Оглавление

Об авторе

Джоэл Спольски, ветеран программной индустрии, ведет веблог «Joel on Software» (Джоэл о программировании) (*www.joelonsoftware.com*) – один из самых независимых и популярных среди программистов сайтов. Сайт Джоэла назвали «манифестом анти-Дильберта». Спольски спроектировал и написал программы, используемые миллионами людей, работал над рядом продуктов – от Microsoft Excel до пользовательского интерфейса Juno. Он основал компанию Fog Creek Software, расположенную в Нью-Йорке.

Благодарности

Я хотел бы поблагодарить своего издателя, редактора и друга Гэри Корнелла, благодаря которому стала возможной эта книга. Сотрудники Apress всегда были любезны и добры. Apress – это редкое компьютерное издательство, в котором стремятся прежде всего заботиться об авторе, и я в восторге от совместной работы с ними.

Но еще до книги был сайт, обязанный своему существованию как бесплатному хостингу и рекламе, обеспеченным Дэйвом Винером (Dave Winer) из UserLand Software, так и его энтузиазму. Я признателен Филипу Гринспену (Philip Greenspun), который убедил меня, что знаниями надо делиться и публиковать их в Интернете, чтобы и другие могли это узнать. И Ною Тратту (Noah Tratt), вселившему в меня вдохновение, сказав однажды, что некоторые свои проповеди я должен предать бумаге.

Хочу поблагодарить своих коллег из бригады Microsoft Excel, которые на старте моей карьеры научили меня тому, как разрабатывать коммерческие приложения.

За истекшие годы мой сайт посетили миллионы людей, и примерно 10 000 из них нашли время, чтобы написать мне чудесное письмо. Эти ободряющие послания – единственная причина, по которой я продолжал писать, и хотя мне не хватит места, чтобы перечислить всех поименно, я очень ценю эти письма.

Введение

Ты никогда не *стремился* стать менеджером. Как и большинство разработчиков программ, с которыми я знаком, ты был бы гораздо счастливее, если бы тебе позволили спокойно сидеть и писать код. Но ты лучший разработчик, и когда с Найджелом, прежним руководителем группы, произошел этот *несчастный* случай на банджи и с лэптопом, всем показалось естественным, что на его место надо выдвинуть тебя, звезду команды.

И вот теперь у тебя собственный кабинет (вместо одной клетки на двоих с вечным летним стажером), и ты должен заполнять все эти оценки эффективности труда, составляемые каждые полгода (вместо того чтобы с удовольствием портить себе зрение, глядя целый день в экран), если, конечно, ты не тратишь попусту время, разбирая причудливые требования примадонн программирования, фамильярно хлопающих по спине ребят из отдела продаж, творчески настроенных «конструкторов UI» (приглашавшихся, вообще-то, в качестве графических дизайнеров), которым нужны сверкающие кнопки OK/Cancel, способные *отражать* (простите, какое значение RGB для цвета «отражающий»?). И искать ответы на глупые вопросы старшего вице-президента, который почерпнул все свои знания о программном обеспечении из статьи в журнале, издаваемом Delta Airlines для своих пассажиров. «Почему мы используем Oracle, а не Java? Я слышал, что он более унифицирован».

Добро пожаловать в администрирование! Знаете, что я вам скажу? Управление программными проектами не имеет *никакого* отношения к программированию. Если вы до сих пор не занимались ничем, кроме написания кода, то вам предстоит открыть, что человеческие существа несколько менее предсказуемы, чем обычный процессор Intel.

Во всяком случае прежний руководитель группы, Найджел, никогда не преуспевал в этом. «Я не собираюсь стать одним из тех руководителей, которые проводят все свое время в бессмысленных совещаниях, – любил он говорить, и в этом была не только бравада. – Я думаю, что все-таки смогу 85% времени заниматься кодированием и только немножко – *администрированием*».

На самом деле Найджел *хотел* сказать другое: «У меня вообще нет *никакого* понятия о том, как руководить этим проектом, и я надеюсь, что если я просто буду продолжать кодировать так, как занимался этим до того, как меня назначили ответственным, то все как-нибудь само устроится». Оно, конечно, не устроилось, что в значительной мере и объясняет, почему Найджел прыгал на банджи вместе с IBM ThinkPad в тот злополучный день.

И Найджелу еще здорово повезло, что он выздоровел, с учетом всех обстоятельств, и сейчас он работает техническим директором небольшой компании WhatTimeIsIt.com, которую он создал вместе со своими друзьями по банджи, и у него есть всего шесть месяцев, чтобы сдать совершенно новую систему, созданную с чистого листа, и больше ему не удастся прикинуться больным.

~

Управление программными проектами не слишком хорошо изучено. Не существует степеней в управлении программными проектами, и не так много книг написано на эту тему. Кто-то из тех, кто работал над действительно удачными программными проектами, разбогател и ушел на покой разводить форель на фермах, не воспользовавшись возможностью передать накопленный ими опыт следующему поколению, а многие другие прогорели и нашли себе менее напряженную работу типа преподавания корректива английского языка хулиганам из городского гетто.

В результате многие программные проекты проваливаются, явно или скрыто, потому что ни у кого в команде нет представления о том, как должен управляться успешный программный проект. Поэтому очень многие команды так никогда и не выпускают свой продукт, или делают его слишком долго, или выпускают продукт, который никому не нужен. Но хуже всего то, что эти люди несчастны и ненавидят каждую потраченную на него минуту. Жизнь слишком коротка, чтобы ненавидеть свою работу.

Пару лет назад я опубликовал на своем сайте «тест Джоэла» – список из двенадцати характерных признаков успешно управляемых команд раз-

работчиков. В их числе такие вещи, как ведение базы данных по ошибкам, предложение поступающим на работу написать код во время собеседования и т. д. (не волнуйтесь, я подробно расскажу об этом). Поразительно, но множество людей написало мне, что их команда заработала всего лишь два или три очка из двенадцати.

Два или три!

Это почти сюрреализм. Представьте себе бригаду столяров, пытающихся делать мебель и не имеющих никакого представления о шурупах. Они употребляют исключительно гвозди и загоняют их в дерево с помощью туфель для чечетки, потому что никто не рассказывал им о существовании молотков.

Управление программными проектами требует совершенно иных навыков и приемов, нежели написание кода; это два совершенно различных и не связанных между собой поля. Написание кода так же отличается от управления проектом, как нейрохирургия от выпекания кренделей. Нет никаких оснований полагать, что у блестящего нейрохирурга, каким-то образом попавшего на производство кренделей в результате некоего разрыва в пространственно-временном континууме, окажется хоть малейшее представление о том, как делать эти самые крендели, даже если он *окончил* Гарвардский медицинский колледж. Но люди, однако, продолжают думать, что можно взять ведущего программиста и без какой-либо переподготовки перевести его в администраторы.

Так же как и вышеупомянутый нейрохирург, ты и Найджел были переведены на новую работу, в администраторы, где требуется контактировать – о, Боже! – с *людьми*, а не с компиляторами. И если тебе казалось, что современные компиляторы Java полны ошибок и непредсказуемы, то ты должен просто подождать, пока возникнет первая проблема с программистом-примадонной. Управление командами, состоящими из людей, заставит тебя смотреть на шаблоны C++ как на явно *элементарную* вещь.

Тем не менее *есть* приемы, позволяющие руководить успешными программными проектами. Это искусство шагнуло дальше гвоздей и туфель для чечетки. У нас в распоряжении молотки, отвертки и торцовочные пилы, умеющие снимать двойную фаску. В этой книге я поставил перед собой задачу познакомить читателя со всеми приемами, которые смог вспомнить, на всех уровнях – от составления графика работы руководителем команды до выработки конкурентоспособной стратегии исполнительным директором. Вы узнаете:

- Как набирать и мотивировать к работе самых лучших работников. (Это самый важный фактор успешного программного проекта.)
- Как делать реальные оценки и графики, и зачем они нужны.
- Как проектировать функции ПО и писать спецификации, которые действительно полезны для работы, а не для подшивок «один раз написал и можно не читать», из которых надстраивают перегородки в офисе.
- Как избежать распространенных ловушек в разработке ПО, и почему программисты всегда неправы, настаивая на том, что надо «все выбросить и начать сначала».
- Как организовывать команды и стимулировать их работу, и почему программистам нужны офисы с закрывающимися дверями.
- Когда следует писать собственный код с чистого листа, даже если есть почти пригодная версия, которую можно загрузить из Интернета.
- Почему всегда кажется, что программные проекты застопориваются через первую пару месяцев работы.
- Что означает стратегия ПО, и почему BeOS была обречена с первого же дня.
- А также многое другое.

Книга весьма субъективна. Ради краткости я был вынужден опустить слова типа «по моему мнению» в начале каждого предложения, потому что, как оказалось, каждое предложение в этой книге представляет мое личное мнение. И она не всеобъемлюща, но, пожалуй, сможет послужить хорошей отправной точкой.

А, так вы были на моем веб-сайте...

Значительная часть содержимого этой книги впервые увидела свет в виде статей на моем веб-сайте, «Джоэл о программировании» (*www.joelonsoftware.com*), где я записываю свои мысли на протяжении последних лет. Книга, которую вы держите в руках, значительно более *связна*, как я надеюсь, чем сайт (под *связностью* я подразумеваю возможность читать, лежа в ванной, не подвергаясь риску поражения электрическим током).

Для удобства чтения мы разделили книгу на три основные секции. Первая секция посвящена разработке программного обеспечения в небольших масштабах: что надо делать, чтобы выпускать ПО, безопасное для человека. Во второй части собран ряд статей об управлении программиста-

ми и командами программистов. Третья часть составлена несколько произвольно, но в основном посвящена созданию устойчивого программного бизнеса. Вы узнаете, почему bloatware всегда побеждает, чем Ben & Jerry's отличается от Amazon, и я попытаюсь доказать, что методологии разработки программного обеспечения обычно служат признаком низкой квалификации работников.

В книге есть материал, посвященный и другим темам, но вы можете сами открыть ее и начать читать.

ЧАСТЬ ПЕРВАЯ

Биты и байты: практика программирования

ГЛАВА ПЕРВАЯ

Выбор языка

5 мая 2002 ГОДА, ВОСКРЕСЕНЬЕ

Почему разработчики в зависимости от конкретной задачи выбирают тот или иной язык программирования?

Иногда, если для меня важна скорость выполнения, я выбираю чистый C.

Если мне нужна программа для Windows с возможно меньшим размером дистрибутива, я часто выбираю C++ со статической компоновкой MFC.

Для GUI, который можно выполнять на Mac, в Windows и Linux, обычно выбирают Java. При этом GUI получается не самым совершенным, но вполне работоспособным.

Для быстрой разработки GUI и отточенного интерфейса пользователя я обращаюсь к Visual Basic, зная при этом, что расплачиваться придется размером дистрибутива и жесткой привязкой к Windows.

Утилиту командной строки, которая должна работать на любой UNIX-машине и не требует особой скорости, вполне можно написать на Perl.

Если выполнение должно происходить внутри веб-броузера, то фактически можно иметь дело только с JavaScript. Хранимые процедуры SQL обычно приходится писать на том диалекте SQL, который предлагает разработчик сервера, или не писать вовсе.

В чем смысл?

Я очень редко выбираю язык, основываясь на синтаксисе. Да, я предпочитаю языки с {}; (такие как C/C++/C#/Java). И у меня есть свое мнение относительно того, каким должен быть «хороший» синтаксис. Но лишь ради

«точки с запятой» я не пойду на то, чтобы исполняемый модуль имел размер 20 Мбайт.

Это наводит меня на некоторые мысли относительно стратегии независимости от языка, реализованной в .NET. Она предполагает, что можно выбрать любой язык из тьмы известных, и все они будут действовать одинаково.

VB.NET и C#.NET фактически идентичны, если не считать мелких синтаксических различий. Прочие языки для участия в .NET должны поддерживать, по крайней мере, базовый набор характеристик и типов, чтобы корректно взаимодействовать с остальными. Но как можно создать в .NET утилиту командной строки UNIX? Как в .NET создать миниатюрную программу Windows EXE объемом меньше 16 Кбайт?

Похоже на то, что .NET позволяет нам «выбирать» язык, исходя из того, что как раз меньше всего нас волнует, – синтаксиса.

ГЛАВА ВТОРАЯ

Обращаясь к основам

11 ДЕКАБРЯ 2001 ГОДА, ВТОРНИК

На своем веб-сайте я уделяю массу времени обсуждению таких «масштабных» вопросов, как сравнительные характеристики .NET и Java, стратегия развития XML, закрепление клиентов, стратегия конкурентной борьбы, проектирование программного обеспечения, архитектура и т. д. Эти темы образуют в некотором роде слоеный пирог. Верхним его слоем оказывается стратегия программного обеспечения. Ниже мы представляем себе архитектуру, например .NET, а под ней – отдельные продукты: продукты для разработки ПО, такие как Java, или платформы, такие как Windows.

Переведем взгляд еще ниже. DLL? Объекты? Функции? Нет! Ниже! В какой-то момент мы видим строки кода, написанного на том или ином языке программирования.

Но и это еще слишком высоко. Сегодня я хочу поговорить о центральном процессоре – маленьком кусочке кремния, в котором перемещаются байты. Представьте себе, что вы – начинающий программист. Забудьте все, что вы знаете о том, как пишут программы, и спуститесь на самый нижний уровень фундаментальных положений фон Неймана. Выкиньте на какое-то время из головы J2EE и подумайте на языке *байтов*.

Зачем нам это нужно? Я думаю, что некоторые крупнейшие ошибки – даже на самых верхних уровнях архитектуры – происходят из-за слабого или неверного понимания некоторых простых вещей на самых нижних уровнях. Вы построили замечательный дворец, но его фундамент никуда не годится. Вместо аккуратных бетонных плит навален булыжник. Здание выглядит прекрасно, но время от времени по совершенно непонятным причинам ванна начинает скользить по полу.

Наберитесь терпения и вместе со мной разберите одно маленькое упражнение с использованием языка программирования C.

Вспомним, как в C устроены строки: они состоят из группы байтов, за которыми следует символ null со значением 0.[1] Отсюда вытекают два очевидных следствия:

1. Невозможно узнать, где оканчивается строка (т. е. какова длина строки), без того чтобы пройти ее всю и найти в конце нулевой символ.
2. В строке не может быть нулей. Поэтому в строке C нельзя хранить произвольный двоичный объект – скажем, картинку в формате JPEG.

Почему строки C работают таким образом? Потому что в микропроцессоре PDP-7, для которого были созданы UNIX и язык программирования C, строки имели тип ASCIZ, т. е. «ASCII с Z (зеро) на конце».

Единственный ли это способ хранения строк? Нет. Но на самом деле он один из худших. Во всех программах, кроме самых тривиальных, в API, операционных системах и библиотеках классов следует избегать ASCIZ-строк, как заразы. Почему?

Для начала напишем некий вариант функции strcat, которая присоединяет одну строку к другой.

```
void strcat( char* dest, char* src )
{
   while (*dest) dest++;
   while (*dest++ = *src++);
}
```

Разберемся, как работает этот код. Сначала мы просматриваем первую строку в поиске нулевого окончания. Найдя его, мы просматриваем вторую строку, копируя каждый раз по символу в первую строку.

Такой способ обработки и конкатенации строк вполне удовлетворял Кернигана и Ритчи,[2] но с ним связаны некоторые проблемы. Приведем пример. Допустим, что есть несколько имен, которые вы хотите соединить в одну большую строку:

```
char bigString[1000];     /* Никогда не знаешь, сколько памяти выделить... */
```

---

[1] Дополнительные сведения о строках символов можно найти на *www-ee.eng.hawaii.edu/Courses/EE150/Book/chap7/subsection2.1.1.2.html*.

[2] Brian Kernighan, Dennis Ritchie «The C Programming Language», Second Edition, Prentice Hall, 1988 (Брайан Керниган и Денис Ритчи «Язык программирования C», 2 изд., Вильямс, 2005).

```
bigString[0] = '\0';
strcat(bigString,"John, ");
strcat(bigString,"Paul, ");
strcat(bigString,"George, ");
strcat(bigString,"Joel ");
```

Все работает, правда? Да. И выглядит четко и ясно.

А насколько эффективен этот код? Это самый быстрый код? А как он масштабируется? Подойдет ли он, если нужно будет добавить миллион строк?

Нет. Этот код построен на *алгоритме маляра Шлемиля*. Кто такой Шлемиль? Это малый из следующего анекдота:

> Шлемиль устроился на работу маляром и должен был наносить разметку посредине дороги. В первый день он взял бочку краски и разметил 300 метров дороги. «Неплохо! – сказал босс. – Ты быстро работаешь!» И заплатил ему денежку.
>
> На следующий день Шлемиль осилил только 150 метров. «Ну что ж, не так здорово, как вчера, но ты все равно быстро работаешь. 150 метров – это не мало», – сказал босс и заплатил ему денежку. Еще через день Шлемиль расчертил 30 метров дороги.
>
> «Всего 30 метров!» – рассвирепел босс. – Это никуда не годится. В первый день ты сделал в десять раз больше. Что случилось?»
>
> «Ничего не могу поделать, – говорит Шлемиль. – С каждым днем приходится все дальше и дальше уходить от бочки с краской».

(Для большей убедительности можно подобрать реальные цифры.)[1] Этот неважный анекдот в точности иллюстрирует механизм соединения строк в приведенном мною коде strcat. Поскольку функции приходится каждый раз просматривать всю результирующую строку и искать этот проклятый завершающий нуль, она работает гораздо медленнее, чем в действительности необходимо, и очень плохо масштабируется. Масса кода, с которым вы сталкиваетесь ежедневно, содержит эту проблему. Многие файловые системы реализованы таким образом, что при работе с ними не рекомендуется помещать в один каталог много файлов, т. к. это резко снижает производительность. Попробуйте открыть в Windows заполненную

[1] См. обсуждение арифметики на *discussfogcreek.com/techInterview/default.asp?cmd=show&ixPos t=153*.

мусорную корзину, и вы увидите, сколько для этого требуется времени, которое явно увеличивается с ростом количества файлов нелинейным образом. Где-то там спрятался алгоритм маляра Шлемиля. Всякий раз, когда чувствуется, что время должно быть линейным, а оно оказывается порядка n-квадрат, ищите Шлемиля. Часто библиотеки функций скрывают от вас это обстоятельство. Сразу и не скажешь, что вызовы strcat, помещенные в цикл или просто расположенные один под другим, вопиют «n-квадрат!», хотя именно так оно и есть в действительности.

Как с этим справиться? Некоторые хитрые С-программисты пишут собственные функции mystrcat:

```
char* mystrcat( char* dest, char* src )
{
    while (*dest) dest++;
    while (*dest++ = *src++);
    return --dest;
}
```

Что мы здесь сделали? Ценой крайне незначительных дополнительных расходов мы возвращаем указатель на *конец* новой, более длинной строки. Благодаря этому код, вызывающий данную функцию, может выполнить дописывание к строке без ее повторного просмотра следующим образом:

```
char bigString[1000];      /* Никогда не знаешь, сколько памяти выделить... */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p,"John, ");
p = mystrcat(p,"Paul, ");
p = mystrcat(p,"George, ");
p = mystrcat(p,"Joel ");
```

Очевидно, что здесь зависимость линейная, а не квадратичная, поэтому количество конкатенируемых строк уже не пугает.

Разработчикам языка Pascal[1] эта проблема была известна, и они «решили» ее, поместив в первый байт строки счетчик байтов. Это и есть т. н. *Pascal-строки*. В них могут содержаться нули, и они не обязаны завершаться нулем.

---

[1] Разработчики PASCAL (собственно, один – Н. Вирт) создавали свое детище раньше, чем разработчики С, поэтому эта проблема была известна и тем, и другим. Оба решения имеют свои недостатки, поэтому до сих пор существует два независимых стиля представления строк: стиль С и стиль PASCAL. – *Примеч. науч. ред.*

Поскольку самое большое число, которое можно записать в байт, – это 255, Pascal-строки не могут быть длиннее 255 байт, но т. к. они не завершаются нулем, то занимают столько же памяти, сколько строки ASCIZ. Замечательно в Pascal-строках то, что не нужно организовывать цикл для того, чтобы вычислить длину строки. Узнать длину строки в Pascal можно с помощью одной команды ассемблера, не выполняя цикла. Это гораздо быстрее.

Раньше в операционных системах на Макинтошах Pascal-строки использовались повсеместно. Многие C-программисты выбирали Pascal-строки на других платформах для увеличения скорости. Excel внутренне использует Pascal-строки, из-за чего во многих случаях длина строки в Excel ограничена 255 байтами, но это и одна из причин, по которым Excel обладает столь высоким быстродействием.

Раньше литерал Pascal-строки внедрялся в код C так:

```
char* str = "\006Hello!";
```

Да, надо было сосчитать байты самому и записать это число в первый байт строки. Ленивые программисты поступали так, замедляя выполнение своих программ:

```
char* str = "*Hello!";
str[0] = strlen(str) - 1;
```

Обратите внимание, что в данном случае получается строка с нулевым окончанием (это делает компилятор) и одновременно Pascal-строка. Когда мне приходится говорить о таких строках, я обычно вспоминаю какое-нибудь короткое ругательство, ведь это проще, чем сказать *Pascal-строки, оканчивающиеся символом null*, но мы с вами находимся в приличном месте, и поэтому придется примириться с длинным названием.

Я опустил важный вопрос. Вспомните следующую строку кода:

```
char bigString[1000];      /* Никогда не знаешь, сколько памяти выделить... */
```

Раз уж мы занялись битами, нельзя оставлять это дело без внимания. Следовало сделать все корректно: узнать, сколько именно байт требуется, и выделить соответствующее количество памяти.

Разве не так?

Ведь в противном случае некий сообразительный хакер прочтет мой код и заметит, что я выделяю только 1000 байт и *надеюсь*, что этого хватит, и придумает какой-нибудь хитрый способ, чтобы с помощью `strcat` ввести 1100 байт в мои 1000 байт памяти и переписать кадр стека, изменив адрес

возврата. Тогда при возврате из функции выполнится код, написанный хакером. Именно это имеется в виду, когда говорят, что некая программа имеет уязвимость в виде *переполнения буфера*. Это было главной причиной взломов и червей в те достопамятные времена, когда Microsoft Outlook еще не сделал хакинг доступным даже подросткам.

Хорошо, значит, все эти программисты просто неучи. Им следовало посчитать, сколько памяти надо выделить.

Но язык C *не* предоставляет средств, с помощью которых это легко сделать. Вернемся к моему примеру с «The Beatles»:

```
char bigString[1000];        /* Никогда не знаешь, сколько памяти выделить... */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p,"John, ");
p = mystrcat(p,"Paul, ");
p = mystrcat(p,"George, ");
p = mystrcat(p,"Joel ");
```

Сколько же памяти *надо* выделить? Попробуем сделать это правильным путем.

```
char* bigString;
int i = 0;
i = strlen("John, ")
   + strlen("Paul, ")
   + strlen("George, ")
   + strlen("Joel ");
bigString = (char*) malloc (i + 1);
/* не забыть место для завершающего нуля! */
...
```

Мои глаза стекленеют. Наверное, вы уже подумываете, не заняться ли чем-нибудь другим. Я вас не виню за это, но, поверьте мне, здесь начинаются действительно любопытные вещи.

Мы должны один раз просмотреть все строки, только чтобы выяснить их размер, а затем снова просмотреть их при конкатенации. Во всяком случае, для Pascal-строк операция strlen, вычисляющая длину строки, выполняется быстро. Может быть, нам удастся написать такую версию strcat, которая будет сама перераспределять память.

И тут мы открываем *еще один* ящик Пандоры, из которого вылетают функции выделения памяти. Вы знаете, как работает malloc? В основе malloc

лежит длинный связанный список свободных блоков памяти, называемый *цепочкой свободной памяти* (*free chain*). Функция malloc при вызове просматривает связанный список в поисках блока памяти, который достаточно велик для запроса. Затем она разрезает этот блок надвое – на блок указанного в запросе размера и блок с оставшимися байтами – и предоставляет вам тот блок, который вы просили, а оставшийся (если он есть) помещает обратно в связанный список. При вызове функции free освобождаемый блок помещается в цепочку свободной памяти. В итоге цепочка свободной памяти разрубается на мелкие кусочки, и когда потребуется выделить сразу много памяти, то участков такого размера не окажется в наличии. Тогда malloc берет тайм-аут и начинает рыться в цепочке свободной памяти, разбираясь в ситуации и сливая мелкие соседние свободные блоки в более крупные. На это уходит уйма времени. В результате всей этой суеты malloc всегда работает не слишком быстро (она обязательно просматривает цепочку свободной памяти), а иногда совершенно непредсказуемо начинает действовать очень медленно, когда производит уборку. (Это, между прочим, в точности соответствует системам со «сборкой мусора», поэтому сетования на то, что сборка мусора снижает производительность системы, не вполне справедливы, поскольку типичные реализации malloc приводили к такого же рода снижению производительности, хотя и более умеренному.)[1]

Сообразительные программисты минимизируют возможные отрицательные эффекты malloc, выделяя блоки памяти, размер которых является степенью двойки. Например, 4 байта, 8 байт, 16 байт, 18446744073709551616 байт и т. д. По причинам, интуитивно ясным всякому, кто имел дело с конструктором Lego, это минимизирует случайную фрагментацию в цепочке свободной памяти. Хотя может показаться, что это бесполезная трата памяти, но нетрудно убедиться, что напрасно тра-

[1] Здесь автор несколько сгущает краски, описывая простейшую реализацию. Проблема эта хорошо изучена (например, Д. Кнутом), и для нее найдены приемлемые решения: а) функция malloc может придерживаться не только стратегии «наилучший фрагмент», но и многих других: «первый достаточный», «самый большой» и т. д., которые существенно уменьшают сегментацию памяти; б) освобождение free может сразу объединять соседние свободные фрагменты, для чего фрагменты памяти организовываются не в односвязный список, а например, в двусвязный; в) динамическая сборка мусора должна отслеживать текущее количество актуальных ссылок на фрагмент, что много сложнее простого признака «занят – свободен». – *Примеч. науч. ред.*

тится не более 50% свободного пространства. В итоге программа тратит не более чем вдвое больше памяти, чем ей нужно, что не так уж страшно.

Допустим, что вы написали умную функцию strcat, которая автоматически выделяет для результата новый буфер. Должна ли она всегда выделять ровно тот объем, который требуется? Мой учитель и наставник Стэн Айзенштат (Stan Eisenstat)[1] предлагает при вызове realloc удваивать размер ранее выделенной памяти. Это означает, что вызывать realloc придется не более чем $lg(n)$[2] раз, что обеспечивает приемлемую производительность даже для очень больших строк, и при этом «пропадает» не более 50% памяти.

Ну да ладно. Когда имеешь дело с байтами, то все получается, как в поговорке «чем дальше в лес, тем больше дров». Хорошо все-таки, что мы не обязаны больше писать на C! У нас есть такие замечательные языки, как Perl, Java, VB и XSLT, благодаря которым не надо думать ни о каких байтах: они как-то сами с этим справляются. Но временами водопроводное хозяйство вылезает посреди жилой комнаты, и приходится думать над тем, какой выбрать класс – String или StringBuilder, или о чем-нибудь примерно таком же, потому что компилятор не настолько сообразителен, чтобы понять все наши замыслы, и пытается помочь нам *избежать* ненароком прокравшихся алгоритмов маляра Шлемиля.

В основе этой главы лежит статья, которую мне пришлось написать, т. к. в блоге я однажды сказал, что эффективная реализация оператора SQL SELECT author FROM books невозможна, если данные хранятся в формате XML.[3] Поскольку люди не поняли, о чем я тогда говорил, а мы как раз говорим об эффективности расходования памяти и ресурсов CPU, это утверждение может быть осмыслено лучше.

Каким образом реляционная база данных реализует SELECT author FROM books? В реляционной базе данных каждая строка таблицы (например, таблицы books) имеет одинаковый размер в байтах, и у каждого поля есть постоянное смещение от начала строки. Например, если все записи таблицы books имеют длину 100 байт, а поле author имеет смещение 23, то авторы хранятся в байтах, начиная с 23, 123, 223, 323 и т. д. Какой же код позволит перемещаться к следующей записи в результатах этого запроса? В принципе он имеет такой вид:

---

1 См. *www.cs.yale.ed/people/faculty/eisenstat.html*.

2 Логарифм по основанию 2 (в русскоязычной литературе в такой нотации принято записывать логарифм по основанию 10). – *Примеч. науч. ред.*

3 См. *www.joelonsoftware.com/articles/fog0000000296.html*.

```
pointer += 100;
```

Единственная команда CPU. Ну о-о-очень быстро.

Теперь взглянем на таблицу books в XML.

```
<?xml blah blah>
<books>
  <book>
    <title>UI Design for Programmers</title>
    <author>Joel Spolsky</author>
  </book>
  <book>
    <title>The Chop Suey Club</title>
    <author>Bruce Weber</author>
  </book>
</books>
```

Вопрос на сообразительность. Какой код позволит переместиться к следующей записи?

Да-да...

Хороший программист скажет здесь, что нужно выполнить анализ XML и создать в памяти дерево, с которым можно работать достаточно быстро. Объем вычислений, который при этом придется выполнить CPU, чтобы осуществить выборку `SELECT author FROM books`, приведет вас в полное отчаяние. Как известно любому разработчику компиляторов, лексический и синтаксический анализ составляют самую медленную часть компиляции. Достаточно отметить, что для лексического и синтаксического анализа и построения в памяти абстрактного синтаксического дерева требуются многочисленные операции со строками, которые, как мы выяснили, медленны, и многочисленные операции распределения памяти, которые тоже (как выяснили мы же) медленны. При этом предполагается, что памяти *достаточно* для одновременного хранения всей конструкции. В реляционных базах данных скорость перемещения от одной записи к другой постоянна и по сути определяется *одной командой CPU*. Так фактически и задумывалось. А благодаря отображению файлов в память достаточно загрузить с диска только те страницы, с которыми вы реально собираетесь работать. В XML же, если заранее провести синтаксический анализ, скорость перемещения от записи к записи будет фиксированной, но зато предварительная работа окажется огромной, а без предварительного анализа скорость перемещения к следующей записи будет зависеть от длины предшествующей записи, измеряясь сотнями команд ЦПУ.

Я делаю из этого вывод, что нельзя основываться на XML, если требуется высокая производительность и объем данных велик. Если данных немного или от программы не требуется высокое быстродействие, то XML может оказаться замечательным форматом. А если вы действительно хотите воспользоваться преимуществами того и другого, то вам нужно придумать способ хранения метаданных наряду с XML, что-то типа счетчика длины в Pascal-строках, который будет подсказывать, где что находится в файле, чтобы не нужно было для поиска проводить синтаксический анализ. Но тогда, конечно, нельзя будет изменять файл с помощью текстового редактора, поскольку метаданные придут в негодность, так что в действительности это уже будет не XML.

Если любезный читатель проследовал вместе со мной до этого места, то я надеюсь, что он узнал нечто новое для себя или переосмыслил известное. Полагаю также, что размышления над такими скучными темами из начального курса информатики, как фактическая реализация strcat и malloc, позволят вам по-новому взглянуть на принятие новых стратегических или архитектурных решений, когда вы столкнетесь с такими технологиями, как XML. В качестве домашнего задания поразмышляйте над тем, почему микросхемы Transmeta всегда будут казаться медленными. Или почему первоначальная спецификация таблиц в HTML была столь скверной, что большие таблицы на веб-страницах трудно было открывать тем, кто подключался к Интернету через модем. Или почему COM действует чертовски быстро, но только если не приходится пересекать границы процессов. Или почему разработчики NT поместили драйвер дисплея в пространство ядра, а не пользователя.

Все это вопросы, требующие размышлений на уровне байтов и оказывающие влияние на глобальные решения в отношении архитектуры и стратегии. Вот почему я думаю, что, преподавая информатику первокурсникам, надо начинать с самых азов – с работы на C и оттуда постепенно продвигаться вверх. Я испытываю совершенное отвращение к многочисленным программам обучения, в которых Java принят в качестве хорошего начального языка, потому что он «прост» и избавляет от мороки со строками и malloc, зато позволяет освоить все эти модные штучки ООП, благодаря которым большие программы становятся чрезвычайно модульными. Такое преподавание ведет нас к катастрофе. Мы окружены целыми поколениями университетских выпускников, придумывающих алгоритмы маляра Шлемиля на каждом шагу, ничуть этого не понимая, потому что они в принципе не понимают, что работа со строками на самом низком

уровне очень обременительна, несмотря на то, что из сценария Perl этого не видно. Если вы действительно хотите кого-то учить по-настоящему, надо начинать с самых основ. Как в «Малыше-каратисте». Полировать до блеска машину. И так три недели подряд. После этого свернуть кому-нибудь шею уже просто.