modula-2 сколь-нибудь заметный прогресс в кодировании достигается лишь после проведения большой работы по предварительному планированию. Вывод: программисты на языке Си быстрее получают результат.

Это утверждение некорректно по крайней мере по трем причинам. Во-первых, оно опирается, хотя бы частично, на тот факт, что компиляторы языка Си с большей легкостью, нежели компиляторы языков более высокого уровня, принимают сомнительный исходный текст. Так в чем же здесь достоинство? Код с ошибками можно скомпилировать на любом языке. Достаточно игнорировать сообщения об ошибках. Во-вторых, если вы хотите определить, насколько далеко вы продвинулись в разработке программы, "первая чистая компиляция" — это довольно бессмысленный критерий. Она может быть важной вехой, если вы приступаете к кодированию лишь по завершении большей части работы по проектированию, а не наоборот. Но если вы действуете по принципу "кодирование, потом отладка", то после первой компиляции основная работа у вас еще впереди.

Наконец, мой опыт свидетельствует, что и утверждение неправильно еще и по своей сути. Я, например, приступаю к первой чистой компиляции уже через несколько минут после начала работы. Дело в том, что я придерживаюсь метода пошагового уточнения (известном также как нисходящее проектирование в сочетании с нисходящим кодированием). На первом этапе я компилирую строк пять кода и пару десятков строк комментария. Программа получается настолько короткой, что компилируется безошибочно — или сразу, или после обнаружения очевидных и легкоисправимых ошибок.

Ну, а если приходится разбивать большую программу на модули? Объем предварительного планирования в этом случае гораздо меньше, чем обычно считают. Если применять метод пошагового уточнения и исходить из того, что дело модуля — присматривать за типом данных, вопрос о том, какие модули необходимы, обычно решается по мере разработки программы. Более того, истинная модульность намного облегчает конструирование и тестирование программ по стадиям — ведь изменения внутри модуля можно вносить независимо от того, что происходит за его пределами.

В тех случаях, когда я фиксировал сроки реализации проектов, оказывалось, что на отладку программы на языке Си у меня уходит примерно в два раза больше времени, чем на решение проблемы равной сложности на языке Modula-2. К скорости набора текста это не имеет никакого отношения, поскольку исходные тексты обычно бывают одной длины. Все дело во времени, затраченном на отладку. При использовании Modula-2 работа в сущности закончена, когда набран последний модуль; отладчики используются редко. А в языке Си без хорошего отладчика не обойтись.

Для руководителя проекта это очень важный фактор. В бюджете большого проекта расходы на оплату труда программистов обычно занимают второе место (после расходов на администрирование), а иногда даже и первое. Пятидесятипроцентная разница в производительности может означать разницу между большой прибылью или большими убытками в проекте.

Указатели: GOTO для структур данных

Несмотря на все достижения в теории и практике структур данных, указатели остаются для программистов настоящим камнем преткновения. Некоторые языки (такие, как Фортран и Лисп) обходятся без явных указателей, но в них усложняется представление некоторых структур данных. (В Фортране, к примеру, вам приходится имитировать все с помощью массивов). При работе со сколько-нибудь сложными приложениями обходиться без указателей довольно трудно.

Это не значит, что мы должны их любить. Ведь на работу с указателями приходится значительная часть времени, расходуемого на отладку программы, и именно они создают большинство проблем, осложняющих ее разработку. Проектировщикам программного обеспечения на языках типа Modula-2 еще предстоит решить непростую задачу: ограничить сферу применения указателей модулями низкого уровня, дабы людям, работающим с программным обеспечением, не приходилось иметь с ними дело. Серьезная и, большей частью, неразрешенная проблема разработчиков языков состоит в том, чтобы создать механизмы, которые избавляют программистов от неприятностей, связанных с использованием указателей.

К сказанному стоит добавить, что можно различать важные и неважные указатели. Важным в нашем понимании считается указатель, необходимый для создания и поддержания структуры данных. Например, для связи элемента очереди с последующим элементом без указателя не обойтись. (Язык может и не называть его указателем явно, но это вопрос отдельный. Любой язык должен предоставлять возможность реализовать операцию "поиск следующего элемента".)

Указатель считается неважным, если он не является необходимым для реализации структуры данных. В типичной программе на языке Си неважных указателей намного больше, чем важных. Причины тому две. Первая состоит в том, что в среде программистов, использующих язык Си, стало традицией создавать указатели даже там, где уже существуют иные ничем не уступающие им методы доступа, например, при просмотре элементов массива. (Следует ли винить язык в том, что эта дурная привычка держится столь упорно? Этого я не знаю; я просто отмечаю, что она более распространена среди программистов на языке Си, нежели среди тех, кто предпочитает ему другие языки).

Вторая причина — правило языка Си, согласно которому все параметры функций должны передаваться по значению. Когда вам нужен эквивалент VAR-параметра языка Паскаль или inout-параметра языка Ada, единственное решение состоит в том, чтобы передать указатель. Этим во многом объясняется плохая читаемость программ на языке Си. (Справедливости ради надо признать, что C++ по крайней мере дает решение этой проблемы).

Ситуация усугубляется, когда бывает необходимо передать важный указатель в качестве входного/выходного параметра. В этом случае функции надо передать указатель на указатель, что создает затруднения даже для самых опытных программистов.

Эффективность времени выполнения

Как мне представляется, среди программистов на языке Си широко бытует следующее убеждение: поскольку этот язык близок машинному языку, программа на Си генерирует более эффективный объектный код, нежели эквивалентная программа, написанная на языке высокого уровня.

Мне неизвестно, предпринимались ли подробные исследования этого вопроса, но я имел возможность познакомиться с результатами некоторых неофициальных сравнительных исследований компиляторов языков Modula-2 и Си. Согласно этим данным, компиляторы Modula-2 генерируют код, который быстрее и компактнее кода компиляторов языка Си. Не следует принимать эти данные как нечто незыблемое, поскольку исследования были недостаточно представительными, но все же они дают некоторые основания полагать, что программы на языке Си, возможно, не столь эффективны, как принято считать.

Я где-то слышал — хотя за давностью уже не помню где — что компиляторы Си генерируют лучший код, чем программист на языке ассемблера. Мне пришлось столкнуться с подобным

явлением при тестировании своего компилятора SGL много лет назад. Я вижу объяснение в том, что если компилятор неплохо работает в таких областях, как распределение регистров, то человек вполне может что-нибудь упустить, когда ему приходится перенапрягаться, сосредотачиваясь на таких требующих особого внимания деталях.

Общее правило, на мой взгляд, таково: компилятор языка высокого уровня работает эффективнее, чем компилятор языка низкого уровня, а главная причина в том, что первый более свободен в выборе метода генерации кода. Если на языке Си вы, скажем, отказываетесь от использования индексов и устанавливаете указатель на массив, то вы принимаете это решение вместо компилятора. Да, такой подход может позволить вам получить более эффективный код, но чтобы быть в этом уверенным, вы должны многое знать о временных характеристиках инструкций данного процессора и о стратегии генерации кода, которая применяется данным компилятором. Вдобавок, это решение не является переносимым и может привести к серьезным трудностям при переходе на другую машину или на другую версию компилятора.

Еще важнее то, что скорость программы обычно в большей мере зависит от принятой вами глобальной стратегии — какие использовать структуры данных, какие алгоритмы сортировки и т.д., — чем от вопросов микроэффективности, связанных с тем, как именно написана каждая строка кода. При работе с языком низкого уровня наподобие Си труднее держать в поле зрения глобальные проблемы.

Верно, что компиляторы Си во многих случаях генерировали лучший код, нежели компиляторы Фортрана начала 1970-х годов. Это объясняется весьма тесной связью между языком Си и языком ассемблера PDP-11. Конструкции типа *p++ в языке Си появились по той же причине, что и три формы оператора IF в языке Fortran II: они были созданы под конкретную особенность архитектуры команд конкретного процессора. Стоит сменить процессор, и это преимущество исчезает.

О языке C++

Язык C++ по идее должен был преодолеть некоторые недостатки языка Си, и в какой-то степени он их преодолевает. Однако у него есть две отрицательные стороны. Он неоправданно сложен, что может заставить программистов либо игнорировать его широкие возможности, либо неправильно их использовать. Вторая проблема в том, что этот язык пытается поддерживать совместимость с языком Си, и таким образом сохраняет большинство небезопасных особенностей последнего.

Эта вторая проблема означает, что многие из недостатков, уже рассмотренных ранее, сохранились и в языке C++. Проверка типов по-прежнему довольно слабая, а программисты по-прежнему могут создавать причудливые трудночитаемые конструкции. И что самое странное, C++ не поддерживает модульность, если не считать грубого механизма #include. А ведь модульное и объектно-ориентированное программирование отлично дополняют друг друга, и, если учесть, сколько сил проектировщики C++ отдали разработке объектно-ориентированных расширений, остается только сожалеть, что они не пошли на те маленькие дополнительные усилия, которые могли бы значительно улучшить язык.

Некоторые аспекты объектно-ориентированного программирования сложны и открыты для злоупотреблений со стороны тех программистов, которые не полностью их понимают. Думаю, что с точки зрения обеспечения надежности программистам следовало бы изучать объектно-ориентированное программирование на более чистых реализациях (таких, как Smalltalk, Modula-3) перед тем, как переходить на C++.

Приятным дополнением можно считать способность передавать параметры функциям по ссылке, но механизм, избранный для этого, мог бы быть более упорядоченным. По-моему, он создан таким с единственной целью — потрафить программистам на языке Фортран, которым жизнь не в радость без конструкции EQUIVALENCE.

Перегрузка (overloading) операций и функций имеет как положительный, так и отрицательный момент. В руках опытного программиста это — мощное оружие, но неосторожное обращение с

ним может привести к полной неразберихе. Такая возможность нравилась бы мне гораздо больше, если бы компиляторы C++ могли хоть как-то выявлять недобросовестных программистов.

Если функция не совпадает с прототипом, что это — ошибка или намеренная перегрузка? Чаще всего это бывает ошибкой, но иногда в таких случаях компилятор C++ будет исходить из более оптимистичного предположения. Нужно с некоторым недоверием относиться к улучшениям в языке, если они повышают вероятность появления необнаруживаемых ошибок.

Не знаю, что и сказать о множественном наследовании. Это сильное средство, столь же сильное, что и оператор goto, но та ли это сила, которая нам нужна? Меня не покидает ощущение, что когда-нибудь в будущем в кодекс "чистого программирования" включат правило, согласно которому наследование объектов должно строго ограничиваться одинарным наследованием. Однако, я готов признать, что свидетельств тому пока нет.

Короче говоря, C++ порождает новые проблемы, не решив по-настоящему старых. Проектировщики решили продолжить традицию языка Си, согласно которой "почти все должно быть легальным". На мой взгляд, в этом была их ошибка.

Библиотеки вместо языковых механизмов

Одна из особенностей языка C++, которая делает его популярным, — это богатый набор библиотечных функций. Что ж, вещь эта и впрямь нужная, но ее не следует путать с неотъемлемыми свойствами языка. Хорошие библиотеки можно написать для любого языка, и во всяком случае большинство нормальных компиляторов позволяют программисту вызывать "чужие" процедуры, написанные на других языках.

Иногда от Си-программистов приходится слышать, что язык им нравится потому, что в нем есть функции argc, argv, printf и т.д. Мне они не нравятся: я столько намучился с printf, sscanf и тому подобным, что был вынужден написать альтернативные функции форматного ввода/вывода — но это вопрос особый. Полюбившиеся энтузиастам языка Си-функции, представляемые ими как примеры "переносимого Си", во многих случаях становятся возможными благодаря особенностям конкретного компилятора и не упоминаются даже в тех стандартах языка Си, которые эти энтузиасты считают самыми стандартными стандартами. Нужна или не нужна та или иная библиотечная функция — это законный предмет для обсуждения, но он не имеет отношения к проблеме свойств языка.

У этих функций только одно отличие от процедур других языков, основанное на чисто языковой особенности. Речь идет о правиле Си, позволяющем использовать функции с переменными числом и типом параметров. Нет слов, эта особенность дает определенные преимущества, но она с неизбежностью приводит к ослаблению проверки типов, осуществляемой компилятором. Лично я потратил много ценного времени на отладку таких вещей, как распечатка long int в формате, подходящем для int и потом долго не мог понять, почему мои вычисления давали неправильный результат. Вызов процедур проверки типов позволил бы сделать все гораздо быстрее, хотя и не так компактно.

Заключительные замечания

Ничто в данной статье не должно восприниматься как критика создателей языка Си. Лично я считаю его замечательным творением своего времени. Но наука и искусство проектирования программного обеспечения с тех пор несколько продвинулись вперед и по-видимому нам следует все же воспользоваться новыми достижениями.

Я не настолько наивен, чтобы полагать, будто критика, подобная этой, может привести к гибели языка. Верность языку — это вопрос скорее эмоциональной привязанности, нежели логических рассуждений. Но я все же надеюсь убедить (по крайней мере, некоторых людей) пересмотреть свои взгляды. Важную роль в выборе языка программирования могут играть факторы, не имеющие отношения к внутреннему качеству языка.

Один из таких факторов — доступность компиляторов. Другой — возможность многократного использования существующего программного обеспечения. Это обстоятельство может заставить людей по-прежнему пользоваться языком, даже когда тот явно уступает языкам-конкурентам. (Я и сам продолжаю применять Си в некоторых проектах — главным образом, именно по этой причине.) Но как бы то ни было, всем нам нужно избегать одного: необдуманно принимать по инерции неверное решение.

Об авторе. Питер Мойлан (Peter J. Moylan) — специалист по системам реального времени, разработке компиляторов, автор мультипроцессной среды PMOS, реализованной на языке Modula-2, профессор Университета Ньюкастла в Австралии (The University of Newcastle).