Практика программирования Б. Керниган, Р. Пайк

Книга:

Практика программирования

Авторы:

Б. Керниган, Р. Пайк.

Серия:

Библиотека программиста

Год издания:

2001

Учебник по технологии программирования

Оглавление

- Введение
- Стиль
- Алгоритмы и структуры данных
- Проектирование и реализация
- Интерфейсы
- Отладка
- Тестирование
- Производительность
- Переносимость
- Нотация
- Эпилог
- Приложение: свод правил

Введение

Приходилось ли вам когда-нибудь:

- тратить кучу времени на то, чтобы закодировать неверный алгоритм?
- использовать слишком сложную структуру данных?
- при тестировании программы пропустить очевидную проблему?
- тратить день на то, чтобы обнаружить ошибку, которую можно было бы найти за пять минут?
- сталкиваться с тем, что программа должна работать в три раза быстрее и использовать меньше памяти?
- затрачивать титанические усилия на то, чтобы перевести программу с рабочей станции на PC или наоборот?
- пытаться внести изменения в чужую программу?
- переписывать программу целиком, потому что разобраться в ней не удалось?

Ну и как — понравилось?

С программистами такое происходит все время. Однако справиться с подобными проблемами часто гораздо труднее, чем хотелось бы, поскольку такие темы, как тестирование, отладка, переносимость, производительность, альтернативы проектирования и стиль, темы, относящиеся к практике программирования, как правило, оказываются вне сферы внимания информатики и учебных курсов по программированию. Большинство программистов изучают их сами по себе, — в основном, на собственном опыте, а некоторые не изучают вообще.

В мире разнообразных интерфейсов, постоянно меняющихся языков, систем и утилит, под постоянным давлением обстоятельств мы зачастую теряем из вида главные принципы, которые должны быть основанием любой хорошей программы, — простоту, четкость и универсальность.

Не уделяется должного внимания инструментам и нотациям, способам записи, которые механизируют некоторые аспекты создания программ, то есть привлекают к процессу программирования сам компьютер.

Эта книга построена как раз на основных принципах, применимых к информационным технологиям на любом уровне. К таким взаимосвязанным принципам относятся: простота, благодаря которой программы остаются короткими и управляемыми, четкость и ясность, которые облегчают понимание программ и людям, и машинам, обобщенность, означающая, что программа способна корректно работать в широком диапазоне ситуаций и нормально адаптироваться к новым ситуациям, и автоматизация, которая позволяет передавать машине наиболее утомительные и скучные части нашей работы. Рассматривая программирование на различных языках, от алгоритмов и структур данных, через проектирование, отладку, тестирование, до улучшения производительности, мы иллюстрируем универсальные концепции, которые не зависят ни от языка, ни от операционной системы, ни от конкретного задания.

Книга родилась из нашего многолетнего опыта в написании и поддержке разнообразнейших программ, в преподавании программирования и в общении с большим количеством программистов. Мы хотим поделиться знаниями, приобретенными благодаря этому опыту, чтобы помочь программистам всех уровней работать более эффективно и профессионально.

Наша книга предназначена для читателей разных категорий. Если вы школьник или студент, вам только что прочитали курс программирования и вы захотели узнать об этом предмете побольше, эта книга расширит ваше образование моментами, которые недостаточно подробно освещаются в школе. Если разработка программ составляет часть вашей работы, но не исчерпывает ее, а только дополняет другие формы, то наша книга наверняка поможет вам делать это более эффективно. Если вы профессиональный программист и чувствуете, что в свое время недостаточно изучили перечисленные выше вопросы (или же просто хотите освежить их в памяти), или если вы руководите группой программистов и хотите ставить своим подчиненным правильные задачи, материал этой книги вам обязательно пригодится.

Мы надеемся, что наши советы помогут вам писать более качественные программы. Единственное, что вам необходимо, — это иметь некоторый опыт в программировании, желательно на C, C++ или Java. Естественно, чем больше ваш опыт, тем проще вам будет понять и применить наши советы, ничто не сможет сделать эксперта из новичка за 21 день.

Для работающих с системами Linux и Unix многие примеры будут более знакомы, чем для тех, кто использовал только системы Windows и Macintosh, однако все без исключения найдут в этой книге рекомендации, которые облегчат им жизнь. Каждая из девяти глав, составляющих книгу, посвящена конкретному глобальному аспекту практики программирования.

В главе 1 обсуждается стиль программирования. Хороший стиль настолько необходим для написания хороших программ, что мы решили начать разговор именно с него. Хорошо написанные программы лучше, чем плохо написанные, — в них меньше ошибок, их легче отлаживать и модифицировать, — поэтому задумываться о стиле надо с самого начала. В этой главе затрагивается и такой важный аспект хорошего программирования, как применение идиом, соответствующих используемому языку программирования.

Алгоритмы и структуры данных — предмет обсуждения главы 2 — занимает центральное место в учебных планах всех курсов программирования. Поскольку большинство читателей в той или иной степени уже знакомо с этим материалом, мы даем лишь краткий обзор ряда алгоритмов и структур данных, которые встречаются во многих программах. Более сложные алгоритмы и структуры данных, как правило, являются производными от этих базовых вариантов, так что главное — разобраться с основами.

В главе 3 описываются проектирование и реализация небольшой программы, которая иллюстрирует применение алгоритмов и структур данных в реальном программировании. Программа реализована на пяти языках; сравнение версий позволяет увидеть, как в них обрабатываются одни и те же структуры данных и как в зависимости от языка изменяются выразительные возможности и производительность.

Интерфейсы между пользователями, программами и частями программ являются одной из фундаментальных основ программирования; успех того или иного программного продукта во многом определяется тем, насколько хорошо спроектированы и реализованы в нем интерфейсы. В главе 4 показана эволюция небольшой библиотеки для синтаксического разбора одного широко используемого формата данных. Хотя пример и невелик,-он иллюстрирует многие понятия, связанные с проектированием интерфейсов: абстракцию, сокрытие информации, управление ресурсами и обработку ошибок.

Как бы мы ни старались писать программы корректно с первого раза, ошибки в них, а следовательно, и отладка вечны. Глава 5 описывает стратегию и тактику систематической и эффективной отладки. Среди рассматриваемых здесь аспектов назовем характеристики частых ошибок и важность "нумерологии", для которой образцы отладочных печатей часто показывают, где таится ошибка.

Тестирование — это попытка дать разумное обоснование работоспособности программы. Глава 6 посвящена систематическому тестированию программ как вручную, так и с помощью'компьютера. Тесты на граничные условия проверяют потенциальные слабые места программ. Механизация и специальная тестовая оснастка позволяют осуществлять комплексное тестирование с достаточно скромными затратами. Особым типом тестов являются стрессовые тесты, они проверяют устойчивость программ.

В наши дни компьютеры настолько быстры, а компиляторы настолько хороши, что большинство программ работает достаточно быстро уже в своем первоначальном виде. Однако некоторые программы все же работают слишком медленно, или используют слишком много памяти, или и то и другое вместе. В главе 7 описываются способы, позволяющие добиться того, чтобы программа рационально использовала ресурсы и работала более эффективно.

В главе 8 разговор идет о переносимости. Удачные программы живут достаточно долго, так что меняются системы, в которых их используют; нередко приходится переносить их на другие платформы или на другие машины или использовать в других странах. Смысл переносимости в том, чтобы уменьшить расходы на поддержание программы, минимизировав количество изменений, необходимых для адаптации к новому окружению.

Существует большое количество языков программирования — и не только универсальных, которые мы используем для создания основной массы программ, но и специализированных, которые предназначены для решения узкого круга проблем. В главе 9 представлено несколько примеров, показывающих всю важность способа записи в программировании, и показано, как они могут использоваться для упрощения программ, облегчения перехода от проекта к реализации и даже для создания программ, которые бы сами писали другие программы.

Разговор о программировании, естественно, не может обойтись без демонстрации изрядного количества кода. Большинство примеров написано специально для этой книги, но некоторые небольшие фрагменты были взяты из ранее написанных программ. Мы очень старались писать свой код хорошо; все примеры тестировались на нескольких системах. Дополнительную информацию можно получить на web-сайте, посвященном этой книге (в английском варианте — The Practice of Programming):

http://tpop.awl.com

Большинство программ написано на C; есть примеры на C++ и Java; предпринят и краткий экскурс в языки скриптов. На нижнем уровне C и C++ практически идентичны, так что наши программы на C являются вполне приемлемыми программами на C++. Языки C++ и Java— прямые наследники C, они унаследовали изрядную долю синтаксиса, эффективности и выразительности C, добавив более широкие системы типов и библиотек. Мы сами в повседневной работе широко используем и эти три языка, и множество других. Выбор языка зависит от задачи: операционные системы лучше всего писать на эффективном и не давящем языке

вроде C или C++; создавать на скорую руку прототипы проще на командных интерпретаторах или языках скриптов вроде Awk или Perl; для пользовательских интерфейсов хорошо подходят Visual Basic, Tcl/Tk и Java.

В выборе языка для наших примеров есть глубокий педагогический смысл. Поскольку нет языка, который был бы одинаково хорош для решения всех задач, ни один конкретный язык не может лучшим образом представить все аспекты. Языки высокого уровня предопределяют некоторые проектные решения. Если же мы используем язык низйого уровня, мы получаем возможность для альтернативных решений проектных вопросов; показывая больше подробностей, мы можем обсуждать эти вопросы лучше. Опыт учит, что даже при использовании возможностей языка высокого уровня важно понимать, как они соотносятся с низким уровнем, без такого понимания можно упереться в проблемы производительности и загадочного поведения программы. Поэтому часто мы используем в своих примерах язык C, даже в тех случаях, когда в реальной работе мы бы выбрали какой-нибудь другой язык.

Однако надо заметить, что в большинстве своем наши советы не привязаны ни к какому конкретному языку. Выбор структуры данных зависит от используемого языка; в некоторых языках вариантов может быть немного, в других же выбор весьма богат. Однако способ, подход, применяемый для выбора, будет одним и тем же для всех случаев. Детали тестирования и отладки различаются в разных языках, но стратегия и тактика одинаковы. Большинство способов повышения эффективности программ может быть применено к любому языку.

На каком бы языке вы ни писали, вам как программисту нужно добиваться наилучшего результата с помощью тех средств, которыми вы располагаете. Хороший программист сумеет справиться со слабым языком и неудобной операционной системой, но самое идеальное программное окружение не спасет плохого программиста. Мы надеемся, что вне зависимости от имеющегося у вас на данный момент опыта программирования эта книга поможет вам писать программы лучше и с большим удовольствием.

Стиль

- Имена
- Выражения
- Стилевое единство и идиомы
- Макрофункции
- Загадочные числа
- Комментарии
- Стоит ли так беспокоиться?
- Дополнительная литература

> Есть старое наблюдение, что лучшие писатели иногда пренебрегают правилами риторики. Однако, когда они это делают, читатель обычно находит в тексте какие-то компенсирующие достоинства, достигнутые ценой этого нарушения. Пока кто-то не уверен, что он сможет сделать то же самое, ему, вероятно, лучше всего следовать правилам.
>
> Вильям Странк, Элвин Б. Уайт. Элементы стиля

Приводимый фрагмент кода взят из большой программы, написанной много лет назад:

```
    if ( (country == SING) || (country == BRNI) ||
 (country == POL) ||[ (country
      == ITALY) ) {
 /*

     * если страна - Сингапур, Бруней или Польша,
 * то временем ответа является текущее время,
 * а не время, передаваемое в параметре.
  * Переустанавливается время ответа
 * и задается день недели. */
    ...
```

Этот фрагмент тщательно написан, форматирован и комментирован, а программа, из которой он взят, работает предельно корректно; программисты, написавшие ее, по праву гордятся своим творением. Однако для постороннего глаза этот фрагмент все же представляется странноватым. По какому принципу объединены Сингапур, Бруней, Польша и Италия? И почему Италия не упомянута в комментариях? Код и комментарий не совпадают, так что один из них неверен. Не исключено, что неверны и оба. Правда, код все же исполнялся и тестировался, так что, скорее всего, он верен, а комментарий просто забыли обновить вместе с кодом. Комментарий не объясняет, что общего у перечисленных в нем стран, хотя эта информация будет просто необходима, если вам придется заниматься поддержкой этого кода.

Наша книга посвящена практике программирования — тому, как писать обычные программы. Мы хотим помочь вам писать программы, по крайней мере, не хуже той, откуда был взят пример, избегая беспокойных мест и слабостей. Мы расскажем, как с самого начала писать грамотный код и как улучшать его по мере его развития.

Начнем мы с непривычного — с обсуждения стиля в программировании. Чем лучше у вас стиль, тем проще читать ваши программы вам самим и другим программистам; хороший стиль просто необходим для хорошего программиста. Мы начинаем со стиля, чтобы в дальнейшем вы обращали на него внимание во всех примерах.

Написать программу — это больше чем добиться правильного синтаксиса, исправить ошибки и заставить ее выполняться достаточно быстро. Программы читаются не только компьютерами, но и программистами. А программу, написанную хорошо, куда проще понять, чем написанную плохо. Культура в написании кода поможет создавать программы, ошибок в которых будет гораздо меньше. К счастью, соблюдать дисциплину не трудно.

Принципы хорошего стиля программирования состоят вовсе не в наборе каких-то обязательных правил, а в здравом смысле, исходящем из опыта. Код должен быть прост и понятен: очевидная логика, естественные выражения, использование соглашений, принятых в языке разработки, осмысленные имена, аккуратное форматирование, развернутые комментарии, а также отсутствие хитрых трюков и необычных конструкций. Логичность и связность необходимы, потому что другим будет проще читать код, написанный вами, а вам, соответственно, — их код, если все будут использовать один и тот же стиль. Детали могут быть продиктованы местными соглашениями, требованиями менеджмента или самой программой, но даже если это не так, то лучше всего подчиняться наиболее распространенным соглашениям. Мы в своем повествовании будем придерживаться стиля, использованного в книге "Язык программирования С" с некоторыми поправками для С+ + и Java.

Правила, о которых пойдет речь, будут во множестве иллюстрироваться простыми примерами хорошего и плохого стиля, на контрасте все видно лучше. Кстати, примеры плохого стиля не придуманы специально; мы будем приводить куски кода из реальных программ, написанных самыми обычными программистами (даже нами самими), работавшими в обычной обстановке, когда работы много, а времени мало.

Некоторые куски будут несколько укорочены для большей выразительности, но не искажены. После "плохих" примеров мы будем приводить варианты, показывающие, как можно их улучшить. Так как все примеры взяты из реальных программ, со многими из РГИХ связано сразу несколько проблем. Обсуждение всех недостатков сразу уводило бы нас слишком далеко, так что некоторые примеры хорошего стиля будут по-прежнему таить в себе другие, неотмечаемые погрешности.

Для того чтобы вы могли без труда отличить хорошие примеры от плохих, мы будем начинать строки сомнительного кода со знака вопроса, как, например, в этом фрагменте (помните — все фрагменты взяты из реальных программ):

```
? «define ONE 1
? «define TEN 10
? «define TWENTY 20
```

Что может быть спорным в этих макроопределениях? А вы представьте себе изменения, которые необходимо будет внести в программу, если массив из TWENTY (двадцати) элементов понадобится увеличить. Нужно по меньшей мере заменить все имена на более осмысленные, указывающие на роль данного значения в программе:

```
# define INPUT_MODE 1
# «define INPUT_BUFSIZE 10
# «define OUTPUT_BUFSIZE 20
```

Имена

Что в имени? Имя переменной или функции помечает объект и содержит некоторую информацию о его назначении. Имя должно быть информативным, лаконичным, запоминающимся и, по возможности, произносимым. Многое становится ясным из контекста и области видимости переменной: чем больше область видимости, тем более информативным должно быть имя.

Используйте осмысленные имена для глобальных переменных и короткие — для локальных. Глобальные переменные по определению могут проявиться в любом месте программы, поэтому их имена должны быть достаточно длинными и информативными, чтобы напомнить читателю об их предназначении. Полезно описание каждой глобальной переменной снабжать коротким комментарием:

```
int npending = 0; // текущая длина очереди ввода
```

Глобальные функции, классы и структуры также должны иметь осмысленные имена, поясняющие их роль в программе.

Для локальных переменных, наоборот, лучше подходят короткие имена; для использования внутри функции вполне сойдет просто п, неплохо будет смотреться npoints, а вот numberO.f Points будет явным перебором.

Обычно используемые локальные переменные по соглашению могут иметь очень короткие имена. Так, употребление i и j для обозначения счетчиков цикла, р и q для указателей, s и t для строк стало настолько привычным, что применение более длинных имен не принесет никакой пользы, а наоборот, может даже навредить. Сравните

```
for (theElementIndex = 0; theElementIndex < numberOfElements; ?
theElementIndex-н-)
? elementArrayftheElementIndex] = theElementIndex;
```

и

```
for (i = 0; i < nelems; i++) elem[i] = i;
```

Бывает, что программисты используют длинные имена независимо от контекста. Это ошибка: ясность часто достигается краткостью.

Существует множество соглашений и местных традиций именования. К наиболее распространенным правилам относятся использование для указателей имен, начинающихся или заканчивающихся на р (от pointer, указатель), например nodep, а также заглавных букв в начале имен Глобальных переменных и всех заглавных — в

именах КОНСТАНТ. Некоторые программистские фирмы придерживаются более радикальных правил, таких как отображение в имени переменной информации об ее типе и способе использования. Например, реп будет означать указатель на символьный тип (pointer to char), a st rFrom и strTo — строки для ввода и вывода, соответственно. Что же касается собственно напи-. сания имен, то.использование npending, numPending или num_pending зависит от личных пристрастий, и принципиальной разницы нет. Главное — соблюдать основные смысловые соглашения; все, что не выходит за их рамки, уже не так важно.

Использование соглашений облегчит вам понимание как вашего собственного кода, так и кода, написанного другими программистами. Кроме того, вам будет гораздо проще придумывать имена для новых переменных, появляющихся в программе. Чем больше размер вашего кода, тем важнее использовать информативные, систематизированные имена.

Пространства имен (namespaces) в C++ и пакеты (packages) в Java позволяют разграничивать области видимости переменных и тем самым помогают сохранить ясность кода без использования чрезмерно длинных имен.

Будьте последовательны. Сходным объектам давайте сходные имена, которые бы показывали их сходство и подчеркивали различия между ними.

Вот пример на языке Java, где имена членов класса не только слишком длинны, но и абсолютно непоследовательны:

```
    ? class UserQueue { .
? int noOfItemsInQ, f rontOfTheQueue, queueCapacity;
? public int noOfIlsersInQueueO {....}
```

Слово "queue" (очередь) используется в названиях как Q, Queue и queue. Однако, поскольку доступ к очереди может быть получен только через переменную типа UserQueue, в именах членов класса упоминание слова "queue" вообще не нужно — все будет понятно из контекста. Так, двойное упоминание очереди

```
 ? queue. queueCapacity
```

избыточно.Класс лучше описать так:

```
     class UserQueue {
 int nitems, front, capacity; public- int nusers( ) { . . . }
```

поскольку выражения вроде

```
    queue.capacity+++;
n=queue.nusers()
```

вполне ясны и понятны. Последняя версия тоже еще не идеальна: термины "items" и "users" обозначают одно и то же, так что для одного понятия следовало бы использовать только один термин.

Используйте активные имена для функций. Имя функции должно базироваться на активном глаголе (в действительном залоге), за которым может следовать существительное:

```
    now = date.getTime();
putchar( "\n-);
```

Функции, которые возвращают логическое значение (истина или ложь — t rue или false), нужно называть так, чтобы их смысл не вызывал сомнений. Так, из вызова

```
? if (checkoctal(c)) ...
```

непонятно, когда будет возвращаться t rue, а когда false, а вот вызов

```
if (isoctal(c)) ...
```

не оставляет сомнений в том, что результат будет истиной, если аргумент — восьмеричное число, и ложью — в противном случае.

Будьте точны. Как мы уже решили, имя не только обозначает объект, но и предоставляет читателю некоторую информацию о нем. Неверно подобранное имя может стать причиной совершенно таинственных ошибок.

Один из нас написал и не один год распространял макрос, называемый isoctal, имеющий некорректную реализацию:

```
«define isoctal(c) ((c) >= '0' && (c) <= '8')?
```

вместо правильного

```
«define isoctal(c) ((c)>= '0' && (c) <= '7')
```

В этом случае имя было правильным, а реализация ошибочной; правильное имя облегчило раскрытие ошибки.

В следующем примере имя и код находятся в полнейшем противоречии:

```
? public boolean inTable(Object obj) {
? int j = this.getIndex(ob]);
? return (j == nTable);
? }
```

Функция getIndex возвращает значение от 0 до nTable-1 в случае, если объект найден, и nTable, если объект не найден. Таким образом, возвращаемое логическое значение будет прямо противоположно тому, что подразумевается в имени функции. Во время написания кода это, может быть, не страшно, но, если программа будет дорабатываться позднее, да еще другим программистом, подобное неправильное имя наверняка станет причиной затруднений.

Упражнение 1-1

Составьте комментарий, объясняющий принцип выбора имен и значений в следующем примере:

```
? «define TRUE O ? «define FALSE 1
? if ((ch = getcharO) == EOF)
? not_eof = FALSE;
```

Упражнение 1-2

Улучшите функцию:

```
    int smaller(char *s, char *t) { if (strcmp(s, t) < 1)
 return 1; else
 return 0;
```

Упражнение 1-3

Прочитайте вслух следующее выражение:

```
    if ((falloc(SMRHSHSCRTCH,
S._IFEXT|0644, MAXRODDHSH)) < 0)
```

Выражения

Итак, имена надо подбирать так, чтобы максимально облегчить жизнь читателю; точно так же и выражения следует писать так, чтобы их смысл был предельно ясен. Пишите самый ясный код, какой только возможен. Вставляйте пробелы вокруг операторов для логической группировки выражений; вообще, форматируйте выражения так, чтобы сделать их наиболее удобочитаемыми. Идея тривиальна, но очень полезна: совсем как мысль о том, что чем аккуратнее убран ваш рабочий стол, тем проще на нем найти необходимую вещь. Правда, в отличие от рабочего стола в вашей программе могут копаться и посторонние — и им тоже должно быть удобно.

Форматируйте код, подчеркивая его структуру. Форматирование кода логично выбранными отступами — самый простой способ добиться того, чтобы структура программы стала самоочевидна. Приведенный фрагмент плохо отформатирован:

```
     ? for(n++;n<100;field[n++]='\(O');
 ? *i = \O'; return( '\n);
```

Проведя простейшее форматирование, мы несколько улучшим его:

```
    ? for (n++; n < 100; field[n++] = '\0')
? ;
 ? **i = \O'; ? return( '\n');
```

Еще лучше — перенести оператор присваивания в тело цикла и выделить инкремент (приращение) счетчика в отдельный оператор, тогда цикл примет более традиционный вид и, следовательно, можно будет быстрее догадаться о его смысле.

```
    for (n++; n < 100; n++)
field[n] = \O'; *i = \O';
return '\n';
```

Используйте естественную форму выражений. Записывайте выражения в том виде, в котором вы произносили бы их вслух. Условные выражения, содержащие отрицания, всегда трудно воспринять:

```
if (!(block_id < actblks) !(blocked >= unblocks))
```

Здесь каждая проверка используется с отрицанием, однако необходимости в этом нет никакой. Достаточно изменить знаки сравнения, и можно обойтись без отрицаний:

```
if ((blocked >= actblks) || (blockj.d < unblocks))
```

Теперь код читается вполне естественно.

Используйте скобки для устранения неясностей. Скобки подчеркивают смысловую группировку кода, и их можно использовать для упрощения понимания его структуры даже тогда, когда с точки зрения грамматики в их применении нет необходимости. Так, в предыдущем примере внутренние скобки не нужны, однако их применение как минимум ничему не мешает. Бывалый программист мог бы и опустить их, поскольку операторы сравнения (< <= == ! = >= >) имеют более низкий приоритет, чем логические операторы (&& и | |).

Все же, когда используются операторы с различным приоритетом, лучше применять скобки. В языке С и ему подобных существует ряд проблем с последовательностью выполнения операций, так что ошибку сделать совсем не сложно. Поскольку логические операции имеют более высокий приоритет, чем присваивание, употребление скобок обязательно для большинства выражений, использующих оба типа операций:

```
while ((c = getchar()) != EOF)

...
```

Побитовые операции & и | имеют более низкий приоритет, чем "обычные" операции сравнения типа ==, так что, несмотря на, казалось бы, очевидный порядок выполнения, выражение

```
if (x&MASK == BITS)
```

на самом деле выполнится как

```
? if (x & (MASK==BITS))
?
```

а это, очевидно, вовсе не то, что подразумевал программист. Поскольку используется комбинация операций сравнения и побитовых операций, скобки просто необходимы:

```
if ((x&MASK) == BITS)
```

Даже если скобки в выражении не обязательны, они могут помочь сгруппировать операции так, чтобы фрагмент кода стал понятен с первого взгляда. В следующем

примере проверки високосности года по его номеру в использовании скобок нет жесткой необходимости:

```
? leap_year = y % 4 == 0 && y % 100 ! = 0 | | y % 400 == 0;
```

но с ними фрагмент станет гораздо проще для понимания:

```
leap_year = ((y%4 == 0) && (y%100 != 0)) | (y%400 ==0);
```

Мы убрали здесь некоторые пробелы: группировка операций с самым высоким приоритетом помогает читателю быстрее разобраться в структуре программы.

Разбивайте сложные выражения. Языки C, C++ и Java имеют богатый и разнообразный синтаксис, поэтому многие увлекаются втискиванием всего подряд в одну конструкцию. Приведенное выражение весьма компактно, однако в нем содержится слишком много операторов:

```
*x += (*xp=(2*k < (n-m) ? c [k+1] : d[k--]));
```

Если разбить это выражение на несколько, оно станет гораздо более удобочитаемым:

```
if (2*k < n-m)
  *xp = c[k+1j;
else
  *xp = d[k--];
*x += *xp;
```

Будьте проще. Безудержная энергия программистов иногда направляется на то, чтобы написать наиболее краткий код или достичь результата самым хитрым и красивым способом. Иногда, впрочем, эти таланты тратятся впустую: настоящая задача в том, чтобы написать код попроще, а не похлеще. Вот, к примеру, что делает это замысловатое выражение?

```
? subkey = subkey » (bitoff - ((bitoff » 3) « 3));
```

Самое внутреннее выражение сдвигает bitoff на три бита вправо. Результат сдвигается обратно влево, при этом три сдвинутых бита замещаются нулями. Затем результат вычитается из первоначального значения, оставляя три младших бита bitoff, которые используются для сдвига subkey вправо.

Таким образом, приведенная конструкция эквивалентна

```
subkey = subkey » (bitoff & 0x7);
```

Над первой версией приходится гадать некоторое время, чтобы понять, что там происходит; со второй же все просто и ясно. Опытные программисты записали бы ее еще короче:

```
subkey »= bitoff & 0x7;
```

Некоторые конструкции кажутся специально созданными для того, чтобы сбить читателя с толку. Например, применение оператора ? : может привести к появлению абсолютно загадочного кода:

```
? child=(!LC&&!RC)?0:(!LC?RC:LC);
```

Согласитесь, что, только разобрав все возможные варианты сравнений, можно догадаться, что же делает этот фрагмент. Приведенная ниже конструкция длиннее, но значительно проще, потому что возможные варианты видны сразу:

```
f (LC == 0 && RC == 0)
child = 0; else if (LC == 0)
child = RC; else
child = LC;
```

Операция ? : хороша только для коротких и простых выражений-, где она может заменить четыре строки if-else одной, как в следующем примере:

```
max = (a > b) ? a : b;
```

или даже в таком случае:

```
printf("The list has %d item%s\n", n, n==1 ? ""
: "s");
```

однако подобная замена все же не является распространенной.

Ясность и краткость — не одно и то же. Часто более ясный код будет и более коротким (как в примере со сдвигом битов), однако он может быть и, наоборот, более длинным (как в примере с if-else). Главным критерием выбора должна служить простота восприятия.

Будьте осторожны с побочными эффектами. Операции типа ++ имеют побочные эффекты: они не только возвращают значение, но еще и изменяют операнд. Эти побочные эффекты иногда могут быть весьма удобны, но они же могут вызвать трудности из-за того, что получение значения и обновление переменной могут происходить не тогда, когда ожидается. В C и C++ порядок выполнения этих побочных эффектов вообще не определен, поэтому нижеприведенное множественное присваивание, скорее всего, будет выдавать неправильный ответ:

```
? str[i++] = str[i++] = ' ';
```

Смысл выражения — записать пробел в следующие две позиции строки str. Однако в зависимости от того, когда будет обновляться i, позиция в st r может быть пропущена, и в результате переменная 1 будет увеличена только на 1. Разбейте выражение на два:

```
str[i++] = ' ';
str[i++] ="-";
```

Даже если в вы ражении содержится только один инкремент, результат может быть неоднозначным:

```
    ? array[i++] = i;
```

Если изначально i имеет значение 3, то элемент массива может принять значение 3 либо 4.

Побочные эффекты есть не только у инкремента и декремента; ввод-вывод — еще один потенциальный источник возникновения закулисных действий. В следующем примере осуществляется попытка прочитать два взаимосвязанных числа из стандартного ввода:

```
    scanf ("' &yr, &profit[yr]);
```

Выражение неверно, поскольку одна его часть изменяет уг, а другая использует ее. Значение prof it[yr] будет правильным только в том случае, если новое значение уг будет таким же, как и старое. Вы, наверное, думаете, что все дело в порядке вычисления аргументов, однако в действительности проблема здесь в том, что все аргументы scanf вычисляются до того, как эта операция вызывается на выполнение, так что &profit[yr] всегда будет вычисляться с использованием старого значения у г. Проблемы подобного рода могут возникнуть в любом языке программирования. Для исправления достаточно опять же разбить выражение на две части:

```
     scanf("%d",                                                    &yr);
scanf("%d", &profit[yr]);
```

Будьте осторожны с любым выражением, вызывающим побочные эффекты.

Упражнение 1-4

Улучшите каждый из приведенных фрагментов:

```
     if ( !(c == 'y' || c == 'V) ) return;
 length = (length < BUFSIZE) ? length : BUFSIZE; ,
flag = flag ? 0 : 1;
quote = (*line == "") ?* 1 : 0;
if (val & 1)
 t else
 bit=0
```

Упражнение 1-5

Найдите ошибку в приведенном фрагменте:

```
    ? i'nt read(int *ip) { ? scanf("%d", ip);
? return *ip;
? }
 ? insert(&graph[vert], read(.&val), read(&ch));
```

Упражнение 1-6

Перечислите все возможные варианты, которые выдаст приведенное выражение в зависимости от порядка вычислений:

```
    ? n=1
? printf(."%d %d\n", n++, n++);
```

Попробуйте пропустить это выражение через разные компиляторы и посмотрите, что получается на практике.

Стилевое единство и идиомы

Чем логичнее и последовательнее оформлена программа, тем она лучше. Если форматирование кода меняется непредсказуемым образом: цикл просматривает массив то по возрастанию, то по убыванию, строки копируются то с помощью strcpy, то с помощью цикла for — все перечисленные вариации сбивают читателя с толку, и ему трудно понять, что в действительности происходит. Но если одни и те же вычисления всегда выполняются одинаково, то каждое отклонение от обычного стиля будет указывать на действительное различие, заслуживающее быть отмеченным.

Будьте последовательны в применении отступов и фигурных скобок.
Отступы помогают воспринять структуру кода, но какой стиль расположения лучше? Следует располагать открывающую фигурную скобку на той же строке, что if, или на следующей? Программисты ведут нескончаемые споры о наилучшем расположении текста кода, однако выбор конкретного стиля гораздо менее важен, чем логичность и последовательность его применения во всем приложении. Выберите себе раз и навсегда один стиль — лучше всего, конечно, наш — и используйте его всегда, не тратьте времени на споры.

Стоит ли вставлять фигурные скобки, когда необходимости в них нет? Как и обычные скобки, дополнительные фигурные скобки могут разрешить неясные моменты и облегчить понимание структуры кода. Многие опытные программисты во имя все того же постоянства всегда заключают в фигурные скобки тела циклов и условных операторов. Однако если тело состоит лишь из одной строки, то скобки явно не нужны, и мы бы посоветовали их опустить. Впрочем, не забудьте вставить их в тех случаях, когда они помогут разрешить неясности с "висящим else", — подобная неопределенность хорошо иллюстрируется следующим примером:

```
    if (month == FEB) { if (year%4 == 0) if (day > 29)
legal = FALSE; else
 if (day > 28)
 legal = FALSE;
```

В данном фрагменте выравнивание выполнено неправильно, поскольку ia самом деле else относится к строке

```
     if (day > 29)
```

и весь код работает неверно. Когда один if следует сразу за другим всегда используйте фигурные скобки:

```
    ? if (month == FEB) {
? if (year%4 == 0) {
 ?if (day > 29)
? legal = FALSE;
? } else {
? if (day > 28)
 ? legal = FA*LSE;
```

Применение средств редактирования с поддержкой синтаксиса уменьшает вероятность возникновения подобных ошибок.

В рассматриваемом примере даже после исправления отмеченной ошибки код трудно понять. В нем будет проще разобраться, если мы заведем переменную для хранения количества дней в феврале:

```
    ? if (month == FEB) { ? int nday;
?
 ? nday = 28;
? if (year%4 == 0) ? nday = 29;
? if (day > nday) 9 legal = FALSE;
 ?}
```

Код все еще неверен: 2000 год является високосным, а 1900-й и 2100-й — не високосные. Тем не менее такая структура уже значительно проще доводится до безупречной.

Кстати, если вы работаете над программой, которую писал кто-то другой, сохраняйте ее оригинальный стиль. При внесении изменений не применяйте свой стиль, даже если он вам больше нравится. Постоянство в форматировании программы гораздо важнее, чем ваше собственное, поскольку только оно сможет облегчить жизнь тем, кому придется вникать в способы функционирования этой программы.

Используйте идиомы для единства стиля. В языках программирования, как и в обычных языках, существуют свои идиомы — это устоявшиеся приемы, используемые опытными программистами для написания типичных фрагментов кода. Одна из центральных проблем, возникающих при изучении языка программирования, — необходимость привыкнуть к его идиомам.

Одной из наиболее типичных идиом является форма написания цикла. Рассмотрим код на C, C++ или Java для просмотра n элементов массива, например для их инициализации. Можно написать этот код так:

```
    ? 1 = 0;
? while (1 <= n-1)
? array[i++] =1.0;
```

или так:

```
    ? for (1=0; 1 < n; )
? array[1++] = 1.0;
```

или даже так:

```
    for (i = n; --1 >~ 0; )
array[i] = 1.0;
```

Все три способа, в принципе, правильны, однако устоявшаяся, идиоматическая форма выглядит так:

```
    for (1 = 0; i < n; i++)
array[i] = 1.0;
```

Выбор именно этой формы не случаен. При такой записи обходятся все n элементов массива, пронумерованные от 0 до n-1. Все управление циклом находится непосредственно в его заголовке; обход происходит в возрастающем порядке; для обновления переменной счетчика цикла используется типичная операция инкремента (++). После выхода индекс цикла имеет известное нам значение как раз за последним элементом массива. Те, для кого этот язык как родной, все понимают без пояснений и воспроизводят конструкцию не задумываясь.

В C++ и Java стандартный вариант включает еще и объявление переменной цикла:

```
    for (int 1=0; i < n; i++)
 array[i] = 1.0;
```

Вот как выглядит стандартный цикл для обхода списка в C:

```
     for (p = list; p != NULL; p = p->next)
     .......
```

И опять все управляющие элементы цикла находятся непосредственно в выражении for.

Для бесконечного цикла мы предпочитаем конструкцию

```
    for (;;)
```

однако популярна и конструкция

```
    while                                                                    (1)
 .......
```

Не используйте других форм, кроме двух приведенных.

Использование отступов также должно быть стандартным, можно сказать — идиоматичным. Вот такое необычное вертикальное расположение управляющих элементов цикла ухудшает читабельность кода, поскольку выглядит как три отдельных выражения, а не как единый заголовок цикла:

```
    for (
 ap = arr;
ap < arr + 128;
 *ap++ = O
  )
 {
  ;
  }
```

Стандартную форму записи цикла воспринять гораздо проще:

```
     for (ap = arr; ap < arr+128; ap++)
 *ap = 0;
```

Небрежно выбранное расположение отступов растянет код на несколько экранов или страниц, что также не улучшает его восприятие.

Еще.один стандартный прием — вставлять присваивание внутрь условия цикла:

```
    while ((c = getchar()) != EOF)
  putchar(c);
```

Выражение do-while используется гораздо реже, чем for и while, поскольку при его применении тело цикла исполняется как минимум один раз вне зависимости от условий, которые проверяются не в начале цикла, а в конце. Во многих случаях это становится причиной появления ошибки, ждущей своего часа, как, например, в следующем варианте цикла для getchar:

```
    do {
  c = getchar();
 putchar(c); } while (c != EOF);
```

В этом случае в переменную будет записано мусорное значение, поскольку проверка производится уже после вызова putchar. Цикл do-while стоит применять только в тех случаях, когда тело цикла обязательно должно быть выполнено хотя бы один раз; некоторые примеры подобных ситуаций мы рассмотрим несколько позже.

Одно из преимуществ последовательного применения идиом состоит в том, что любой нестандартный цикл, потенциальный источник ошибок, сразу же привлечет внимание. Например:

```
    int i, *iArray, nmemb;'
  lArray = malloc(nmemb * sizeof(int));
 for (i = 0; i <= nmemb; i++) iArray[i] = i;
```

Место в памяти выделяется для элементов в количестве nmemb — от iArray[0] до iArray[nmemb-1], но, поскольку в условии цикла проверка производится на "меньше или равно" (<=), цикл вылезет за границу массива и испортит новой записью то, что хранится за пределами выделенной памяти. К сожалению, ошибки подобного типа нередко можно обнаружить только после того, как данные уже разрушены.

В C и C++ есть также идиомы для выделения места под строки и для работы с ними, а код, который их не использует, зачастую таит в себе ошибку:

```
    char *p, buf[256];
 gets(buf);
 p = malloc(strlen(buf));
  strcpy(p, but);
```

Никогда не следует употреблять gets, поскольку не существует способа ограничить размер вводимой информации. Это ведет к проблемам, связанным с безопасностью, к которым мы вернемся в главе 6; там мы увидим, что всегда лучше использовать fgets. Однако существует и еще одна проблема: функция strlen вычисляет размер строки, не учитывая завершающего строку символа ' \0', а функция st rcpy его копирует. Таким образом, выделяется недостаточно места, и st rcpy пишет за пределами выделенной памяти. Стандартной формой является следующая:

```
     p = malloc(strlen(buf)+1);
  strcpy(p, buf);
```

или

```
    p = new char[strlen(buf)+1];
strcpy(p, buf);
```

в C++. Таким образом, если вы не видите +1, то стоит проверить все еще раз.

В Java такой проблемы нет, поскольку строки там не представляются в виде массивов, оканчивающихся на 0. Кроме того, индексы массивов проверяются, так что в Java выйти за границы массива невозможно.

В большинстве сред C и C++ предусмотрена библиотечная функция strdup, которая создает копию строки, используя для этого malloc и st rcpy, что гарантирует от обсуждавшейся чуть выше ошибки. К сожалению, st rdup не входит в стандарт ANSI C.

Кстати говоря, ни в первой, ни в окончательной версии не проверяется значение, возвращаемое функцией malloc. Мы опустили эту проверку для того, чтобы сфокусировать ваше внимание на теме данного раздела, однако в настоящей программе значения, возвращаемые функциями malloc, realloc, strdup, а также другими функциями, выделяющими память, должны в обязательном порядке проверяться.

Используйте цепочки else-if для многовариантных ветвлений. Стандартной формой записи многовариантного ветвления является последовательность операторов if. . .else if...else:

```
    if (условие 1)
 выражение 1 else if (условие 2)
выражение 2
 else if (условие п)
выражение п else
выражение по умолчанию
```

Условия читаются сверху вниз; по достижении первого условия, которое оказалось верным, выполняется следующее за ним выражение, после чего оставшаяся часть конструкции пропускается. Под выражением мы, естественно, понимаем здесь либо одно выражение, либо группу выражений, заключенную в фигурные скобки. Последнее else обрабатывает ситуацию "по умолчанию" — когда ни одна из перечисленных альтернатив не была выбрана. Это замыкающее else может быть опущено, если не предусматривается никаких действий "по умолчанию", однако и в таком случае полезно оставить эту часть с сообщением об ошибке, чтобы отловить условия, которых "не может быть никогда".

Все операторы else стоит выровнять по вертикали, вместо того чтобы устанавливать каждое else на одном уровне с соответствующим ему if. Вертикальное выравнивание означает, что проверки производятся последовательно; кроме того, предотвращается выползание кода за правый край страницы.

Последовательность вложенных выражений if —. предвестник трудно читаемого кода, если не заведомых ошибок:

```
    ? if (argc == 3)
 ? if ((fin = fopen(argv[1], "r")) \= NULL)
? if ((tout = fopen(argv[2], >")) != NULL) {
? while ((c = getc(fin)) != EOF)
```

```
 ? putc(c, fout);
  fclose(fin); fclose(fout); }
 else
 printf("Не открыть выходной файл %s\n", argv[2]);
  else
  printf("Не открыть входной файл %s\n", argv[1]);
```

Последовательность условных операторов заставляет нас напрягаться, запоминая, какие тесты в каком порядке следуют, с тем чтобы в нужной точке вставить соответствующее событие (если мы еще в состоянии его вспомнить). Там, где должно быть произведено хотя бы одно действие, лучше использовать else if. Изменение порядка, в котором производятся проверки, ведет к тому, что код становится более понятным, кроме того, мы избавляемся от утечки ресурса, которая присутствовала в первой версии (файлы остались незакрытыми):

```
    if (argc != 3)
 printf ("Использование: ср входной_файл выходной_файл \п");
 else if ((fin = fopen(argv[1], "r")) == NULL)
  printf("Не открыть входной файл %s\n", argv[1]); else if ((fout
      = fopen(argv[2], "w")) == NULL) {
  printf("Не открыть выходной файл %s\n", argv[2]);
  fclose(fin); } else {
 while ((c = getc(fin)) != EOF) putc(c, fout);
  fclose(fin);
 fclose(fout);
```

Мы производим проверки до тех пор, пока не находим первое выполненное условие, совершаем соответствующее действие и опускаем все последующие проверки. Правило тут такое: каждое действие должно быть расположено как можно ближе к той проверке, с которой связано. Другими словами, каждый раз, произведя проверку, совершите что-нибудь.

Попытки повторного использования уже написанных блоков кода ведут к появлению программ, похожих на затянутые узлы:

```
    switch (c) {
 case '-': sign = -1;
  case '+': c = getchar();
 ? case '.' ?
 default:
  }
```

Здесь используется замысловатая последовательность перескоков в выражении-переключателе, а все для того, чтобы избежать дублирования одной строки кода! В стандартной, идиоматической форме каждый вариант должен оканчиваться выходом (break), с редкими и обязательно откомментированными исключениями. Традиционную структуру проще воспринимать, хотя она и получается несколько длиннее:

```
    ? switch
    (c) {
  ? case ' - ' : ? sign = -1;
 ? /* перескок вниз */
  , ? case ' + ',: ? c = getchar();
  ? break;
  ? case ' . ' : ? break;
```

```
? default: ? if (lisdigit(c))
? return 0;
? break;
```

Некоторое удлинение кода с лихвой окупается увеличением ясности. В данной конкретной ситуации, однако, применение последовательности выражений else-if будет даже более понятным:

```
    if (c == '-') {
sign = -1;
 c = getchar(); } else if (c == '+' ) {'
c = getchar(); } else if (c != '. ' && ! isdigit(c)) {
return 0;
```

В этом примере фигурные скобки вокруг блока из одной строки подчеркивают, что структура выражения параллельна.

Использование перескоков оправдано в случае, когда несколько условий имеют одинаковый код. Тогда их размещают таким образом:

```
    case '0' :
case ' 1' :
case '2' :
break;
```

При такой записи комментарии не нужны.

Упражнение 1-7

Перепишите выражения C/C++ более понятным образом:

```
    if (istty(stdin)) ;
else if (istty(stdout)) ;
else if (istty(stderr)) ; else return(O);
 if (retval != SUCCESS) {
return (retval);
 }

    /* Все идет хорошо! */
 return SUCCESS;
for (k = 0; k++ < 5; x += dx) scanf("%lf"! &dx);
```

Упражнение 1-8

Найдите ошибки в этом фрагменте кода на языке Java и исправьте их, переписав цикл в стандартной форме:

```
    int count = 0;
while (count < total) {
 count++;
if (this.getNaine(count) == nametable. userName()) { return (true);
```

Макрофункции

В среде программистов, давно пишущих на С, существует тенденция писать макросы вместо функций для очень коротких вычислений, которые будут часто вызываться: операции ввода-вывода, такие как getchar, и проверки символов вроде isdigit — это, так сказать, официально утвержденные примеры. Причина — в производительности: у макросов нет "накладных расходов", которые свойственны вызовам функций.

На самом деле этот аргумент был не слишком убедительным уже в те времена, когда С только появился, — в эпоху медленных машин и "дорогих" вызовов функций; теперь же он просто нелеп. Для современных машин и компиляторов недостатки макрофункций перевешивают их достоинства.

Избегайте макрофункций. В C++ встраиваемые (inline) функции делают использование макрофункций ненужным; в Java макросов вообще не существует. В С они больше проблем создают, чем решают.

Одна из наиболее серьезных проблем, связанных с макрофункциями: параметр, который появляется в определении более одного раза, может быть вычислен также более одного раза; если же аргумент вызова включает в себя выражение с побочными эффектами, то результатом будет трудно отлавливаемая ошибка. В приведенном коде сделана попытка самостоятельно реализовать одну из проверок символов из <ctype . h>:

```
? «define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

Обратите внимание на то, что параметр с дважды появляется в теле макроса. Если же наш макрос is up ре г вызывается в контексте вроде такого:

```
? while (isupper(c = getchar()))
?
```

то каждый раз, когда вводимый символ будет больше или равен А, он будет пропущен, и для сравнения с Z будет считан еще один символ. Стандарт С написан таким образом, чтобы функции, аналогичные isupper, можно было реализовать как макросы, но только если они гарантируют, что аргумент будет вычисляться лишь единожды, так что приведенная выше реализация не отвечает требованиям стандарта.

Всегда лучше использовать уже имеющиеся функции ctype, чем реализовывать их самостоятельно; кроме того, функции с побочными эффектами, типа getcha г, лучше не применять внутри составных выражений. Если разбить условие цикла на два выражения, то оно будет более понятно для читателя и, кроме того, даст возможность четко отследить конец файла:

```
while ((c = getcharO) != EOF && isupper(c))
```

Иногда многократные вычисления не несут в себе прямых ошибок, но снижают производительность. Рассмотрим такой пример:

```
    ? «define ROUND_TO_INT(x) ((int.) ((x) + (( (x)>0)?0. 5: -0. 5)))
?
? size = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

Вычисление квадратного корня будет производиться в два раза чаще, чем требуется (он передается как аргумент, который дважды участвует в вычислении). Даже при задании простого аргумента сложное выражение вроде ROUND_TO__INT преобразуется во множество машинных команд, которые лучше хранить в одной функции, вызываемой при необходимости. Обращения к макросу увеличивают размер скомпилированной программы. (Встраиваемые (inline) функции в C++ имеют тот же недостаток.)

Заключайте тело макроса и аргументы в скобки. Если вы все же решили использовать макрофункции, будьте с ними осторожны. Макрос работает за счет простой подстановки текста: параметры в описании заменяются аргументами вызова, и результат (текст!) замещает собой текст вызова. В этом состоит важное отличие макросов от функций, делающее макросы столь ненадежными. Так, выражение

```
    1 / square(x)
```

будет работать отлично, если square — это функция, однако если это макрос вроде следующего:

```
    ? «define square(x) (x) * (x)
```

то выражение будет преобразовано в ошибочное:

```
    ? 1 / (x) * (x)
```

Этот макрос надо переписать так:

```
     «define square(x) ((x) * (x))
```

Скобки здесь необходимы, однако даже грамотная расстановка скобок не спасет макрос от его главной беды — многократного вычисления аргументов. Если операция настолько сложна или настолько часто повторяется, что ее стоит вынести в отдельный блок, используйте для этого функцию.

В C++ встраиваемые (inline) функции позволяют избежать синтаксических проблем, сохраняя при этом высокую производительность, присущую макросам. Они хорошо подходят для коротких функций, которые получают или устанавливают только одно значение.

Упражнение 1-9

Определите все проблемы, связанные с приведенным описанием макроса:

```
    ? «define ISDIGIT(c) ((c >= '0') && (c <= '9')) ?
1 :   O
```

Загадочные числа

Загадочные числа — это константы, размеры массивов, позиции символов и другие числовые значения, появляющиеся в программе непосредственно, как "буквальные константы".

Давайте имена загадочным числам. В принципе нужно считать, что любое встречающееся в программе число, отличное от 0 и 1, является загадочным и должно получить собственное имя. Просто "сырые" числа в тексте программы не дают представления об их происхождении и назначении, затрудняя понимание и изменение программы. Вот отрывок из программы, которая печатает гистограмму частот букв на терминале с разрешением 24 X 80. Этот отрывок неоправданно запутан из-за целого сонма непонятных чисел:

```
    fac = lim / 20; /* установка масштаба */
if (fac < 1)
fac = 1;
 /* генерация гистограммы */ for (i = 0, col = 0; i < 27; i++, j++)
    {
 col += 3;
k = 21 - (let[i] / fac);
 star = (let[i] == 0) ? ' ' : '*';
for (j = k; j < 22; j++)
draw(j, col, star); } draw(23, 2, ' '); /* разметка оси x */
```

Наряду с другими в коде присутствуют числа 20, 21, 22, 23 и 27. Они как-то тесно связаны... или нет? На самом деле есть только три критических числа, существенных для программы: 24 — число строк экрана; 80 — число столбцов экрана и, наконец, 26 — количество букв в английском алфавите. Однако, как мы видим, ни одно из этих чиеел в коде не встречается, отчего числа, используемые в коде, становятся еще более загадочными.

Присвоив имена числам, имеющим принципиальное значение, мы облегчим понимание кода. Например, станет понятно, что число 3 берется из арифметического выражения (80-1)/26, а массив let должен иметь 26 элементов, а не 27 (иначе возможна ошибка его переполнения на единицу — из-за того, что экранные координаты индексируются, начиная с 1). Сделав еще пару усовершенствований, мы придем к следующему результату:

```
    enum {
MINROW =1, /* верхняя граница */
MINCOL =1, /* левая граница */
MAXROW = 24, /* нижняя граница (<=) */
MAXCOL = 80, /* правая граница (<=) */
LABELROW =1, /* позиция подписей оси */
NLET = 26, f /* количество букв алфавита */
HEIGHT = MAXROW - 4, /* высота столбиков */•
WIDTH = (MAXCOL-1)/NLET /* ширина столбиков */
 };
fac = (lim + HEIGHT-1) / HEIGHT; /* установка масштаба */
if (fac <
    1)
 fac = 1; for (i = 0; i < NLET; i++) { /* генерация гистограммы */
if (let[i] == 0) continue;
```

```
for (j = HEIGHT - let[i]/fac; j < HEIGHT; j++)
 draw(j+1 + LABELROW, (i+1)*WIDTH, '*'•); }
draw(MAXROW-1, MINCOL+1, ' '); /* разметка оси x */
for (i = 'A'; i <=
     'Z'; i++)
printf("%c ", i);
```

Теперь стало гораздо понятнее, что же делает основной цикл: в нем используется стандартный идиоматический цикл от 0 до NLET-1, то есть по всем элементам данных. Вызовы функции d raw также стали более понятны — названия вроде MAXROW и MINCOL дают четкое представление об аргументе. Главное же — программу теперь можно без труда адаптировать к другому разрешению экрана или другому алфавиту: с чисел и с кода снята завеса таинственности.

Определяйте числа как константы, а не как макросы. Программисты, пишущие на C, традиционно использовали для определения загадочных чисел директиву #def те. Однако препроцессор C — мощный, но несколько туповатый инструмент, а макросы — вообще довольно опасная вещь, поскольку они изменяют лексическую структуру программы. Пусть лучше язык делает свойственную ему работу. В C и C++ целые (integer) константы можно определять с помощью выражения en urn (которое мы и использовали в предыдущем примере). В C++ любые константы можно определять с помощью ключевого слова const:

```
const int MAXROW = 24, MAXCOL = 80;
```

В Java для этого служит слово final:

```
static final int MAXROW = 24, MAXCOL = 80;
```

В C также можно определять значения с помощью ключевого слова const, но эти значения нельзя использовать как границы массива, так что зачастую придется прибегать все к тому же enum.

Используйте символьные, а не целые константы. Для проверки свойств символов должны использоваться функции из <ctype. h> или их эквиваленты. Если проверку организовать так:

```
? if (c >= 65 && c <= 90)
?....
```

то ее результат будет всецело зависеть от представления символов на конкретной машине. Лучше использовать

```
if (c >= 'A' c <= 'Z')
?...
```

но и это может не принести желаемого результата, если буквы в имеющейся кодировке идут не по порядку или если в алфавите есть и другие буквы. Лучшее решение — привлечь на помощь библиотеку:

```
if (isupper(c))
```

в C и C++ или

```
if (Character. isllpperCase(c))
```

в Java.

Сходный вопрос — использование в программе числа 0. Оно используется очень часто и в различных контекстах. Компилятор преобразует этр число в соответствующий тип, однако читателю гораздо проще понять роль каждого конкретного 0, если тип этого числа каждый раз обозначен явным образом. Так, например, стоит использовать (void *)0 или NULL для обозначения нулевого указателя в C, а ' \0' вместо просто 0 — для обозначения нулевого байта в конце строки. Другими словами, не пишите

```
? str = 0; ? nameti] = 0; ? x = 0;
```

а пишите

```
str = NULL; name[i] = '\0'; x = 0.0;
```

Мы рекомендуем использовать различные явные константы, оставив О для простого целого нуля, — такие константы обозначат цель использования данного значения. Надо отметить, правда, что в C++ для обозначения нулевого указателя принято использовать все же 0, а не NULL. Лучше всего проблема решена в Java — там ключевое слово null определено как ссылка на объект, которая ни к чему не относится.

Используйте средства языка для определения размера объекта. Не используйте явно заданного размера ни для каких типов данных — так, например, используйте sizeof (int) вместо прямого указания числа 2,4 и т.п. По сходным причинам лучше использовать sizeof(array[OJ) вместо sizeof (int) — меньше придется исправлять при изменении типа массива.

Использование оператора sizeof избавит вас от необходимости выдумывать имена для чисел, обозначающих размер массива. Например, если написать

```
char buf[1024];
fgets(buf, sizeof(buf), stdin);
```

то размер буфера хоть и станет "загадочным числом", от которого мы предостерегали ранее, но зато оно появится только один раз — непосредственно в описании. Может быть, и не стоит прилагать слишком большие усилия, чтобы придумать имя для размера локального массива, но определенно стоит постараться и написать код, который не нужно переписывать при изменении размера или типа: У массивов в Java есть поле length, которое содержит количество элементов:

```
char buf[] = new char[1024];
for (int 1=0; i < but.length; i++)
.....
```

В C и C++ нет эквивалента этому полю, но для массива (не указателя), описание которого является видимым, количество элементов можно вычислить с помощью следующего макроса:

```
    «define NELEMS(array) (sizeof(array)
 / sizeof(array[0]))
double dbuf[100];
for (i = 0; i < NELEMS(dbuf); i++)
```

Здесь опять-таки размер массива задается лишь в одном месте, и при его изменении весь остальной код менять не придется.

В данном макросе нет проблем с многократным вычислением аргумента, поскольку в нем нет никаких побочных эффектов, и на самом деле все вычисление происходит во время компиляции. Это пример грамотного использования макроса — здесь он делает то, чего не может сделать функция: вычисляет размер массива исходя из его описания.

Упражнение 1-10

Как бы вы переписали приведенные определения, чтобы уменьшить число потенциальных ошибок?

```
    ?define FT2METER
?define METER2FT
?define MI2FT
?define MI2KM
?define SQMI2SQKM 2.589988
```

Комментарии

Комментарии должны помогать читать программу. Они не помогут, повторяя то, что и так понятно из кода или противореча коду или отвлекая читателя от сути типографскими ухищрениями. Лучший комментарий помогает понимать программу, кратко отмечая наиболее значимые детали или предоставляя укрупненную картину происходящего.

Не пишите об очевидном. Комментарии не должны содержать самоочевидной информации типа того, что оператор i++ увеличивает i. Ниже приведены некоторые из наших чемпионов бессмысленности:

```
    /*
 по умолчанию
 */
default: break;
/* возвращаем SUCCESS */ return SUCCESS;
? zerocount++; /* Увеличиваем счетчик нулевых элементов */
? /* Устанавливаем "total" в "number_received" */
    ? node->total = node->number_received;
```

Все эти комментарии надо удалить, они лишь мешают.

В комментариях должна содержаться информация, не вытекающая из кода, или же информация, относящаяся к большому фрагменту кода и собранная в одно место. Когда в коде происходит что-то трудноуловимое, комментарий должен уточнять происходящее, но если все действия и так очевидны, описывать их еще раз в словах просто бессмысленно:

```
    while ((c = getchar()) != EOF && isspace(c))
 ; /* пропуск пробелов */
 /* конец файла */
 if (c == EOF)
type = endoffile; else if (c == ' (')
 type = leftparen; else if (c == ')')
type = rightparen; else if (c == '; ')
 type = semicolon; else if (is_op(c))
type = operator; else if (isdigit(c))
  /* открывающая скобка */
 /* закрывающая скобка */
/* точка с запятой */
 /* оператор */
 /* цифра */
```

Эти комментарии также стоит удалить, поскольку грамотно подобранные имена уже содержат всю необходимую информацию.

Комментируйте функции и глобальные данные. Конечно же, комментарии могут и должны быть полезны. Мы советуем комментировать функции, глобальные переменные, определения констант, поля в структурах и классах и вообще все элементы, в понимании сути которых может помочь краткое резюме.

Глобальные переменные имеют тенденцию появляться в разных местах программы, поэтому иногда их стоит сопроводить комментарием с напоминанием об их роли. В качестве примера мы выбрали отрывок программы, приведенной в главе 3 этой книги:

```
    struct State { /* префикс + список суффиксов */
char *pref[NPREF]; /* префиксы */
Suffix *suf; /* список суффиксов */
 State *next; /* следующий в хэш-таблице */
  ):
```

Иногда код действительно сложен, возможно, из-за сложного алгоритма или замысловатых структур данных. В таком случае читателю может помочь комментарий, указывающий на источник, в котором описан данный алгоритм или структура. В него можно также включить пояснения, касающиеся того, почему были приняты те или иные конкретные решения. Ниже приведен комментарий, предваряющий весьма эффективную реализацию обратного дискретного косинус-преобразования (discrete cosine transform - DCT), используемого в декодере изображений в формате JPEG:

```
    /*
 * idct: масштабированная целочисленная
реализация
 * обратного двумерного (8x8) дискретного
 * косинус-преобразования (ОСТ).
 * Алгоритм Чен-Ванга (Chen-Wang)
 * (IEEE ASSP-32, с. 803-816, август 1984) *
 * 32-битовая целочисленная арифметика
 8-битовые коэффициенты
 * 11 умножений, 29 сложений на одно DCT *
 * Коэффициенты увеличены до 12 битов для
совместимости
* с IEEE 1180-1990
 */ static void idct(int b[8*8])
```

В этом содержательном комментарии имеется ссылка на описание алгоритма, кратко представлены используемые данные, обозначена производительность алгоритма и показано, когда и для чего была произведена модификация кода.

Не комментируйте плохой код, а передишите его. Все необычное или потенциально смущающее стоит комментировать, но когда комментарий по своему размеру приближается к коду, то, скорее всего, код стоит подправить. В приводимом примере используются длинный, путаный комментарий и условно компилируемое выражение для распечатки отладочной информации — и все это для пояснения всего лишь одного выражения:

```
    /* Если result = 0, то было найдено совпадение,
поэтому возвращаем true
    (не 0). В противном случае result не равен нулю,
поэтому возвращаем false.(ноль).
     */
tfifdef DEBUG
printf("*** isword возвращает ! result = %d\n", ! result);
 fflush(stdout);
tfendif
return(! result);
```

В отрицаниях всегда легко запутаться, и поэтому лучше обойтись без них. В рассмотренном примере проблема во многом заключается в неграмотном выборе имени для переменной — result. Лучше выбрать более информативное имя — matchfound (совпадение найдено), тогда комментарий станет вообще не нужен и печатаемое при отладке сообщение будет более понятным:

```
#ififdef DEBUG
printf("*** isword возвращает matchfound = %d\n", matchfound);
fflush(stdout);
 #endif
return matchfound;
```

Не противоречьте коду. Как правило, все комментарии согласуются с кодом, когда он пишется, но по мере устранения ошибок и развития программы комментарии часто остаются без изменений и перестают соответствовать коду. Скорее всего, именно из-за этого и возникли проблемы во фрагменте, приведенном в самом начале этой главы.

В чем бы ни крылась причина несоответствия, комментарий, противоречащий коду, сильно сбивает читателя с толку. Страшно представить, сколько времени и сил было потрачено на отладку программ впустую только из-за того, что некорректный или устаревший комментарий принимался на веру. Поэтому запомните: каждый раз, изменив код, убедитесь в том, что комментарии ему соответствуют.

Комментарии должны не только соответствовать коду, но и. поддерживать его. В следующем примере комментарий помещен грамотно — он поясняет назначение следующих за ним строк, но противоречит коду, поскольку в нем говорится о переводе строки, в то время как код относится к пробелам:

```
    time(&now);
strcpy(date, ctime(&now));
 /* избавляемся от замыкающего символа
перевода строки,
скопированного из ctime */ 1 = .0;
while(date[i] >= ' ') i++; date[i] - 0;
```

Первое, что надо сделать, это переписать код в более привычном идиоматическом виде:

```
    time(&now);
strcpy(date, ctime(&now));
/* избавляемся от замыкающего символа перевода строки,
 скопированного из ctime */ for (i = 0; date[i] != Дп'; i++)
date[i] = '\0';
```

Теперь код и комментарий соответствуют друг другу, но и тот и другой можно улучшить, сделав их более прямолинейными. Задача фрагмента — удалить символ перевода строки, который функция ctime помещает в конец возвращаемой ею строки. Собственно, это комментарий и должен сообщить, а код сделать:

```
    time(&now);
 strcpy(date, ctime(&now));
 /* ctimeQ помещает символ перевода строки в конец
возвращаемой строки; его надо удалить'*/
date[strlen(date)-1] =' \O';
```

Последнее выражение — идиома языка С, предназначенная для удаления последнего символа из строки. Теперь наш код стал коротким, стандартизованным и понятным, а комментарий поясняет причину, по которой в этом коде возникла необходимость.

Вносите ясность, а не сумятицу. Комментарии предназначены для того, чтобы помочь читателю разобраться в критических местах кода, а не для того, чтобы создавать дополнительные препятствия. В следующем примере выполнены наши рекомендации по комментированию функций и разъяснению необычных участков кода; однако речь идет о функции strcmp, и все такие необычные участки не имеют никакого отношения к тому, что на самом деле нужно сделать — реализовать стандартный и всем привычный интерфейс:

```
    ? int strcmp(char *s1, char *s2)
? /* процедура сравнения строк возвращает -1, если */ /*
 s1 стоит выше
    s2 в списке по возрастанию, */ /* 0, если строки равны, и 1,
 если s1 ниже
```

```
     s2 */
 ? {
 ? while(*s1==*s2) {
? if(*s1=='\0') return(O);
? S1++;
? s2++;
 ? }
if(*s1>*s2) return(l);
? return(-1);
 ? }
```

Если нескольких слов недостаточно для пояснения происходящего, то скорее всего код стоит переписать. В рассматриваемом примере код, наверное, можно несколько улучшить, но главная беда кроется все же в комментарии, который почти того же размера, что и вся реализация, да к тому же еще и не совсем понятен (что значит "выше", например?). Мы так подробно разбираем этот пример, чтобы подчеркнуть: если во фрагменте реализуется стандартная функция, комментарий должен лишь обобщать сведения о ее поведении и сообщать, откуда взялось ее определение, вот и все:

```
    /* strcmp: возвращает <
     0 если s1<s2,
> 0 если s1>s2
 и 0 если s1=s2 */
 /* ANSI C, раздел 4.11,4.2 */ ,
  int strcmp(const char *s1, const char *s2)
{
....
}
```

Студентам внушают, что комментировать надо все подряд; профессиональным программистам часто вменяется в обязанность комментировать весь созданный ими код. Однако если слепо следовать правилам, то смысл написания комментариев может потеряться. Комментарии должны помогать читателю разобраться в тех частях программы, смысл и назначение которых нельзя (или просто трудно) понять из самого кода. Всюду, где это возможно, старайтесь писать код, который было бы просто понять; чем лучше вам это удастся, тем меньше комментариев вам потребуется. Хороший код меньше нуждается в комментариях, чем плохой.

Упражнение 1-11

Прокомментируйте эти комментарии.

```
    ?void diet::insert(string& w)
?// возвращает 1, если w в словаре, в противном случае - О
?if (п > MAX || п % 2 > 0) // проверка на четность
 ?// Пишем сообщение
?// Увеличить счетчик строк для каждой написанной строки
? void writejnessage()
? // увеличить счетчик строк
? line_number =,line_number + 1;
 ? fprintf(fout, "%d %s\n%d %s\n%d %s\rv",
 ? line^number, HEADER,
7 line_number + 1, BODY,
? line_number + 2, TRAILER);
 ? // увеличить счетчик строк
```

```
? line_number = line_number + 2;
7 }
```

Стоит ли так беспокоиться?

В этой главе мы поговорили об основных составляющих хорошего стиля программирования: информативных именах, ясности в выражениях, прозрачной логике, читабельности кода и комментариев, а также обсудили важность использования соглашений и идиоматических блоков для достижения всего вышеперечисленного.

Однако стоит ли так беспокоиться о стиле? Кому какая разница, как программа выглядит изнутри, если она работает? Не слишком ли много времени придется тратить на ее "причесывание"? Не слишком ли расплывчаты рекомендуемые правила?

Ответ на эти вопросы состоит в следующем: хорошо и красиво написанный код проще читать и воспринимать; в нем, без сомнения, содержится меньше ошибок, и он почти наверняка будет короче, чем код, бездумно скомпонованный и оставленный без улучшений. Когда программа пишется в спешке, когда и так трудно успеть к установленным срокам, кажется вполне естественным не обращать внимания на стиль, оставив заботы о нем на потом. Однако подобное решение может оказаться весьма накладным. Некоторые примеры из этой главы уже дали вам представление о том, что может случиться, если пренебрегать хорошим стилем. Небрежно оформленный код — плохой код, и не только потому, что выглядит он некрасиво и читать его трудно; как правило, в таком коде содержатся и ошибки.

Основная мысль состоит в том, что хороший стиль должен просто войти в привычку. Если вы будете задумываться о стиле при написании кода, если вы будете выкраивать время для того, чтобы проверять и улучшать его стиль, вы выработаете у себя очень полезную привычку. После того как все это вы будете проделывать автоматически, ваше подсознание позаботится о многих деталях, и даже код, который вы будете писать в спешке, станет гораздо лучше.

Дополнительная литература

Как мы уже заявляли в начале главы, писать хороший код и просто хорошо писать по-английски — достаточно схожие понятия. "Элементы стиля" В. Странка и Э. Б. Уайта (W. Strunk, E. B. White. The Elements of Style. Allyn & Bacon) — самый хороший из небольших учебников письменного английского.

Эта глава во многом взята из книги Брайана Кернигана и П. Дж. Пла-угера "Элементы стиля программирования" (Brian Kernighan, P. J. Plau-ger. The Elements of Programming Style. McGraw-Hill, 1978). "Создание надежного кода" Стива Магьюира (Steve Maguire. Writing Solid Code. Microsoft Press, 1993) — идеальный источник советов по программированию. Кроме того, интересное обсуждение стиля имеется в книгах Мак-Коннелла "Все о коде" (Steve McConnell. Code Complete. Microsoft Press, 1993) и Питера ван дер Линдена "Профессиональное программирование на C: Секреты C" (Peter van der Linden. Expert C Programming: Deep C Secrets. Prentice Hall, 1994).

Алгоритмы и структуры данных

- Поиск
- Сортировка
- Библиотеки
- Быстрая сортировка на языке Java
- "О большое"
- Динамически расширяемые массивы
- Списки
- Деревья
- Хэш-таблицы
- Заключение
- Дополнительная литература

> В конце концов, только знание инструментов и технологий обеспечит правильное решение поставленной задачи, и только определенный уровень опыта обеспечит устойчиво профессиональные результаты.
>
> Реймонд Филдинг. Технология специальных эффектов в кинематографе

Исследование алгоритмов и структур данных является одной из основ программирования, а также богатым полем элегантных технологий и сложных математических изысканий. И это — что-то большее, чем развлечение для теоретически подготовленных: хороший алгоритм или структура данных могут позволить решить в течение нескольких секунд проблему, которая без них решалась бы годы.

В таких специальных областях, как графика, базы данных, синтаксический разбор, цифровой анализ и моделирование, возможность решения задачи целиком и полностью зависит от наличия специальных алгоритмов и структур данных. Если вы разрабатываете программы в новых для вас областях программирования, то вы должны выяснить, какие наработки уже существуют, иначе вы потратите свое время впустую в попытках плохо сделать то, что уже кем-то было сделано хорошо.

Каждая программа зависит от алгоритмов и структур данных, но редко бывает нужно изобретать новые алгоритмы. Даже в сложной программе, например в компиляторе или Web-браузере, структуры данных по большей части являются массивами, списками, деревьями и хэш-таб-лицами. Когда программе нужна более изощренная структура, она, скорее всего, будет основываться на этих более простых структурах. Соответственно, задача программиста — знать, какие алгоритмы и структуры доступны, а также понимать, как выбрать среди них нужные.

Такова, вкратце, ситуация. Есть лишь горстка основных алгоритмов, которые применяются практически в каждой программе, — это, прежде всего, поиск и сортировка, и даже эти алгоритмы зачастую включены в библиотеки. Почти все структуры данных также сделаны на основе нескольких фундаментальных структур. Поэтому материал данной главы знаком почти всем программистам. Мы написали работающие версии программ, чтобы дискуссия была более конкретной, и при желании вы можете обратиться непосредственно к исходному коду, но делайте это, только если вы разобрались в том, что вам могут предложить ваш язык программирования и его библиотеки.

Поиск

Ничто не сравнится с массивом, если нам нужно хранить статические табличные данные. Инициализация во время компиляции делает задачу конструирования таких массивов простой и легкой. (В Java инициализация происходит во время выполнения, но это можно считать незначительной деталью реализации, пока массивы не слишком велики.) В программе для распознания слов, слишком часто употребляемых в плохой прозе, мы можем написать:

```
    char *flab[] = {
"actually",
"just",
"quite",
"really",
 NULL
  };
```

Функция поиска должна знать, сколько в массиве элементов. Один из способов сообщить ей это — передать длину массива в виде аргумента; второй способ, использованный здесь, — поместить в конце массива элемент-маркер NULL:

```
    /* lookup: последовательный поиск слова в массиве */
 int lookup(char *word, char *array[])
  {
 int i;
 for (i = 0; array[i] != NULL; i++)
if (strcmp(word, array[i]) ==-
0) return  i;
return -1;
}
```

В C и C++ для передачи в качестве параметра массив строк можно описать как char *array[] или char **array. Эти две формы эквивалентны, но в первой сразу видно, как будет использоваться параметр.

Предлагаемый поисковый алгоритм называется последовательным, поиском, потому что он просматривает по очереди все элементы, сравнивая их с искомым. Когда данных немного, последовательный поиск работает достаточно быстро. Есть стандартные библиотечные функции, которые выполняют последовательный поиск для определенных типов данных. Например, в языках C и C++ функции st rch r и st rst r ищут первое вхождение заданного символа или подстроки в строку, в Java у класса String есть метод indexOf, а обобщенные функции поиска в C++ find применимы к большинству типов данных. Если такая функция существует для нужного вам типа данных, то используйте ее.

Последовательный поиск достаточно прост, но время его работы прямо пропорционально количеству данных, которые нужно просмотреть; удвоение количества элементов приведет к удвоению времени на поиск, если искомого элемента в массиве нет. Это линейное соотношение (время выполнения является линейной функцией от размера данных), поэтому такой метод также называется линейным поиском.

Вот пример массива более реалистичного размера из программы, выполняющей синтаксический разбор текста, написанного на HTML, где определены имена более чем сотни отдельных символов:

```
    typedef struct Nameval Nameval; struct Nameval {
char *name;
 int value;
/* символы HTML, например AElig - лигатура1 A и E.
*/ /* значения в кодировке
    Unicode/18010646 */ Nameval htmlchars[] = {
"AElig", OxOOcб, /* лигатура A и E
"Aacute", OxOOd, /* A с акцентом
"Acirc", OxOOc2, /* A с кружочком
  /*...*/
"zeta", OxO3Ьб, /* греческая дзета */
 };
```

Для объемистого массива вроде этого более эффективно было бы использовать двоичный поиск. Алгоритм двоичного поиска является систематизированной версией поиска слова в словаре. Проверяем средний элемент. Если это значение больше, чем нужное, то ищем далее в первой части; в противном случае ищем во второй части. Повторяем до тех пор, пока не найдем нужный элемент или не убедимся, что его в массиве нет.

Для двоичного поиска таблица должна быть отсортирована, как в данном случае (в любом случае это полезно; люди тоже быстрее находят требуемое в отсортированных таблицах), а также должно быть известно, сколько элементов в таблице. Здесь может помочь макрофункция NELEMS из первой главы:

```
    printf("Таблица HTML содержит %d слов\n", NELEMS(htmlchars));
```

Функция двоичного поиска для этой таблицы могла бы выглядеть так:

```
    /* lookup: двоичный поиск имени name
 в таблице tab;
возвращается индекс */
int lookup(char *name, Nameval tab[], int ntab) {
 int low, high, mid, cmp;
low = 0;
high = ntab - 1;
 while (low <= high) {
mid = (low + high) / 2;
cmp = strcmp(name, tab[mid].name);
if (cmp < 0)
high = mid - 1; else if (cmp > 0) low = mid + 1:
 else /* совпадение
    найдено */
return mid; } return -1;
 /* совпадений нет */
```

Объединяя все это вместе, мы можем написать:

```
    half = lookup("frac12", htmlchars, NElEMS(htmlchars));
```

для определения индекса, под которым символ 1/2 (одна вторая) стоит в массиве htmlchars.

Двоичный поиск отбрасывает за каждый шаг половину данных, поэтому количество шагов пропорционально тому, сколько раз мы можем поделить n на 2, пока у нас не останется один элемент. Без учета округления это число равно log2 n. Если у нас в

массиве 100"0 элементов, то линейный поиск займет до 1000 шагов, в то время как двоичный — только около 10; при миллионе элементов линейный поиск займет миллион шагов, а двоичный — 20. Очевидно, чем больше число элементов, тем больше преимущество двоичного поиска. Начиная с некоторого зависящего от реализации размера данных, двоичный поиск работает быстрее, чем линейный.

Сортировка

Двоичный поиск работает только в том случае, если элементы отсортированы. Если по одному и тому же набору данных планируется неоднократный повторный поиск, то выгоднее один раз отсортировать данные, а затем использовать двоичный поиск. Если набор данных известен заранее, то он может быть отсортирован при написании программы и проинициализирован во время компиляции. Иначе придется сортировать его во время выполнения программы.

Один из самых лучших алгоритмов сортировки — быстрая сортировка (quicksort), которая была придумана в 1960 году Чарльзом Хоаром (C. A. R. Hoare). Быстрая сортировка — замечательный пример того, как можно избежать лишних вычислений. Она работает при помощи разделения массива на большие и маленькие элементы:

> выбрать один элемент массива ("разделитель"):
> разбить оставшиеся элементы на две группы:
> "маленькие", то есть меньшие, чем разделитель,
> "большие", то есть большие или равные разделителю,
> рекурсивно отсортировать обе группы.

Когда этот процесс закончится, то массив будет отсортирован. Быстрая сортировка работает быстро, потому что, как только мы узнаем, что элемент меньше, чем разделитель, нам уже не нужно его сравнивать с большими элементами; аналогично, большие элементы не сравниваются с маленькими. Поэтому данный алгоритм существенно быстрее, чем такие простые методы сортировки, как сортировка вставкой или пузырьком, когда каждый элемент сравнивается напрямую со всеми остальными.

Алгоритм быстрой сортировки практичен и эффективен; он хорошо изучен, и существует множество его вариаций. Версия, которую мы здесь представим, является одной из самых простых реализаций, но, конечно, далеко не самой быстрой.

Наша функция quicksort сортирует массив целых чисел:

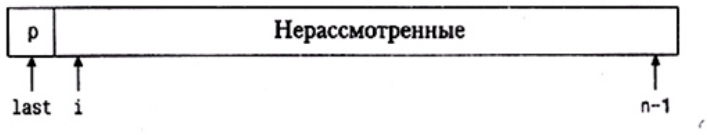
```
    /* quicksort: сортирует v[0]..v[n-1]
по возрастанию */ void quicksortClrrt
    v[], int n)
> {
  int i, last;
 if (n <= 1) /* ничего не делать */
return;
 swap(v, 0, rand() % n); /* поместить
 разделитель в v[0] */ last = 0;
 for (i = 1; i < n; i++) /* разделить */
 if (v[i] < v[0])
 swap(v, ++last, i);
```

```
    swap(v, 0, last); /* сохранить разделитель */
quicksort(v, last); /* рекурсивно
    отсортировать */ quicksort(v+last+1, n-last-1);
/* каждую часть */ }
```

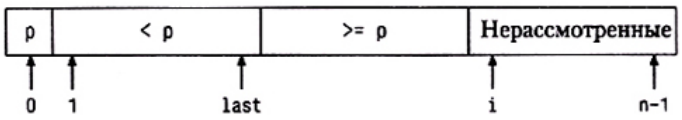
Операция swap, которая меняет местами два элемента, встречается в quicksort трижды, поэтому лучше всего вынести ее в отдельную функцию:

```
    /* swap:
     поменять местами v[i] и v[j] */
void swap(int v[], int i, int j)
{
 int temp;
 temp = v[i]; v[i] = v[j]; v[j] = temp;
```

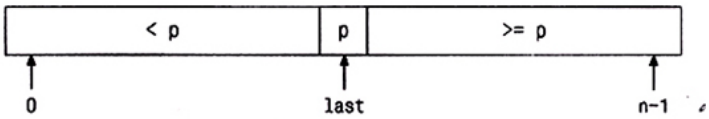
При разделении прежде всего случайным образом выбирается элемент-разделитель, который временно переставляется в начало массива, затем просматриваются остальные. Элементы, меньшие разделителя ("маленькие" элементы), перемещаются ближе к началу массива (на позицию last), а "большие" элементы — в сторону конца массива (на позицию i). В начале процесса, сразу после перемещения разделителя в начало, last = 0 и элементы с i = 1 по n-1 еще не исследованы:



В начале 1-й итерации элементы с первого по last строго меньше разделителя, элементы с last+1 no i-1 больше или равны разделителю, а элементы с i по n-1 еще не рассмотрены. Пока не выполнилось первый раз условие v[i] >='v[0], алгоритм будет переставлять элемент v[i] сам с собой; это, конечно, занимает дополнительное время, но не столь страшно.



Когда все элементы просмотрены, нулевой элемент переставляется на позицию last, чтобы разделитель занял свою окончательную позицию; тогда порядок элементов будет правильным. Теперь массив выглядит так:



Та же самая операция применяется к левой и правой частям массива; после окончания этого процесса весь массив будет отсортирован.

Насколько быстро работает быстрая сортировка? В наилучшем случае

- первый проход делит массив из п элементов на две группы примерно по п/2 элементов;
- второй проход разделяет две группы по п/2 элементов на 4 группы, в каждой из которых примерно по п/4 элементов;
- на следующем проходе четыре группы по п/4 делятся на восемь групп по (примерно) п/8 элементов;
- и т. д.

Данный процесс продолжается примерно log2 n раз, поэтому общее время работы в лучшем случае пропорционально п + 2 X и/2 + 4 x и/4 + + 8 X п/8 ... (log2 и слагаемых), что равно п log? п. В среднем алгоритм работает совсем не намного дольше. Обычно принято использовать именно двоичные логарифмы, поэтому мы можем сказать, что быстрая сортировка работает пропорционально n long.

Эта демонстрационная реализация быстрой сортировки наиболее прозрачна, но у нее есть одна слабина. Если каждый выбор разделителя разбивает массив на две примерно одинаковые группы, то наш анализ корректен, однако если разделение слишком часто происходит неровно, то время работы будет расти скорее как п1. В нашей реализации в качестве разделителя берется случайный элемент, чтобы уменьшить шанс того, что плохие входные данные приведут к слишком большому количеству неровных разбиений массива. Но если все входные значения одинаковы, то наша реализация за каждый проход будет отделять только один элемент, поэтому время работы будет расти как п2.

Поведение некоторых алгоритмов сильно зависит от входных данных. Неправильный или неудачный ввод может заставить в среднем хороший алгоритм работать крайне медленно или использовать огромное количество памяти. В случае быстрой сортировки, хотя простые реализации вроде нашей иногда могут работать медленно, более продуманные реализации способны уменьшить шанс патологического поведения почти до нуля.

Библиотеки

В стандартные библиотеки языков С и С++ входят функции сортировки, которые устойчивы к неблагоприятным входным данным и настроены на предельно быструю работу.

Библиотечные функции написаны так, что они могут сортировать данные любого типа, но мы в свою очередь должны адаптироваться к их интерфейсу, что может быть несколько сложнее, чем в рассмотренном выше примере. В языке С библиотечная функция называется qso rt, и ей нужно предоставлять функцию сравнения двух значений. Поскольку значения могут быть любых типов, то функции сравнения передаются два нетипизированных указателя (void *) на сравниваемые значения. Функция преобразует указатели к нужному типу, извлекает значения данных, сравнивает их и возвращает результат (отрицательный, нуль или положительный в зависимости от того, меньше ли первый элемент, равен ли второму или больше его).

Рассмотрим реализацию функции сравнения для сортировки массива строк, случая, встречающегося довольно часто. Мы написали функцию scmp, которая -преобразует параметры к другому типу и затем вызывает st rcmp для выполнения самого сравнения:

```
    /* scmp: сравнение строк *p1 и *p2 */
int scmp(const void *p1, const void *p2)
{
char *v1, *v2;
v1 = *(char **) p1; v2 = *(char **) p2;
 return strcmp(v1, v2); }
```

Мы могли бы написать эту функцию в одну строку, но при использовании временных переменных код становится более удобочитаемым.

Мы не можем напрямую использовать strcmp как функцию сравнения, поскольку qsort передает адрес каждого элемента в массиве &s t r [i] (типаспаг **), ане str[i] (типа char *), как показано на рисунке:

Для сортировки элементов массива строк с st r[0] по st r[N-1] функция qsort должна получить массив, его длину, размер сортируемых элементов и функцию сравнения:

```
    char *str[N];
qsort(str, N, sizeof(str[OJ), scmp);
```

А вот аналогичная функция icmp для сравнения целых:

```
    {
int v1, v2;
 v1 = *(int *) p1; v2 = *(int *.) p2;
 if (v1 < v2)
 return -1; else
 if (v1 == v2)
return 0; else
return 1; >
 }
```

Мы могли бы написать

```
    ? return v1-v2;
```

но если v2 — большое положительное число, а v1 — большое по абсолютному значению отрицательное или наоборот, то получившееся переполнение привело бы к неправильному ответу. Прямое сравнение длиннее, но надежнее.

И здесь при вызове qsort нужно передать массив, его длину, размер сортируемых элементов и функцию сравнения:

```
    int arr[N];
qsort(arr, N, sizeof(arr[0]), icmp);
```

В стандарте ANSI C определена также функция двоичного поиска bsea rch. Как и qso rt, этой функции нужен указатель на функцию сравнения (часто на ту же, что используется для qsort); bsearch возвращает указатель на найденный элемент или на NULL, если такого элемента нет. Вот наша программа для поиска имен в HTML-файле, переписанная с использованием bsearch:

```
    /* lookup: использует bsearch для поиска name в таблице tab,
возвращает индекс */
int lookup(char *name, Nameval tab[],
 int ntab) {
Nameval key, *np;


 key.value = 0; /* не используется;
годится любое значение */
np = (Nameval *) bsearch(&key, tab, ntab,
sizeof(tab[0]), nvcmp); if (np == NULL)
 return -1; else
return np-tab;
```

Как и в случае qsort, функция сравнения получает адреса сравниваемых значений, поэтому ключевое (искомое) значение должно иметь этот же тип; в данном примере нам пришлось создать фиктивный элемент типа Nameval для передачи в функцию сравнения. Сама функция сравнения nvcmp сравнивает два значения типа Nameval, вызывая st rcmp для их строковых компонентов, полностью игнорируя численные компоненты:

```
    /* nvcmp: сравнивает два вначения типа
Nameval */
 int nvcmp(const void *va, const void *vb)
 {
 const Nameval *a, *b;
a = (Nameval *) va;* b = (Nameval *) vb;
 return strcmp(a->name, b->name); }
```

Это похоже на scmp, только с тем отличием, что строки хранятся как члены структуры.

Неудобство передачи ключевого значения показывает, что bsearch предоставляет меньше возможностей, чем qso rt. Хорошая многоцелевая функция сортировки занимает одну-две страницы кода, а двоичный поиск — ненамного больше, чем код для интерфейса с bsearch. Тем не менее лучше использовать bsearch, чем писать свою собственную версию. Как показывает опыт, программистам на удивление трудно написать двоичный поиск без ошибок.

В стандартной библиотеке C++ имеется обобщенная функция sort, которая обеспечивает время работы 0(п log n). Код для ее вызова проще, потому что нет необходимости в преобразовании типов и размеров элементов. Кроме того, для порядковых типов не требуется задавать функцию сравнения:

```
     int arr[N]; sort(arr, arr+N);
```

Эта библиотека содержит также обобщенные функции двоичного поиска, с теми же преимуществами в вызове.

Упражнение 2-1

Алгоритм быстрой сортировки проще всего выразить рекурсивно. Реализуйте его итеративно и сравните две версии. (Хоар рассказывает, как было трудно разработать итеративный ваариант быстрой сортировки и как легко все оказалось, когда он сделал ее рекурсивной.)

Быстрая сортировка на языке Java

В Java ситуация другая. В ранних версиях не было стандартной функции сортировки, поэтому приходилось писать собственную. В последних версиях появилась такая функция, работающая с классами, реализующими интерфейс Comparable, поэтому теперь мы можем просить библиотеку сортировать то, что нам потребуется. Но поскольку используемые технологии полезны и в других ситуациях, в данном разделе мы опишем все детали реализации быстрой сортировки в Java.

Адаптировать быструю сортировку для каждого конкретного типа данных легко; однако же более поучительно написать обобщенную функцию для сортировки объектов любых типов, что больше похоже на интерфейс qsort.

Одно из крупных отличий от C и C++ заключается в том, что в Java мы не можем передать функцию сравнения в другую функцию — здесь не существует указателей на функции. Вместо этого мы создаем интерфейс (interface), единственным содержимым которого будет функция, сравнивающая два объекта типа Object. Далее для каждого сортируемого типа данных мы создаем класс с функцией (методом), которая реализует интерфейс для этого типа данных. Мы передаем экземпляр класса в функцию сортировки, которая в свою очередь использует функцию сравнения из этого класса для сравнения элементов.

Сначала опишем интерфейс Cmp, который определяет единственную функцию cmp, сравнивающую два значения типа Object:

```
     interface Cmp {
int cmp(0bject x, Object y);
}
```

Теперь мы можем написать функции сравнения, которые реализуют этот интерфейс; например, следующий класс определяет функцию, сравнивающую объекты типа Integer:

```
    // Icmp: сравнение целых class
 Icmp implements Cmp {
public int cmp(0bject o1, Object o2)
 {
 int 11 = ((Integer) o1). intValueO;
int 12 =
((Integer) o2). intValueO; if (11 < 12)
 return -1; else if (11 == 12)
return 0; else
 return 1; } }
;
```

а эта функция сравнивает объекты типа St ring:

```
    // Scmp: сравнение
    строк class
Scmp implements Cmp {
 public int cmp(0bject o1, Object o2)
{
 String s1 = (String) o1; String s2 =
(String) o2; return s1.compareTo(s2);
 } }
```

Данным способом можно сортировать только типы, наследуемые от класса Object; подобный механизм нельзя применять для базовых типов, таких как int или double. Поэтому мы сортируем элементы типа Integer,
а не int.

С этими компонентами мы теперь можем перенести функцию быстрой сортировки из языка С в Java и вызывать в ней функцию сравнения из объекта Стр, переданного параметром. Наиболее существенное изменение — это использование индексов left и right, поскольку в Java нет указателей на массивы.

```
     // Quicksort.sort: сортировать v[left]..v[right]
 // алгоритмом quicksort
static void sort(0bject[] v, int left,
 int right, Cmp cmp)
{
 int i, last;
 if (left >= right) // ничего делать не надо
 return;
 swap(v, left, rand(left,right)); // поместить
 разделитель last = left; // в v[left]
for (i = left+1; i <= right; i++) // разделить
массив if (cmp.cmp(v[i], v[left]) < 0)
swap(v, ++last, i);
swap(v, left, last); // вернуть разделитель
sort(v, left, last-1, cmp); // рекурсивно
отсортировать sort(v, last+1, right, cmp);
// обе части }
```

Quicksort.sort использует cmp для сравнения двух объектов, как и раньше, вызывает swap для их обмена.

```
    // Quicksort.swap: обменять v[i]
и v[j] static void swap
(0bject[] v, int i, int j) {
Object temp;
 temp = v[i]; v[i] = v[j]
 ; v[j] = temp; }
```

Генерация случайного номера происходит в функции rand, которая а возвращает случайное число в диапазоне с left по right включительно:

```
    static
```

```
    Random rgen = new Random();
// Quicksort, rand: возвращает
 случайное целое число // из [left, right]
static int rand(int left, int right)
{
return left + Math.abs
(rgen.nextlnt())%(right-left+1);
}
```

Мы вычисляем абсолютное значение (используя функцию Math.abs), поскольку в Java генератор случайных чисел возвращает как положительные, так и отрицательные значения.

Функции sort, swap и rand, а также объект-генератор случайных чисел rgen являются членами класса Quicksort.

Наконец мы готовы написать вызов Quicksort, sort для сортировки массива типа String:

```
    String[] sarr = new String[n];
// заполнить n элементов sarr...
Quicksort. sort
(sarr, 0, sarr. length-1, new ScmpO);
```

Так вызывается so rt с объектом сравнения строк, созданным для этой цели.

Упражнение 2-2

Наша реализация быстрой сортировки в Java делает несколько преобразований типов, сначала переводя исходные данные из их первоначального типа (вроде Integer) в Object, а затем обратно. Поэкспериментируйте с версией Quickso rt. so rt, которая использует конкретный тип при сортировке, и попробуйте вычислить, какие потери производительности вызываются преобразованием типов.

"О большое"

Мы описывали трудоемкость алгоритма в зависимости от n, количества входных элементов. Поиск в неотсортированных данных занимает время, пропорциональное n; при использовании двоичного поиска по отсортированным данным время будет пропорционально log n. Время сортировки пропорционально n^2 или n logn.

Нам нужно как-то уточнить эти высказывания, при этом абстрагируясь от таких деталей, как скорость процессора и качество компилятора (и программиста). Хотелось бы сравнивать время работы и затраты памяти алгоритмов вне зависимости от языка программирования, компилятора, архитектуры компьютера, скорости процессора, загруженности системы и других сложных факторов.

Для этой цели существует стандартная форма записи, которая называется "О большое". Основной параметр этой записи — n, размер входных данных, а сложность или время работы алгоритма выражается как функция от n. "О" — от английского order, то есть порядок. Например, фраза "Двоичный поиск имеет сложность 0(log n)" означает, что для поиска в массиве из n элементов требуется порядка log n действий. Запись O(f(n)) предусматривает, что при достаточно больших n время выполнения пропорционально f(n), не быстрее, например, $O(n^2)$ или 0(n log n). Асимптотические оценки вроде этой полезны при теоретическом

анализе и грубом сравнении алгоритмов, однако на практике разница в деталях может иметь большое значение. Например, алгоритм класса $0(n^2)$ с малым количеством дополнительных вычислений для малых п может работать быстрее, чем сложный алгоритм класса O(n logn), однако при достаточно большом п алгоритм с медленнее возрастающей функцией поведения неизбежно будет работать быстрее.

Нам нужно различать также случаи наихудшего и ожидаемого поведения. Трудно строго определить, что такое "ожидаемое" поведение, потому что определение зависит от наших предположений о возможных входных данных. Обычно мы можем точно указать самый плохой случай, хотя иногда и здесь можно ошибиться. Для quicksort в самом плохом случае время работы растет как $O(n^2)$, а среднее ("ожидаемое") время — как O(n log n). Если каждый раз аккуратно выбирать элемент-разделитель, то мы можем свести вероятность квадратичного (то есть $0(n^2)$) поведения практически к нулю; хорошо реализованная quicksort действительно обычно ведет себя как O(n log n).

Вот основные случаи:

Запись	Название времени	Пример
0(1)	Константное	Индексирование массива
0(log n)	Логарифмическое	Двоичный поиск
0(n)	Линейное	Сравнение строк
0(n log n)	n logn	Quicksort
$0(n^2)$	Квадратичное	Простые методы сортировки
$O(n^3)$	Кубическое	Перемножение матриц
$0(2^n)$	Экспоненциальное	Перебор всех подмножеств

Доступ к элементу в массиве — операция, работающая за константное (O(1)) время. Алгоритм, за каждый шаг отсеивающий половину входных данных, как двоичный поиск, обычно займет время 0(log n). Сравнение двух строк длиной в n символов с помощью strcmp займет O(n).

Традиционный алгоритм перемножения двух квадратных матриц порядка п занимает O(и!), поскольку каждый элемент получается в результате перемножения и пар чисел и суммирования результатов, а всего элементов n^2.

Экспоненциальное время работы алгоритма обычно является результатом перебора всех вариантов: у множества из п элементов — 2"различных подмножеств, поэтому алгоритм, которому надо пройтись по всем подмножествам, будет выполняться за время 0(2"), то есть будет экспоненциальным. Экспоненциальные алгоритмы обычно слишком долго работают, если только п не очень мало, поскольку добавление одного элемента удваивает время работы алгоритма. К сожалению, существует много задач, таких как, например, знаменитая "задача коммивояжера", для которых известны только экспоненциальные решения. Когда задача такова, часто вместо точных решений берут алгоритмы, находящие некоторое приближение к ответу.

Упражнение 2-3

Каковы входные данные для алгоритма quicksort, которые заставляют его работать медленнее всего, как в наихудшем случае? Попробуйте найти несколько наборов данных, сильно замедляющих библиотечную версию алгоритма. Автоматизируйте процесс, чтобы вы легко могли задавать параметры и проводить большое число экспериментов.

Упражнение 2-4

Придумайте и реализуйте алгоритм, который будет сортировать массив из n целых как можно медленнее. Только напишите его честно: алгоритм должен постепенно прогрессировать и в конце концов завершиться, и ваша реализация не должна использовать всяческие трюки вроде лишних пустых циклов. Какова получилась сложность вашего алгоритма как функция от n?

Динамически расширяемые массивы

Массивы, использованные в нескольких предыдущих разделах, были статическими, их размер и содержимое задавались во время компиляции. Если потребовать, чтобы некоторую таблицу слов или символов HTML можно было изменять во время выполнения, то хэш-таблица была бы для этой цели более подходящей структурой. Вставка в отсортированный массив n элементов по одному занимает $O(n^2)$, чего стоит избегать при больших n.

Однако часто нам нужно работать с переменным, но небольшим числом значений, и тогда массивы по-прежнему могут применяться. Для уменьшения потерь при перераспределении памяти изменение размера должно происходить блоками, и для простоты массив должен храниться вместе с информацией, необходимой для управления им^ В C++ и Java это делается с помощью классов из стандартных библиотек, а в C мы можем добиться похожего результата с помощью структур.

Следующий код определяет расширяемый массив с элементами типа Nameval: новые элементы добавляются в хвост массива, который удлиняется при необходимости. Доступ к каждому элементу по его индексу происходит за константное время. Эта конструкция аналогична векторным классам из библиотек C++ и Java.

```
    typedef struct Nameval Nameval;
struct Nameval {
 char *name;
int value; };
struct NVtab {
int nval; /* текущее количество
 элементов */
int max; /* под сколько элементов
выделена память */
Nameval *nameval; /* массив пар */
 } nvtab;
enum { NVINIT = 1, NVGROW = 2 };
/* addname: добавить новое имя
и значение в nvtab */
 int addname(Nameval newname)
 {
Nameval *nvp;
if (nvtab.nameval == NULL) { /* первый вызов
```

```
*/ nvtab.nameval =
(Nameval *) malloc(NVINIT * sizeof(Nameval));
if (nvtab.nameval == NULL)
return -1; nvtab.max = NVINIT; nvtab.nval = 0;
 } else if (nvtab.nval >= nvtab.max) { /* расширить
 */ nvp = (Nameval *) realloc(nvtab.nameval,
(NVGROW*nvtab.max) * sizeof(Nameval));
    if (nvp == NULL)
 return -1;
nvtab.max *= NVGROW;
nvtab.nameval = nvp; }
nvtab.nameval[nvtab.nval]
 = newname; return nvtab.
```

Функция addname возвращает индекс только что добавленного элемента или -1 в случае возникновения ошибки.

Вызов realloc увеличивает массив до новых размеров, сохраняя существующие элементы, и возвращает указатель на него или NULL при недостатке памяти. Удвоение размера массива при каждом вызове realloc сохраняет средние "ожидаемые" затраты на копирование элемента постоянными; если бы массив увеличивался каждый раз только на один элемент, то производительность была бы порядка 0(п2). Поскольку при перераспределении памяти адрес массива может измениться, то программа должна обращаться к элементам массива по индексам, а не через указатели. Заметьте, что код не таков:

```
    ? nvtab.nameval = (Nameval *) realloc(nvtab.nameval,
? (NVGROW*nvtab.jnax) * sizeof (Nameval));
```

потому что при такой форме если вызов realloc не сработает, то исходный массив будет утерян.

Мы задаем очень маленький начальный размер массива (NVINIT = 1). Это заставляет программу увеличивать массив почти сразу, что гарантирует проверку данной части программы. Начальное значение может быть и увеличено, когда программа начнет реально использоваться, однако затраты на стартовое расширение массива ничтожны.

Значение, возвращаемое realloc, не обязательно должно приводиться к его настоящему типу, поскольку в С нетипизированные указатели (void *) приводятся к любому типу указателя автоматически. Однако в С++ это не так; здесь преобразование обязательно. Можно поспорить, что безопаснее: преобразовывать типы (это честнее и понятнее) или не преобразовывать (поскольку в преобразовании легко может закрасться ошибка). Мы выбрали преобразование, потому что тогда программа корректна как в С, так и в С++. Цена такого решения — уменьшение количества проверок на ошибки компилятором С, но это неважно, когда мы производим дополнительные проверки с помощью двух компиляторов.

Удаление элемента может быть более сложной операцией, поскольку нам нужно решить, что же делать с образовавшейся "дыркой" в массиве. Если порядок элементов не имеет значения, то проще всего переставить последний элемент на место удаленного. Однако, если порядок должен быть сохранен, нам нужно сдвинуть элементы после "дырки" на одну позицию:

```
    /* delname: удалить первое
 совпавшее имя в массиве nvtab */
int delname(char *name)
{
 int i;
for (i = 0; i < nvtab.nval; i++)
 if (strcmp(nvtab.nameval[i].name,
name)
== 0) { meminove(nvtab. nameval+i, nvtab.
nameval'+i+1, (nvtab.nval-(i+1)
) * sizeof(Nameval));
 nvtab.nval--; return 1;
}
    return 0:
}
```

Вызов memmove сдвигает массив, перемещая элементы вниз на одну позицию; memmove — стандартная библиотечная функция для копирования блоков памяти любого размера.
В стандарте ANSI

определены две функции: memcpy, которая работает быстрее, но может затереть память, если источник и приемник данных пересекаются, и memmove, которая может работать медленнее, но зато всегда корректна. Бремя выбора между скоростью и корректностью не должно взваливаться на программиста; должна была бы быть только одна функция. Считайте, что это так и есть, и всегда используйте memmove.

Мы могли бы заменить вызов memmove следующим циклом:

```
    nt j;
 for (j = i; j
< nvtab.nval-1; j++)
 nvtab.namevalfj]
= nvtab.nameval[j+1];i
```

Однако мы предпочитаем использовать memmove, чтобы обезопасить себя от легко возникающей ошибки копирования элементов в неправильном порядке. Если бы мы вставляли элемент, а не удаляли, то цикл должен был бы идти вверх, чтобы не затереть элементы. Вызывая memmove, мы ограждаем себя от необходимости постоянно задумываться об этом.

Альтернатива перемещению элементов — помечать удаленные элементы как неиспользуемые. Тогда для добавления элемента нам надо сначала найти неиспользуемую ячейку и увеличивать массив, только если свободного места не найдено. В данном примере элемент можно пометить как неиспользуемый, установив значения поля name в NULL.

Массивы — простейший способ группировки данных; вовсе не случайно большинство языков имеют эффективные и удобные индексируемые массивы и даже представляют строки в виде массивов символов. Массивы просты в использовании, обладают константным доступом к любому элементу, хорошо работают с двоичным поиском и быстрой сортировкой, а также почти совсем не тратят лишних ресурсов. Для наборов данных фиксированного размера, которые могут быть созданы даже во время компиляции, или же для гарантированно небольших объемов данных массивы подходят идеально. Однако хранение меняющегося набора значений в массиве может быть весьма ресурсоемким,

поэтому, если количество элементов непредсказуемо и потенциально неограниченно, может оказаться удобнее использовать другую структуру данных.

Упражнение 2-5

В приведенном выше коде функция delname не вызывает realloc для возврата системе памяти, освобожденной удалением элемента. Имеет ли смысл это делать? Как решить, стоит это делать или нет?

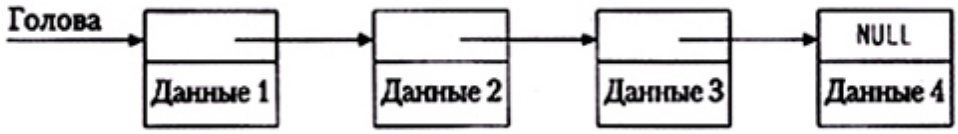
Упражнение 2-6

Внесите необходимые изменения в функции delname и addname, чтобы удаленные элементы помечались как неиспользуемые. Насколько остальная программа не зависит от этого изменения?

Списки

По своей встречаемости в типичных программах списки занимают второе место после массивов. Многие языки имеют встроенные типы списков, некоторые, такие как Lisp, даже построены на них, но в языке C мы должны конструировать их самостоятельно. В C++ и Java работа со списками поддерживается стандартными библиотеками, но и в этом случае нужно знать их возможности и типичные применения. В данном параграфе мы собираемся обсудить использование списков в C, но уроки из этогр обсуждения можно извлечь и для более широкого применения.

Простым цепным списком (single-linked list) называется последовательность элементов, каждый из которых содержит данные и указатель на следующий элемент. Головой списка является указатель на первый элемент, а конец помечен нулевым указателем. Ниже показан список из четырех элементов:



Есть несколько важных различий между массивами и списками. Во-первых, размер массивов фиксирован, а список всегда имеет именно такой размер, который нужен для хранения содержимого, плюс некоторое дополнительное место для указателей на каждый элемент. Во-вторых, списки можно перестраивать, изменяя несколько указателей, что дешевле, чем копирование блоков, необходимое при использовании массива. Наконец, при удалении или вставке элементов остальные элементы не перемещаются; если мы будем хранить указатели на отдельные элементы в другой структуре данных, то при изменениях в списке они останутся корректными.

Эти различия подсказывают, что если набор данных будет часто меняться, особенно при непредсказуемом количестве элементов, то нужно использовать список; напротив, массив больше подходит для относительно статичных данных.

Фундаментальных операций со списком совсем немного: добавить элемент в начало или конец списка, найти указанный элемент, добавить новый элемент до или после

указанного элемента и, возможно, удалить элемент. Простота списков позволяет при необходимости легко добавлять новые операции.

Вместо того чтобы определить специальный тип List, обычно в С начинают определение списка с описания типа для элементов, вроде нашего Nameval, и добавляют в него указатель на следующий элемент:

```
    typedef struct Nameval Nameval;
struct Nameval {
char *name:
int value;
Nameval «next;
 /* следующий в списке */ };
```

Инициализировать непустой список во время компиляции трудно, поэтому списки, не в пример массивам, создаются динамически. Для начала нам нужен способ создания элемента. Наиболее простой подход — выделить под него память специальной функцией, которую мы назвали newitem:

```
    /* newitem: создать новый элемент
по полям name и value */ Nameval
*newitem(char *name, int value)
 {
 Nameval *newp;
newp = (Nameval *) emalloc(sizeof(Nameval));
newp->name = name; newp->value
 = value; newp->next = NULL; return newp; }
```

Функцию emalloc мы будем использовать и далее во всей книге; она вызывает mall ос, а при ошибке выделения памяти выводит сообщение и завершает программу. Мы представим код этой функции в главе 4, а пока считайте, что эта функция всегда корректно и без сбоев выделяет память.

Простейший и самый быстрый способ собрать список — это добавлять новые элементы в его начало:

```
    /* addfront: добавить элемент newp
в начало списка listp */ Nameval
*addfront
(Nameval *listp, Nameval *newp)
{
 newp->next = listp;
return newp; }
```

При изменении списка у него может измениться первый элемент, что и происходит при вызове addf ront. Функции, изменяющие список, должны возвращать указатель на новый первый элемент, который хранится в переменной, указывающей на список. Функция addfront и другие функции этой группы передают указатель на первый элемент в качестве возвращаемого значения; вот типичное использование таких функций:

```
     nvlist = addf ront(nvlist, newitem( "smiley", Ox263A));
```

Такая конструкция работает, даже если существующий список пуст (NULL), она хороша и тем, что позволяет легко объединять вызовы функций в выражениях. Это более естественно, чем альтернативный вариант — передавать указатель на указатель на голову списка.

Добавление элемента в конец списка — процедура порядка 0(п), поскольку нам нужно пройтись по всему списку до конца:

```
    /* addend: добавить элемент newp
в конец списка listp */
Nameval *addend(Nameval *listp,
Namevai *newp)
{
Nameval *p;
 if (listp == NULL)
 return newp; for (p = listp; p->next !=
NULL; p = p->next)
p->next = newp; return listp; }
```

Чтобы сделать addend операцией порядка 0(1), мы могли бы завести отдельный указатель на конец списка. Недостаток этого подхода, кроме того, что нам нужно заботиться о корректности этого указателя, состоит в том, что список теперь уже представлен не одной переменной, а двумя. Мы будем придерживаться более простого стиля.

Для поиска элемента с заданным именем нужно пройтись по указателям next:

```
    /* lookup: последовательный поиск
имени в списке */
Nameval *lookup(Nameval *listp, char *name)
{
 for ( ; listp != NULL; listp = listp->next)
if (strcmp(name, listp->name) == 0)
 return listp;
 return NULL; /* нет совпадений */ }
```

Поиск занимает время порядка 0(п), и, в принципе, эту оценку не улучшить. Даже если список отсортирован, нам все равно нужно пройтись по нему, чтобы добраться до нужного элемента. Двоичный поиск к спискам неприменим.

Для печати элементов списка мы можем написать функцию, проходящую по списку и печатающую каждый элемент; для вычисления длины списка — функцию, проходящую по нему, увеличивая счетчик, и т. д. Альтернативный подход — написать одну функцию, apply, которая проходит по списку и вызывает другую функцию для каждого элемента. Мы можем сделать функцию apply более гибкой, предоставив ей аргумент, который нужно передавать при каждом вызове функции. Таким образом, у apply три аргумента: сам список, функция, которую нужно применить к каждому элементу списка, и аргумент для этой функции:

```
    /* apply: применить
    функцию fn
для каждого элемента списка listp
*/ void apply(Nameval *listp,
 void (*fn)(Nameval*, void*), void *arg)
 {
 for ( ; listp != NULL; listp
 = listp->next) (*fn)(listp, arg);
/* вызов функции */ }
```

Второй аргумент apply — указатель на функцию, которая принимает два параметра и возвращает void. Стандартный, хотя и весьма неуклюжий, синтаксис

```
    * void (*fn)(Nameval*, void*)
```

определяет f n как указатель на функцию с возвращаемым значением типа void, то есть как переменную, содержащую адрес функции, которая возвращает void. Функция имеет два параметра — типа Nameval * (элемент списка) и void * (обобщенный указатель на аргумент для этой функции).

Для использования apply, например для вывода элементов списка, мы можем написать тривиальную функцию, параметр которой будет восприниматься как строка форматирования:

```
    /* printnv: вывод имени и значения
 с использованием формата в arg
*/ void printnv(Nameval *p, void *arg) {
char *fmt;
 fmt = (char *) arg; printf(fmt,
 p->name, p->value); }
```

тогда вызывать мы ее будем так:

```
     apply(nvlist, printnv, "%s: %x\n");
```

Для подсчета количества элементов мы определяем функцию, параметром которой будет указатель на увеличиваемый счетчик:

```
    /* inccounter: увеличить счетчик *arg
*/ void inccounter(Nameval
 *p, void *arg) {
 int *ip;
/*' p не используется
 */ ip = (int *) arg; (*ip)++; }
```

Вызывается она следующим образом:

```
    int n;
n = 0;
 apply(nvlist, inccounter, &n);
printf("В nvlist %d элементов\n",
 n);
```

Не каждую операцию над списками удобно выполнять таким образом. Например, при удалении списка надо действовать более аккуратно:

```
    /* freeall:
     освободить все
 элементы списка listp */
 void freeall(Nameval *listp)
{
Nameval *next;
for ( ; listp != NULL; listp = next)
{ next = listp->next; /* считаем, что
 память, занятая строкой name,
 освобождена где-то в
 другом месте */ free(listp);
 }
 }
```

Память нельзя использовать после того, как мы ее освободили, поэтому до освобождения элемента, на который указывает listp, указатель listp->next нужно сохранить в локальной переменной next. Если бы цикл, как и раньше, выглядел так:

```
    ? for ( ; listp != NULL; listp = listp->next) ?
free(listp);
```

то значение listp->next могло быть затерто вызовом free и код бы не работал.

Заметьте, что функция freeall не освобождает память, выделенную под строку listp->name. Это подразумевает, что поле name каждого элемента типа Nameval было освобождено где-то еще либо память под него не была выделена. Чтобы обеспечить корректное выделение памяти под элементы и ее освобождение, нужно согласование работы newitem и f ree-all; это некий компромисс между гарантиями того, что память будет освобождена, и того, что ничего лишнего освобождено не будет. Именно здесь при неграмотной реализации часто возникают ошибки. В других языках, включая Java, данную проблему за вас решает сборка мусора. К теме управления ресурсами мы еще вернемся в главе 4.

Удаление одного элемента из списка — более сложный процесс, чем добавление:

```
    /* delitem: удалить первое вхождение
"name" в listp */
 Nameval *delitem(Nameval *listp, char
*name)
 {
 Nameval *p, *prev;
prev = NULL;
for (p = listp; p != NULL; p = p->next)
 { if (strcmp(name, p->name) == 0)
{ if (prev == NULL)
 listp = p->next; else
 prev->next = p->next; f ree(p);
return listp; }
prev = p; } eprintf("delitem: %s в
 списке отсутствует",
name);

     return NULL;
/* сюда не дойдет */
 }
```

Как и в f reeall, delitem не освобождает память, занятую полем name.

Функция eprintf выводит сообщение об ошибке и завершает программу, что в лучшем случае неуклюже. Грамотное восстановление после произошедших ошибок может быть весьма трудным и требует долгого обсуждения, которое мы отложим до главы 4, где покажем также реализацию eprintf.

Представленные основные списочные структуры и операции применимы в подавляющем большинстве случаев, которые могут встретиться в ваших программах. Однако есть много альтернатив. Некоторые библиотеки, включая библиотеку стандартных шаблонов (Standard Template Library, STL) в C++, поддерживают двухсвязные списки (double-linked lists: списки с двойными связями), в которых у каждого элемента есть два указателя: один — на последующий, а другой — на предыдущий элемент. Двухсвязные списки требуют больше ресурсов, но поиск последнего элемента и удаление текущего — операции порядка O(1). Иногда память под указатели списка выделяют отдельно от данных, которые они связывают;

такие списки несколько труднее использовать, но зато одни и те же элементы могут встречаться более чем в одном списке одновременно.

Кроме того, что списки годятся для ситуации, когда происходят удаления и вставки элементов в середине, они также хороши для управления данными меняющегося размера, особенно когда доступ к ним происходит по принципу стека: последним вошел, первым вышел (last in, first out — LIFO). Они используют память эффективнее, чем массивы, при наличии нескольких стеков, которые независимо друг от друга растут и уменьшаются. Они также хороши в случае, когда информация внутренне связана в цепочку неизвестного заранее размера, например как последовательность слов в документе. Однако если вам нужны как частые обновления, так и случайный доступ к данным, то разумнее будет использовать не такую непреклонно линейную структуру данных, а что-нибудь вроде дерева или хэш-таблицы.

Упражнение 2-7

Реализуйте некоторые другие операции над списком: копирование, слияние, разделение списка, вставку до или после указанного элемента. Как эти две операции вставки отличаются по сложности? Много ли вы можете использовать из того, что мы написали, и много ли вам надо написать самому?

Упражнение 2-8

Напишите рекурсивную и итеративную версии процедуры reverse, переворачивающей список. Не создавайте новых элементов списка; используйте старые.

Упражнение 2-9

Напишите обобщенный тип List для языка C. Простейший способ — в каждом элементе списка хранить указатель void *, который ссылается на данные. Сделайте то же для C++, используя шаблон (template), и для Java, определив класс, содержащий списки типа Obj ect. Каковы сильные и слабые стороны этих языков с точки зрения данной задачи?

Упражнение 2-10

Придумайте и реализуйте набор тестов для проверки того, что написанные вами процедуры работы со списками корректны. Стратегии тестирования подробнее обсуждаются в главе 6.

Деревья

Дерево — иерархическая структура данных, хранящая набор элементов. Каждый элемент имеет значение и может указывать на ноль или более других элементов. На каждый элемент указывает только один другой элемент. Единственным исключением является корень дерева, на который не указывает ни один элемент. Входящие в дерево элементы называются его вершинами.

Есть много типов деревьев, которые отражают сложные структуры, например деревья синтаксического разбора (parse trees), хранящие синтаксис предложения или программы, либо генеалогические деревья, описывающие родственные связи. Мы продемонстрируем основные принципы на деревьях двоичного поиска, в которых

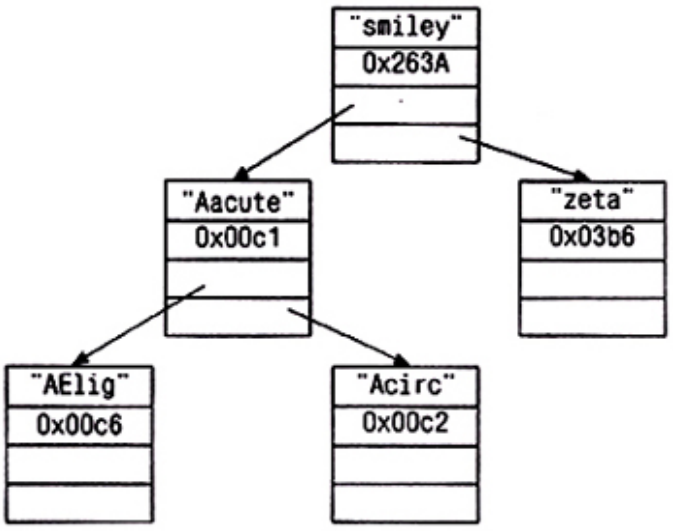
каждая вершина (node) имеет по две связи. Они наиболее просто реализуются и демонстрируют наиболее важные свойства деревьев. Вершина в двоичном дереве имеет значение и два указателя, left и right, которые показывают на его дочерние вершины. Эти указатели могут быть равны null, если у вершины меньше двух дочерних вершин. Структуру двоичного дерева поиска определяют значения в его вершинах: все дочерние вершины, расположенные левее данной, имеют меньшие значения, а все дочерние вершины правее — большие. Благодаря этому свойству мы можем использовать разновидность двоичного поиска для быстрого поиска значения по дереву или определения, что такого значения нет.

Вариант структуры Nameval для дерева пишется сразу:

```
    typedef struct Namevai
Nameval; struct Nameval {
char *name;
int value;
Nameval *left; /* меньшие */
Nameval * right; /* большие */
 };
```

Комментарии "большие" и "меньшие" относятся к свойствам связей: левые "дети" хранят меньшие значения, правые — большие.

В качестве конкретного примера на приведенном рисунке показано подмножество таблицы символов в виде дерева двоичного поиска для структур Nameval, отсортированных по ASCII-значениям имен символов.



Поскольку в каждой вершине дерева хранится несколько указателей на другие элементы, то многие операции, занимающие время порядка O(n) в списках или массивах, занимают только 0(log n) в деревьях. Наличие нескольких указателей в каждой вершине сокращает время выполнения операций, так как уменьшается количество вершин, которые необходимо посетить, чтобы добраться до нужной.

Дерево двоичного поиска (которое в данном параграфе мы будем называть просто "деревом") достраивается рекурсивным спуском по дереву; на каждом шаге спуска выбирается соответствующая правая или левая ветка, пока не найдется место для вставки новой вершины, которая должна быть корректно инициализированной

структурой типа Nameval: имя, значение и два нулевых указателя. Новая вершина добавляется как лист дерева, то есть у него пока отсутствуют дочерние вершины.

```
    /* insert: вставляет вершину
newp в дерево treep,
возвращает treep */
Nameval *insert(Nameval *treep,
 Nameval *newp) {
int cmp;
 if (treep == NULL)
 return newp;
cmp = strcmp(newp->name,~treep->name);
 if (cmp == 0)
 weprintf("insert: дубликат значения
 %s проигнорирован",
newp->name);
else if (cmp < 0)
treep->left = *insert(treep->left, newp);
else
treep->right = insert(treep->right, newp);
 return treep; }
```

Мы ничего еще не сказали о дубликатах — повторах значений. Данная версия insert сообщает о попытке вставки в дерево дубликата (cmp == 0). Процедура вставки в список ничего не сообщала, поскольку для обнаружения дубликата надо было бы искать его по всему списку, в результате чего вставка происходила бы за время O(n), а не за 0(1). С деревьями, однако, эта проверка оставляется на произвол программиста, правда, свойства структуры данных не будут столь четко определены, если будут встречаться дубликаты. В других приложениях может оказаться необходимым допускать дубликаты или, наоборот, обязательно их игнорировать.

Процедура weprintf — это вариант eprintf; она печатает сообщение, начинающееся со слова warning (предупреждение), но, в отличие от eprintf, работу программы не завершает.

Дерево, в котором каждый путь от корня до любого листа имеет примерно одну и ту же длину, называется сбалансированным. Преимущество сбалансированного дерева в том, что поиск элемента в нем занимает время порядка 0(log n), поскольку, как и в двоичном поиске, на каждом шаге отбрасывается половина вариантов.

Если элементы вставляются в дерево в том же порядке, в каком они появляются, то дерево может оказаться несбалансированным или даже весьма плохо сбалансированным. Например, если элементы приходят уже в отсортированном виде, то код каждый раз будет спускаться на еще одну ветку дерева, создавая в результате список из правых ссылок, со всеми "прелестями" производительности списка. Если же элементы поступают в произвольном порядке, то описанная ситуация вряд ли произойдет и дерево будет более или менее сбалансированным.

Трудно реализовать деревья, которые гарантированно сбалансированы; это одна из причин существования большого числа различных видов деревьев. Мы просто обойдем данный вопрос стороной и будем считать, что входные данные достаточно случайны, чтобы дерево было достаточно сбалансированным.

Код для функции поиска похож на функцию insert:

```
    /* lookup: поиск имени name
в дереве treep
*/ Nameval *lookup(Nameval *treep,
char *name) {
int cmp;
 if (treep == NULL)
return NULL;
cmp = strcmp(name, treep->name);
 if (cmp == 0)
return treep; else if (cmp < 0)
return lookup(treep->left, name); else
return lookup(treep->right, name); }
```

У нас есть еще пара замечаний по поводу функций lookup и i nse rt. Во-первых, они выглядят поразительно похожими на алгоритм двоичного поиска из начала главы. Это вовсе не случайно, так как они построены на той же идее "разделяй и властвуй", что и двоичный поиск, — основе логарифмических алгоритмов.

Во-вторых, эти процедуры рекурсивны. Если их переписать итеративно, то они будут похожи на двоичный поиск еще больше. На самом деле итеративная версия функции lookup может быть создана в результате элегантной трансформации рекурсивной версии. Если мы еще не нашли элемента, то последнее действие функции заключается в возврате результата, получающегося при вызове себя самой, такая ситуация называется хвостовой рекурсией (tail recursion). Эта конструкция может быть преобразована в итеративную форму, если подправить аргументы и стартовать функцию заново. Наиболее прямой метод — использовать оператор перехода goto, но цикл while выглядит чище:

```
    /* nrlookup: нерекурсивный поиск имени
в дереве treep */ *Nameval
 *nrlookup
 (Nameval *treep, char *name)
 int cmp;
 while (treep != NULL) {
cmp = strcmp(name, treep->name);
if (cmp == 0)
 return treep; else if (cmp < 0)
 treep = treep->left; else
 treep = treep->right; }
return NULL;
}
```

После того как мы научились перемещаться по дереву, остальные стандартные операции реализуются совершенно естественно. Мы можем использовать технологии управления списками, например написать общую процедуру обхода дерева, которая вызывает заданную функцию для каждой вершины. Однако в этом случае нужно сделать выбор: когда выполнять операцию над данной вершиной, а когда обрабатывать оставшееся дерево? Ответ зависит от того, что представляет собой дерево: если оно хранит упорядоченные данные, как в дереве двоичного поиска, то мы сначала посещаем левую часть, а затем уже правую.

Иногда структура дерева отражает какое-то внутреннее упорядочение данных, как в генеалогических деревьях, и порядок обхода листьев будет зависеть от отношений, которые дерево представляет.

Фланговый порядок (in-order) обхода выполняет операцию в данной вершине после просмотра ее левого поддерева и перед просмотром правого:

```
    /* applyinorder: применение
функции fn во время
обхода дерева во фланговом порядке
*/ void applyinorder(Nameval *treep,
 void (*fn)(Nameval*, void*), void *arg)
 {
if (treep == NULL)
return;
 applyinorder(treep->left, fn, arg);
 (*fn)(treep, arg);
 applyinorder(treep->right, fn, arg);
 }
```

Эта последовательность действий используется, когда вершины должны обходиться в порядке сортировки, например для печати их по порядку, что можно сделать так:

```
     applyinorder(treep, printnv, "%s: %x\n");
```

Сразу намечается вариант сортировки: вставляем элементы в дерево, выделяем память под массив соответствующего размера, а затем используем фланговый обход для последовательного размещения их в массиве. Восходящий порядок (post-order) обхода вызывает операцию для данной вершины после просмотра вершин-детей:

```
    * applypostorder: применение
функции fn во время
обхода дерева в восходящем порядке
*/ void applypostorder(Nameval *treep,
 void (*fn)(Nameval*, void*), void *arg) {
 if (treep == NULL)
 return;
applypostorder(treep->left, fn, arg);
applypostorder(treep->right, fn, arg);
 (*fn)(treep, arg);
}
```

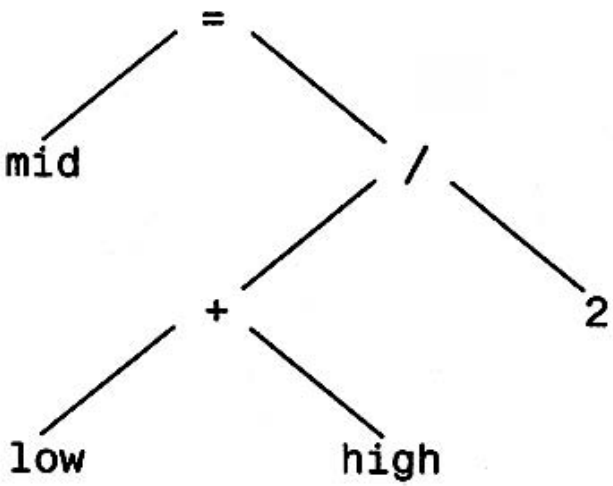
Восходящий порядок применяется, когда операция для вершины зависит от поддеревьев. Примерами служат вычисление высоты дерева (взять максимум из высот двух поддеревьев и добавить единицу), обработка дерева в графическом пакете (выделить место на странице для каждого поддерева и объединить их в области для данной вершины), а также вычисление общего занимаемого места.

Нисходящий порядок (pre-order) используется редко, так что мы не будем его рассматривать.

На самом деле деревья двоичного поиска применяются редко, хотя B-деревья, обычно сильно разветвленные, используются для хранения информации на внешних носителях. В каждодневном программировании деревья часто используются для представления структур действий и выражений. Например, действие

```
     mid = (low + high) / 2;
```

может быть представлено в виде дерева синтаксического разбора (parse tree), показанного на рисунке ниже. Для вычисления значения дерева нужно обойти его в восходящем порядке и выполнить в каждой вершине соответствующее действие.



Более детально мы будем рассматривать деревья синтаксического разбора в главе 9.

Упражнение 2-11

Сравните быстродействие lookup и nrlookup. Насколько рекурсия медленнее итеративной версии?

Упражнение 2-12

Используйте фланговый обход для создания процедуры сортировки. Какова ее временная сложность? При каких условиях она может работать медленно? Как ее производительность соотносится с нашей процедурой quicksort и с библиотечной версией?

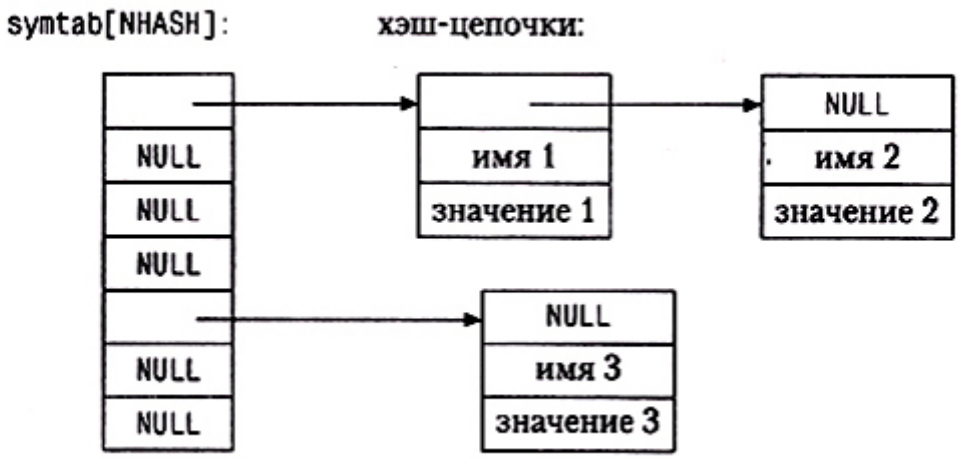
Упражнение 2-13

Придумайте и реализуйте набор тестов, удостоверяющих, что процедуры работы с деревьями корректны.

Хэш-таблицы

Хэш-таблицы (hash tables) — одно из величайших изобретений информатики. Сочетание массивов и списков с небольшой добавкой математики позволило создать эффективную структуру для хранения и* получения динамических данных. Типичное применение хэш-таблиц -символьная таблица, которая ассоциирует некоторое значение (данные) с каждым членом динамического набора строк (ключей). Ваш любимый компилятор практически наверняка использует хэш-таблицу для управления информацией о переменных в вашей программе. Ваш web-браузер наверняка использует хэш-таблицу для хранения адресов страниц, которые вы недавно посещали, а при соединении вашего компьютера с Интернетом, вероятно, она применяется для оперативного хранения (cache — кэширования) недавно использованных доменных имен и их IP-адресов.

Идея состоит в том, чтобы пропустить ключ через хэш-функцию (hash function) для получения хэш-значения (hash value), которое было бы равномерно распределено по диапазону целых чисел приемлемого размера. Это хэш-значение используется как индекс в таблице, где хранится информация. Java предоставляет стандартный интерфейс к хэш-таб-лицам. В C и C++ обычно с каждым хэш-значением (или "bucket" -"корзиной") ассоциируется список элементов, которые обладают этим значением, как показано на следующем рисунке:



На практике хэш-функция предопределена, а массив соответствующего размера выделяется нередко даже во время компиляции. Каждый элемент массива — это список, который сцепляет вместе элементы, имеющие общее хэш-значение. Другими словами, хэш-таблица из п элементов — это массив списков, средняя длина которых равна п/'(размер_массива). Получение элемента является константной (O(1)) операцией, если мы взяли хорошую хэш-функцию и списки не становятся слишком большими.

Поскольку хэш-таблица — массив списков, то тип элементов для нее такой же, как и для списка:

```
    typedef struct Nameval
Nameval; struct Nameval {
char *name;
int value;
Nameval *next; /* следующий в цепи */ };
Nameval *symtab[NHASH]; /*
таблица символов */
```

Способы работы со списками из раздела 2.7 можно использовать для управления отдельными хэш-цепочками. Как только у вас есть хорошая хэш-функция, все становится легко: получаете цепочку и дальше спокойно проходите вдоль списка, ища подходящий элемент. Приведем текст процедуры поиска/вставки в хэш-таблицу. Если элемент найден, то он возвращается. Если элемент не найден и задан флаг создания create, то lookup добавляет элемент в таблицу. Копия имени символа не создается — считается, что вызывающая сторона сама сделала себе надежную копию.

```
    /* lookup: поиск имени в таблице;
возможна вставка */ Nameval* lookup
 (char *name, int create, int value)
 {
```

```
int h;
Nameval *sym;
h = hash(name);
 for (sym =-symtab[h];
sym != NULL; sym = sym->next)
 if (strcmp(name, sym->name) == 0)
 return sym; if (create) {
sym = (Nameval *)
emalloc(sizeof(Nameval));


sym->name = name; /*
считаем, что память
 уже выделена */ sym->value = value;
sym->next = symtab[h]; symtab[h] = sym;
}
 return sym;
 }
```

Поиск и возможная вставка комбинируются часто. Иначе приходится прилагать лишние усилия. Если писать

```
    if (lookup("name") == NULL)
 additem(newitem("name", value));
```

то хэш-функция вычисляется дважды.

Насколько большим должен быть массив? Основная идея заключается в том, что он должен быть достаточно большим, чтобы любая хэш-цепочка была бы длиной всего в несколько элементов и поиск занимал время O(1). Например, у компилятора размер массива должен быть порядка нескольких тысяч, так как в большом исходном файле — несколько тысяч строчек и вряд ли различных идентификаторов имеется больше, чем по одному на строчку кода.

Теперь нам нужно решить, что же наша хэш-функция, hash, будет вычислять. Она должна быть детерминированной, достаточно быстрой и распределять данные по массиву равномерно. Один из наиболее распространенных алгоритмов хэширования для строк получает хэш-значение, добавляя каждый байт строки к произведению предыдущего значения на некий фиксированный множитель (хэш). Умножение распределяет биты из нового байта по всему до сих пор не считанному значению, так что в конце цикла мы получим хорошую смесь входных байтов. Эмпирически установлено, что значения 31 и 37 являются хорошими множителями в хэш-функции для строк ASCII.

```
    enum { MULTIPLIER = 31 };
 /* hash: вычислить
хэш-функцию строки */
unsigned int hash(char *str)
 {
unsigned int h;
 unsigned char *p;


h = 0;
 for (p = (unsigned char *)
str; *p != '\0'; p++)
```

```
 h = MULTIPLIER * h +
  *p; return h % NHASH; }
```

В вычислениях символы принимаются неотрицательными принудительно (тем, что используется тип unsigned char), так как ни в C, ни в C++ наличие знака у символов не регламентировано, а мы хотим, чтобы наша хэш-функция оставалась положительной.

Хэш-функция возвращает результат по модулю размера массива. Если хэш-функция распределяет данные равномерно, то точный размер массива неважен. Трудно, однако, гарантировать, что хэш-функция независима от размера массива, и даже у хорошей функции могут быть проблемы с некоторыми наборами входных данных. Поэтому имеет смысл сделать размер массива простым числом, чтобы слегка подстраховаться, обеспечив отсутствие общих делителей у размера массива, хэш-мультипликатора и, возможно, значений данных.

Эксперименты показывают, что для большого числа разных строк трудно придумать хэш-функцию, которая работала бы гораздо лучше, чем эта, зато легко придумать такую, которая работает хуже. В ранних версиях Java была хэш-функция для строк, работавшая более эффективно, если строка была длинной. Хэш-функция работала быстрее, анализируя только 8 или 9 символов через равные интервалы в строках длиннее, чем 16 символов, начиная с первого символа. К сожалению, хотя хэш-функция работала быстрее, у нее были очень плохие статистические показатели, что сводило на нет выигрыш в производительности. Пропуская части строки, она нередко выкидывала именно различающиеся части строк. Имена файлов начинаются с длинных идентичных префиксов — имен каталогов — и могут отличаться только несколькими последними символами (например, .Java и .class). Адреса в Internet обычно начинаются с http://www. и заканчиваются на . html, поэтому все различия у них в середине. Хэш-функция частенько проходила только по неразличающейся части в имени, в результате чего образовывались более длинные хэш-цепочки, что замедляло поиск. Проблема была решена заменой хэш-функции на эквивалентную той, что мы привели выше (с мультипликатором 37), которая исследует каждый символ в строке.

Хэш-функция, которая неплохо подходит для одного набора данных (например, имен переменных), может плохо подходить для другого (адреса Internet), так что потенциальная хэш-функция должна быть протестирована на разных наборах типичных входных данных. Хорошо ли она перемешивает короткие строки? Длинные строки? Строки одинаковой длины с небольшими изменениями?

Хэшировать можно не только строки. Можно "смешать" три координаты частицы при физическом моделировании, тем самым уменьшив размер хранилища до линейной таблицы (0(количество точек)) вместо трехмерного массива (0(размер_по_х хразмер_nojj X размер_no_z)).

Один примечательный случай использования хэширования — программа Джерарда Холзманна (Gerard Holzmann) для анализа протоколов и параллельных систем Supertrace. Supertrace берет полную информацию о каждом возможном состоянии наблюдаемой системы и хэширует эту информацию для получения адреса единственного бита в памяти. Если этот бит установлен, то данное состояние наблюдалось и раньше; если нет, то не наблюдалось. В Supertrace используется хэш-таблица длиной во много мегабайт, однако в каждой ячейке хранится только один бит. Цепочки не строятся; если два разных состояния совпали по своей хэш-функции, то программа этого не заметит. Supertrace рассчитана на то, что вероятность коллизии мала (она не обязана быть нулем, поскольку Supertrace

использует вероятностные, а не детерминированные вычисления). Поэтому хэш-функция здесь очень аккуратна; она использует циклический избыточный код (cyclic redundancy check — CRC) — функцию, которая тщательно перемешивает данные.

Хэш-таблицы незаменимы для символьных таблиц, поскольку они предоставляют (почти всегда) доступ к каждому элементу за константное время. У них есть свои ограничения. Если хэш-функция плоха или размер массива слишком мал, то списки могут стать достаточно длинными. Поскольку списки не отсортированы, это приведет к линейному доступу (0(и)). Элементы нельзя напрямую получить в отсортированном виде, однако их легко подсчитать, выделить память под массив, заполнить его указателями на элементы и отсортировать их. Что и говорить, константное время операций поиска, вставка и удаление — это такое свойство хэш-таблицы, которого не достичь никакими другими технологиями.

Упражнение 2-14

Наша хэш-функция замечательна для повседневного хэширования строк. Однако на нарочно придуманных данных она может работать плохо. Сконструируйте набор данных, приводящий к плохому поведению нашей хэш-функции. Проще ли найти плохой набор для других значений NHASH?

Упражнение 2-15

Напишите функцию для доступа к последовательным элементам хэш-таблицы в несортированном порядке.

Упражнение 2-16

Измените функцию lookup так, что если средняя длина списка превысит некое значение x, то массив автоматически расширяется в y раз и хэш-таблица перестраивается.

Упражнение 2-17

Спроектируйте кэш-функцию для хранения координат точек в двумерном пространстве. Насколько легко ваша функция адаптируется к изменениям в типе координат, например при переходе от целого типа данных к значениям с плавающей точкой, или при переходе от декартовой к полярной системе координат, или при увеличении количества измерений?

Заключение

При выборе алгоритма нужно сделать несколько шагов. Во-первых, следует изучить существующие алгоритмы и структуры данных. Подумайте, какой объем данных может обработать программа. Если задача предполагает скромные размеры данных, то выбирайте простые технологии; если количество данных может расти, то исключите решения, плохо приспосабливающиеся к этому росту. Там, где возможно, используйте библиотеку или специальные средства языка. Если ничего готового нет, то напишите или достаньте короткую, простую, понятную реализацию. Попробуйте ее в действии. Если измерения показывают, что она слишком медленная, только тогда вам стоит перейти к более продвинутым технологиям.

Хотя есть много структур данных, часть которых просто необходима для приемлемой производительности в определенных условиях, большинство программ основано на

массивах, списках, деревьях и хэш-таб-лицах. Каждая из этих структур поддерживает набор операций-примитивов, обычно включающий в себя: создание нового элемента, поиск элемента, добавление элемента куда-либо, возможно, удаление элемента и применение некоторой операции ко всем элементам.

У каждой операции есть ожидаемое время выполнения, которое часто определяет, насколько выбранный тип данных или его реализация подходит для конкретного приложения. Массивы предоставляют доступ к элементам за константное время, но зато плохо изменяют свои размер; Списки хорошо приспособлены для вставки и удаления, но случайный доступ к элементам происходит лишь за линейное время. Деревья и хэш-таблицы предоставляют разумный компромисс: быстрый доступ к заданным элементам в сочетании с легкой расширяемостью, пока в их структуре соблюдается определенный баланс.

Есть еще множество изощренных структур данных для специальных задач, однако этот базовый набор вполне достаточен для написания подавляющего большинства программ

Дополнительная литература

Серия книг "Алгоритмы" Боба Седжвика (Bob Sedgewick. Algorithms. Addison-Wesley) содержит доступные сведения о большом числе полезных алгоритмов. В третьем издании "Алгоритмов на C++" (Algorithms in C++, 1998) идет неплохое обсуждение хэш-функций и размеров хэш-таблиц. "Искусство программирования" Дона Кнута (Don Knuth. The Art of Computer Programming. Addison-Wesley) — первейший источник для строгого анализа многих алгоритмов; в третьем томе (2nd ed., 19981) рассматриваются поиск и сортировка.

Программа Supertrace описана в книге Джерарда Холзманна "Дизайн и проверка протоколов" (Gerard Holzmann. Design and Validation of Computer Protocols. Prentice Hall, 1991).

Ион Бентли и Дуг Мак-Илрой описывают создание скоростной и устойчивой версии быстрой сортировки в статье "Конструирование функции сортировки" (Jon Bentley, Doug Mcllroy. Engineering a sort function. Software - Practice and Experience, 23, 1, p. 1249-1265, 1993).

Проектирование и реализация

- Алгоритм цепей Маркова
- Варианты структуры данных
- Создание структуры данных в языке C
- Генерация вывода
- Java
- C++
- Awk и Perl
- Производительность
- Уроки
- Дополнительная литература

> Покажите мне свои блок-схемы и спрячьте таблицы, и я ничего не пойму. Покажите мне таблицы, и блок-схемы мне не понадобятся — все будет очевидно и так.
>
> Фредерик П. Брукс-мл. Мифический человекомесяц

Согласно приведенной цитате из классической книги Брукса, проектирование структур данных — центральный момент в создании программы. После того как структуры данных определены, алгоритмы, как правило, стремятся сами встать на свое место, и кодирование становится относительно простым делом.

Это, конечно, слишком упрощенный, но тем не менее верный взгляд. В предыдущей главе мы разобрали основные структуры данных; они являются строительными блоками для большинства программ. В этой главе мы скомбинируем рассмотренные структуры, спроектировав и реализовав небольшую программу. Мы покажем, насколько решаемая проблема влияет на структуры данных и насколько очевидным становится написание кода после того, как определены используемые структуры данных.

Одним из аспектов этой точки зрения является то, что выбор конкретного языка программирования оказывается сравнительно неважным для общего проектирования. Мы сначала спроектируем программу абстрактно, а потом реализуем ее на C, Java, C++, Awk и Perl. Сравнив реализации, мы увидим, как тот или иной язык может облегчать или, наоборот, затруднять кодирование и в каких аспектах выбор языка не является важным. Выбранный для реализации язык может, конечно, чем-то украсить программу, но не доминирует в ее разработке.

Проблема, которую мы будем решать, необычна, однако в общем виде она типична для большинства программ: некие данные поступают в программу, некие данные программа выдает на выходе, а обработка данных требует некоторого мастерства.

В данном конкретном случае мы собираемся генерировать случайный английский текст, который был бы читабелен. Если мы будем выдавать просто случайным образом выбранные буквы или слова, получится, естественно, полная чепуха. Программа, случайным образом выбирающая буквы (и пробелы — для разделения "слов"), выдавала бы что-нибудь вроде

xptmxgn xusaja afqnzgxl lhidlwcd
rjdjuvpydrlwnjy

что, естественно, не слишком нас устраивает. Если присвоить буквам вес, соответствующий частоте их появления в нормальном тексте, мы получим что-нибудь такое:

idtefoae tcs trder jcii ofdslnqetacp t ola

что звучит не лучше. Набор слов, выбранных случайным образом из словаря, тоже не будет иметь особого смысла:

polydactyl equatorial splashily jowl verandah
circumscribe

Для того чтобы получить более сносный результат, нам нужна статистическая модель с более утонченной структурой. Например, можно рассматривать частоту вхождения целых фраз. Но где нам взять такую статистику?

Мы могли бы взять большой отрывок английского текста и детально изучить его, но есть более простой и более занимательный способ. Главная суть его состоит в том, что мы можем использовать любой кусок текста для построения статистической модели языка, используемого в этом тексте, и генерировать случайный текст, имеющий статистику, схожую с оригиналом.

Алгоритм цепей Маркова

Элегантный способ выполнить подобную обработку - использовать технику, известную как алгоритм цепей Маркова. Ввод можно представить себе как последовательность перекрывающихся фраз; алгоритм разделяет каждую фразу на две части: префикс, состоящий из нескольких слов, и следующее за ним слово — суффикс (или окончание). Алгоритм цепей Маркова создает выходные фразы, выбирая случайным образом суффикс, следующий за префиксом; все это в соответствии со статистикой текста-оригинала (в нашем случае). Хорошо выглядят фразы из трех слов, когда префикс из двух слов используется для подбора слова-суффикса:

```
присвоить w, и w2 значения двух первых слов текста
печатать W-, и w2
цикл:
случайным образом выбрать w3 из слов,
следующих за префиксом w, w2 в тексте
печатать w3
заменить да/ и w2 на w2 и w-s
повторить цикл
```

Для иллюстрации сгенерируем случайный текст, основываясь на нескольких предложениях из эпиграфа к этой главе и используя префикс из двух слов:

Show your flowcharts and conceal your tables
and I will be mystified.
Show your tables and your flowcharts
will be obvious, (end)

Вот несколько пар слов, взятых из этого отрывка, и слова, которые следуют за ними:

Префикс	
Show your	
your flowcharts	
flowcharts and	
flowcharts will	
your tables	
will be	
be mystified	
be obvious	

Подходящие суффиксы	
flowcharts	tables
and	will
conceal	
be	
and	and
mystified,	obvious.
Show	
(end)	

Обработка этого текста по предлагаемому алгоритму markov' начинается с того, что будет напечатано Show your, после чего случайным образом будет выбрано или flowcharts, или tables. Если будет выбрано первое слово, то текущим префиксом станет your flowcharts, а следующим словом будет выбрано and или will. Если же выбранным окажется tables, то после него последует слово and. Так будет продолжаться до тех пор, пока не будет сгенерирована фраза заданного размера или в качестве суффикса не будет выбрано слово-метка конца ввода (end).

Наша программа прочтет отрывок английского текста и использует алгоритм markov для генерации нового текста, основываясь на частотах вхождения фраз фиксированной длины. Количество слов в префиксе, которое в разобранном примере равно двум, в нашей программе будет параметром. Если префикс укоротить, текст будет менее логичным, если длину префикса увеличить, наше творение будет походить на дословный пересказ вводимого текста. Для английского текста использование двух слов для выбора третьего дает разумный компромисс: сохраняется стиль прототипа и привносится достаточно своеобразия.

Что такое слово? Очевидный ответ — последовательность символов алфавита, однако нам было бы желательно сохранить и пунктуационные различия, то есть различать "words" и "words.". Приписывание знаков препинания к словам повышает качество генерируемого текста, вводя в него пунктуацию, а следовательно (косвенным образом), и грамматику, влияет на выбор слов; правда, при этом в текст могут просочиться несбалансированные разрозненные скобки и кавычки.Таким образом, мы определим "слово" как нечто, ограниченное с двух сторон пробелами, — при этом получится, что нет ограничений на используемый язык, а знаки пунктуации привязаны к словам. Поскольку в большинстве языков

программирования имеются средства, позволяющие разбить текст на слова, разделенные пробелами, воплотить задуманное будет несложно.

Исходя из выбранного метода можно сказать, что все слова, фразы из двух слов и фразы из трех слов должны присутствовать во вводимом тексте, но появятся новые фразы из четырех и более слов. Ниже приведены несколько предложений, сгенерированных программой, разработке которой посвящена данная глава, полученных на основе текста седьмой главы книги "И восходит солнце" Эрнеста Хемингуэя:

```
    As I started up the undershirt onto his chest black,
and big stomach
    muscles bulging under the light. "You see them?"
Below the line
    where his ribs stopped were two raised whate welts.
 "See on the forehead.""Oh,
    Brett, I love you.""Let's not talk. Talking's all bailge. I'm
    going away tomorrow"."Tomorrow?""Yes. Didn't I say
    so?I am". " Let's have a drink, them."
```

Здесь нам повезло - пунктуация оказалась корректной, но этого могло и не случиться.

Варианты структуры данных

На какой размер вводимого текста мы должны рассчитывать? Насколько быстро должна работать программа? Представляется логичным, чтобы программа была в состоянии считать целую книгу, так что нам надо быть готовыми к размеру ввода в n = 100 000 слов и более. Вывод должен составлять сотни, а возможно, и тысячи слов, а работать программа должна несколько секунд, но отнюдь не минут. Имея 100 000 слов вводимого текста, надо признать, что n получается достаточно большим, так что для того, чтобы программа работала действительно быстро, алгоритм придется писать довольно сложный.

Для того чтобы начать генерировать текст, алгоритм markov должен сначала просмотреть весе введенный фрагмент, поэтому исходный текст нам придется каким-то образом сохранять. Первая возможность — ввести полностью исходный текст и сохранить его как длинную строку, но нам явно нужно разбить его на отдельные слова. Если сохранить его как массив указателей на слова, генерация вывода будет происходить просто: для выбора нового слова надо просканировать введенный текст и посмотреть, какие существуют слова-суффиксы для только что введенного префикса, и выбрать из них случайным образом одно. Однако это будет означать сканирование всех 100 000 слов ввода для генерации каждого нового слова; при размере выводимого текста в 1000 слов необходимо осуществить сотни миллионов сравнений строк, а это вовсе не быстро.

Другая возможность — хранить только уникальные слова исходного текста вместе со списком, указывающим, где именно они появлялись в оригинале. В этом случае мы сможем находить устраивающие нас слова более быстро. Мы могли бы использовать хэш-таблицу вроде той, что обсуждалась в главе 2, однако та версия не соответствует специфическим требованиям алгоритма markov, для которого нам надо по заданному префиксу быстро находить все возможные суффиксы.

Нам нужна структура данных, которая бы более успешно представляла префикс и ассоциированные с ним суффиксы. Программа будет работать в два прохода: при проходе ввода будет создаваться структура данных, представляющая фразы, а при проходе вывода эта структура будет использоваться для случайной генерации текста. При обоих проходах нам надо будет отыскивать префикс (причем быстро отыскивать!): при первом — для обновления его суффиксов, а при втором — для случайного выбора одного из связанных с ним суффиксов. Из этих требований логично вырисовывается такой вид хэш-таблицы: ключами ее являются префиксы, а значениями -наборы (множества) суффиксов для соответствующего префикса.

Для целей описания мы зафиксируем длину префикса в два слова, так что каждое выводимое слово будет базироваться на двух предшествующих. Количество слов в префиксе не влияет на проектирование, и программа сможет обрабатывать префиксы с любым количеством слов, но выбрав это число сейчас, мы просто сделаем разговор более конкретным. Префикс и множество всех возможных для него суффиксов мы назовем состоянием (state), что является стандартным термином для алгоритмов, связанных с марковскими цепями.

Для каждого заданного префикса мы должны сохранить все употребляемые после него суффиксы, чтобы потом иметь возможность их использовать. Суффиксы не упорядочены и добавляются по одному зараз. Мы не знаем их количества заранее, поэтому нам потребуется структура] данных, которая могла бы увеличиваться легко и эффективно; такими структурами являются список и расширяемый массив. При генерации выходного текста мы должны иметь возможность выбрать случайным 3 образом один суффикс из всего множества суффиксов, возможных для конкретного префикса. Элементы никогда не удаляются.

Что делать, если фраза встречается более одного раза? Например, фраза "может появиться дважды" может появиться дважды, а фраза "может появиться единожды" — только единожды. В таком случае можно поместить слово "дважды" два раза в список суффиксов для префикса "может появиться" или один раз, но при этом установить соответствующий суффиксу счетчик в 2. Мы попробовали и со счетчиками, и без них; без счетчиков проще, поскольку при добавлении суффикса не надо проверять, нет ли его уже в списке. Эксперименты показали, что разница в скорости при обоих способах практически незаметна.

Итак, давайте подведем итоги. Каждое состояние включает в себя префикс и список суффиксов. Эта информация хранится в хэш-таблице, ключами которой являются префиксы. Каждый префикс представляет собой набор слов фиксированного размера. Если суффикс появляется более одного раза после данного префикса, то каждое новое появление отмечается еще одним включением этого суффикса в список.

Теперь следует решить, как представлять сами слова. Простейший способ — хранить их как отдельные строки. Поскольку в большинстве текстов много слов появляется более одного раза, для сохранения места лучше использовать еще одну хэш-таблицу — для отдельных слов, каждое из которых будет храниться лишь единожды. Это ускорит хэширование (помещение в хэш-таблицу) префиксов, поскольку мы сможем сравнивать указатели, а не отдельные символы: уникальные строки будут иметь уникальные адреса. Проектирование этой структуры мы оставим вам для самостоятельных упражнений; пока же строки будут храниться раздельно.

Создание структуры данных в языке С

Начнем с реализации на С. Для начала надо задать некоторые константы:

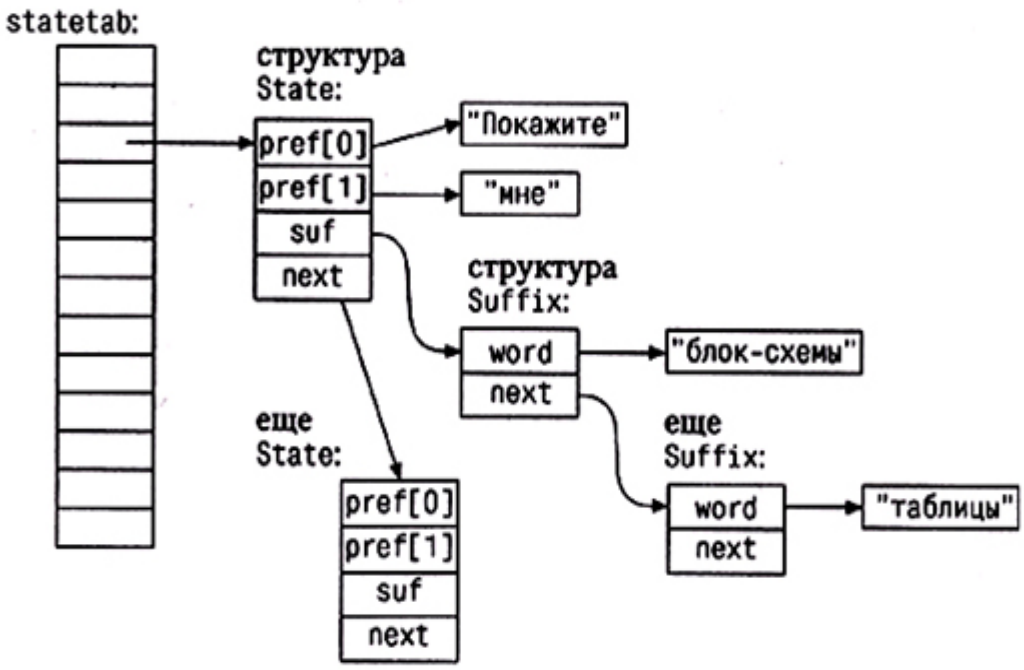
```
    enum {
 NPREF = 2,
 /* количество слов в префиксе */
NHASH = 4093,
 /* размер массива
 хэш-таблицы состояний */
MAXGEN = 10000
/* максимум генерируемых слов */
};
```

В этом описании определяются количество слов в префиксе (NPREF), размер массива хэш-таблицы (NHASH) и верхний предел количества генерируемых слов (MAXGEN). Если NPREF — константа времени компиляции, а не переменная времени исполнения, то управление хранением упрощается. Мы задали очень большой размер массива, поскольку программа должна быть способна воспринимать очень большие документы, возможно, даже целые книги. Мы выбрали NHASH = 4093, так что если во введенном . тексте имеется 10 000 различных префиксов (то есть пар слов), то среднестатистическая цепочка будет весьма короткой — два или три префикса. Чем больше размер, тем короче будет ожидаемая длина цепи и, следовательно, тем быстрее будет осуществляться поиск. Эта программа создается для развлечения, поэтому производительность ее не критична, однако если мы сделаем слишком маленький массив, то программа не сможет обработать текст ожидаемого размера за разумное время; если же сделать его слишком большим,'он может не уложиться в имеющуюся память.

Префикс можно хранить как массив слов. Элементы хэш-таблицы будут представлены как тип данных State, ассоциирующий список Suffix с префиксом:

```
    typedef struct State State;
typedef struct Suffix Suffix;
struct State { /* список префиксов +
суффиксы */ char *pref[NPREFJ; /*
слова префикса */ Suffix *suf; /* список суффиксов */
State *next; /* следующий в хэш-таблице */
 };
struct Suffix { /* список суффиксов */
char *word; /* суффикс */
Suffix *next; /* следующий суффикс
в списке */ };
State *statetab[NHASH]; /*
 хэш-таблица состояний */
```

Графически структура данных выглядит так:



Нам потребуется хэш-функция для префиксов, которые являются массивами строк. Нетрудно преобразовать хэш-функцию из главы 2, сделав цикл по строкам в массиве, таким образом хэшируя конкатенацию этих строк:

```
    /* hash: вычисляет хэш-значение
 для массива из NPREF строк */
 unsigned int hash(char *s[NPREF])
 {
  unsigned int h; unsigned char
 *p;
int i;
```

```
     /* hash: вычисляет хэш-значение
 для массива из NPREF строк */
 unsigned int hash(char *s[NPREF])
 {
  unsigned int h;
 unsigned char *p;
 int i;
```

Выполнив схожим образом модификацию алгоритма lookup, мы завершим реализацию хэш-таблицы:

```
    /* lookup: ищет префикс;
создает его при необходимости. */
 /* возвращает указатель, если
префикс существует или создан; *,
 /* NULL в противном случае. */
/* при создании не делает strdup,
 так что строки
 /* не должны изменяться после
 помещения в таблицу. */
State* lookup(char *prefix[NPREF],
```

```
 int create)
 {

     int i, h;
 State *sp;
 h = hash(prefix);
for (sp = statetabfh]; sp != NULL;
sp = sp->next)
{ for (i = 0; i,< NPREF; i++)
if (strcmp(prefix[i], sp->pref[i]) != 0)
 break;
if (i == NPREF) /* нашли'*/ return sp;
 I> if (create) { sp = (State *)
 emalloc(sizeof(State)); for
(i = 0; i < NPREF; i++)
 sp->pref[i] = prefix[i]; sp->suf = NULL;
sp->next = statetabfh];
statetab[h] = sp; }
 return sp; }
```

Обратите внимание на то, что lookup не создает копии входящей строки при создании нового состояния, просто в sp->pref [ ] 'сохраняются указатели. Те, кто использует вызов lookup, должны гарантировать, что впоследствии данные изменены не будут. Например, если строки находятся в буфере ввода-вывода, то перед тем, как вызвать lookup, надо сделать их копию; в противном случае следующие куски вводимого текста могут перезаписать данные, на которые ссылается хэш-таблица. Необходимость решать, кто же владеет совместно используемыми ресурсами, возникает достаточно часто. Эту тему мы подробно рассмотрим в следующей главе.

Теперь нам надо прочитать файл и создать хэш-таблицу:

```
    /* build; читает ввод,
создает таблицу префиксов */
 void build(char *prefix[NPREF],
FILE *f)
{
  char buf[100], fmt[10];
/* создать форматную строку:
 %s может переполнить buf */ sprintf(fmt,
"%%%ds", sizeof(buf)-1); while
 (fscanf(f, fmt, buf) != EOF)
  add(prefix, estrdup(buf)); }
```

Необычный вызов sprintf связан с досадной проблемой, присущей f scant, — если бы не эта проблема, функция f scanf идеально подошла бы для нашего случая. Все дело в том, что вызов fscanf с форматом %s считывает следующее ограниченное пробелами слово из файла в буфер, но без проверки его длины: слишком длинное слово может переполнить входной буфер, что приведет к разрушительным последствиям. Если буфер имеет размер 100 байтов (что гораздо больше того, что ожидаешь встретить в нормальном тексте), мы можем использовать формат %99s (оставляя один байт на заключительный ' \0'), что будет означать для fscanf приказ остановиться после 99 байт. Тогда длинное слово окажется разбитым на части, что некстати, но вполне безопасно. Мы могли бы написать

```
    ? enum { BUFSIZE = 100 };
? char fmtrj = "%99s"; /1 BUFSIZE-1 ./
```

однако это потребовало бы двух констант для одного, довольно произвольного значения — размера буфера; обе эти константы пришлось бы поддерживать в непротиворечивом состоянии. Проблему можно решить раз и навсегда, создавая форматную строку динамически — с помощью sprintf, и именно так мы и поступили.

Аргументы build — массив prefix, содержащий предыдущие NPREF введенных слов и указатель FILE. Массив prefix и копия введенного слова передаются в add, которая добавляет новый элемент в хэш-таблицу и обновляет префикс:

```
    /* add: добавляет слово в
 список суффиксов,
обновляет префикс */
 void add(char *prefix[NPREF],
 char *suffix) {
 State *sp;
 sp = lookup(prefix, 1);
 /* создать, если не найден
 */ addsuffix(sp, suffix); /*
сдвиг слов вниз в префиксе */
memmove(prefix, prefix+1,
 (NPREF-1)*sizeof(prefix[0]));
prefix[NPREF-1] = suffix;
 }
```

Вызов memmove — идиоматический способ удаления из массива. В префиксе элементы с 1 по NPREF-1 сдвигаются вниз на позиции с 0 по NPREF-2, удаляя первое слово и освобождая место для нового слова в конце.
Функция addsuf fix добавляет новый суффикс:

```
    /* addsuffix: добавляет
в состояние. */
 /* Суффикс впоследствии не
должен меняться */
void addsuffix
(State *sp, char *suffix)
{
 Suffix *suf;
 suf = (Suffix *) emalloc
 (sizeof(Suffix)); suf->word =
suffix; suf->next = sp->suf; sp->suf = suf;
}
```

Мы разделили процесс обновления состояния на две функции — add просто добавляет суффикс к префиксу, тогда как addsuffix выполняет тесно связанное с реализацией действие — добавляет слово в список суффиксов. Функция add вызывается из build, но addsuffix используется только изнутри add — это та часть кода, которая может быть впоследствии изменена, поэтому лучше выделить ее в отдельную функцию, даже если вызываться она будет лишь из одного места.

Генерация вывода

Теперь, когда структура данных построена, пора переходить к следующему шагу — генерации нового текста. Основная идея остается неизменной: у нас есть префикс, мы случайным образом выбираем один из возможных для него суффиксов, печатаем его, затем обновляем префикс. Это повторяющаяся часть обработки; нам еще надо продумать, как начинать и как заканчивать алгоритм. Начать будет нетрудно, если мы запомним слова первого префикса и начнем с них. Закончить алгоритм также нетрудно; для этого нам понадобится слово-маркер. Прочтя весь вводимый текст, мы можем добавить некий завершитель — "слово", которое с гарантией не встретится ни в одном тексте:

```
    build(prefix, stdin);
add(prefix, NONWORD);
```

В этом фрагменте NONWORD — некоторое значение, которое точно никогда не встретится в тексте. Поскольку по нашему определению слова разделяются пробелами, на роль завершителя подойдет "слово", равносильное пробелу, но отличное от него, например символ перевода строки:

```
    char NONWORD[] = "\n"; /* никогда не встретится */
```

Еще одна проблема — что делать, если вводимого текста недостаточно для запуска алгоритма? Для решения этой проблемы существуют два принципиальных подхода — либо прерывать работу программы, если введенного текста недостаточно, либо считать, что для генерации хватит любого фрагмента, и просто не утруждать себя проверкой. Для данной программы лучше выбрать второй подход.

Можно начать процесс генерации с создания фиктивного префикса, который даст гарантию, что для работы программы всегда хватит вводимого текста. Для начала можно инициализировать все значения массива префиксов словом NONWORD. Это даст дополнительное преимущество — первое слово во вводимом файле будет первым суффиксом нашего вымышленного префикса, так что в генерирующем цикле печатать надо будет только суффиксы.

На случай, если генерируемый текст окажется непредсказуемо большого размера, можно встроить ограничитель — прерывать алгоритм после вывода заданного количества слов или при появлении NONWORD в качестве суффикса.

Добавление нескольких NONWORD в концы структур данных значительно упрощает основные циклы программы — это хороший пример использования специальных значений для маркировки границ — сигнальных меток (sentinel).

Как правило, надо стараться обработать все отклонения, исключения и особые случаи непосредственно в данных. Код писать труднее, так что старайтесь добиться того, чтобы управляющая логика была как можно более проста и прямолинейна.

Функция generate использует алгоритм, который мы только что описали в общих словах. Она генерирует текст по слову в строке; эти слова можно группировать в более длинные строки при помощи любого текстового редактора — в главе 9 будет показано простое средство для такого форматирования — процедура f mt.

Благодаря использованию в начале и в конце строк NONWORD, generate начинает и заканчивает работу без проблем:

```
    /* generate: генерирует
вывод по одному слову в строке */
void generate(int nwords)
{
State *sp;
 Suffix *suf;
char *prefix[NPREF], *w;
int i, nmatch; »
 for (i = 0; i < NPREF; i++)
 /* начальные префиксы
 */ prefix[i] = NONWORD;
for (i = 0; i < nwords; i++)
{ sp = lookup(prefix, 0); nmatch =
 0; for (suf = sp->suf; suf != NULL; suf = suf->next)
 if (rand() % ++nmatch == 0)
/* prob = 1/nmatch */
 w = suf->word; if
 (strcmp(w, NONWORD) == 0)
 break;
printf("%s\n", w);
 memmove(prefix, prefix+1,
(NPREF-1)*sizeof(prefix[0]));
 prefix[NPREF-1] = w; } }
```

Обратите внимание на алгоритм случайного выбора элемента, когда число всех элементов нам неизвестно. В переменной nmatch подсчитывается количество совпадений при сканировании списка. Выражение

```
    rand()
 ++nmatch == 0
```

увеличивает nmatch и является истинным с вероятностью 1/nmatch. Таким образом, первый подходящий элемент будет выбран с вероятностью 1, второй заменит его с вероятностью 1/2, третий заменит выбранный из предыдущих с вероятностью 1/3 и т. п. В каждый момент времени каждый из k просмотренных элементов будет выбран с вероятностью i/k.

Вначале мы устанавливаем prefix в стартовое значение, которое с гарантией присутствует в хэш-таблице. Первые найденные значения Suffix будут первыми словами документа, поскольку только они следуют за стартовым префиксом. После этого суффикс выбирается случайным образом. В цикле вызывается lookup для поиска в хэш-таблице элемента (множества суффиксов), соответствующего данному префиксу; после этого случайным образом выбирается один из суффиксов, он печатается, а префикс обновляется.

Если выбранный суффикс оказывается NONWORD, то все заканчивается, поскольку это означает, что мы достигли состояния, относящегося к концу введенного текста. Если суффикс не NONWORD, то мы его печатаем, а далее с помощью вызова memmove удаляем первое слово из префикса и делаем суффикс вторым словом нового префикса, после чего цикл повторяется.

Теперь все написанное можно свести воедино в функцию main, которая читает стандартный ввод и генерирует не более заданного количества слов:

```
    /* markov main: генерация
случайного текста */ /*
по алгоритму цепей Маркова */
int main(void) {
int i, nwords = MAXGEN;
char *prefix[NPREF]; /*
текущий вводимый префикс */
 for (i = 0; i < NPREF; i++)
/* начальный префикс */
 prefixfi] = NONWORD;
 build(prefix, stdin);
add(prefix, NONWORD);
generate(nwords); return 0; }
```

На этом разработка программы на C завершена. В конце главы мы сравним программы, написанные на разных языках. Главные достоинства C состоят в том, что он предоставляет программисту возможность полного управления реализацией и что программы, написанные на C, работают, как правило, весьма быстро. Расплачиваться за это приходится тем, что программист вынужден выполнять большую работ}»;: он сам должен выделять и освобождать память, создавать хэш-таблицы\1 связные списки и т. п. C — инструмент с остротой бритвы: с его помощью можно создать и элегантную программу, и кровавое месиво.

Упражнение 3-1

Алгоритм случайного выбора элемента из списка неизвестной длины зависит от качества генератора случайных чисел. Спроектируйте и осуществите эксперименты для проверки метода на практике.

Упражнение 3-2

Если вводимые слова хранятся в отдельной хэш-таблице, то каждое слово окажется записанным лишь единожды, следовательно — экономится место. Оцените, сколько именно — измерьте какие-нибудь фрагменты текста. Подобная организация позволит нам сравнивать указатели, а не строки в хэш-цепочках префиксов, что выполняется быстрее. Реализуйте этот вариант и оцените изменения в быстродействии и размере используемой памяти.

Упражнение 3-3

Удалите выражения, которые помещают сигнальные метки NONWORD в начало и конец данных, и измените generate так, чтобы она нормально запускалась и останавливалась без их использования. Убедитесь, что вывод корректен для О, 1, 2, 3 и 4 слов. Сравните код с использованием сигнальных меток и код без них.

Java

Вторую реализацию алгоритма markov мы создадим на языке Java. Объектно-ориентированные языки вроде Java заставляют нас обращать особое внимание на взаимодействие между компонентами программы. Эти компоненты инкапсулируются в независимые элементы данных, называемые объектами или классами; с ними ассоциированы функции, называемые методами.

Java имеет более богатую библиотеку, чем C. В частности, эта библиотека включает в себя набор классов-контейнеров (container classes) для группировки существующих объектов различными способами. В качестве примера можно привести класс Vector, который представляет собой динамически растущий массив, где могут храниться любые объекты типа Object. Другой пример— класс Hashtable, с помощью которого можно сохранять и получать значения одного типа, используя объекты другого типа в качестве ключей.

В нашем приложении экземпляры класса Vector со строками в качестве объектов — самый естественный способ хранения префиксов и суффиксов. Так же естественно использовать и класс Hashtable, ключами в котором будут векторы префиксов, а значениями — векторы суффиксов. Конструкции подобного рода называются отображениями (тар) префиксов на суффиксы; в Java нам не потребуется в явном виде задавать тип State, поскольку Hashtable неявным образом сопоставляет префиксы и суффиксы. Этот дизайн отличается от версии C, где мы создавали структуры State, в которых соединялись префиксы и списки суффиксов, а для получения структуры State использовали хэширование префикса.

Hashtable предоставляет в наше распоряжение метод put для хранения пар ключ-значение и метод get для получения значения по заданному ключу:

```
    Hashtable h = new HashtableQ;
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

В нашей реализации будут три класса. Первый класс, Prefix, содер,-жит слова префиксов:

```
    class Prefix {
public Vector pref;
// NPREF смежных слов из ввода
  ...
```

Второй класс, Chain, считывает ввод, строит хэш-таблицу и генерирует вывод; переменные класса выглядят так:

```
     class Chain {
static final int NPREF = 2;
 // размер префикса static
 final String NONWORD = "\hf";
// "слово", которое не может
 встретиться в тексте Hashtable
 statetab = new Hashtable();
// ключ = Prefix, значение =
suffix Vector Prefix prefix =
new Prefix(NPREF, NONWORD);
 // начальный префикс Random rand =
new Random();
```

Третий класс — общедоступный интерфейс; в нем содержится функция main и происходит начальная инициализация класса Chain:

```
    class Markov {
 static final int MAXGEN = 10000;
// максимальное количество
// генерируемых слов public static
```

```
void main(String[] args.) throws IOException
 {
Chain chain = new ChainQ;
 int nwords = MAXGEN;
 chain.build(System.in);
chain.generate(nwords); } }
```

После того как создан экземпляр класса Chain, он в свою очередь создает хэш-таблицу и устанавливает начальное значение префикса, состоящее из NPREF - констант NONWORD. Функция build использует библиотечную функцию StreamTokenizer для разбора вводимого текста на слова, разделенные пробелами. Первые три вызова перед основным циклом устанавливают значения этой функции, соответствующие нашему определению термина "слово":

```
    // Chain build: создает
    таблицу
состояний из потока ввода
void build(InputStream in)
throws IOException
 i
 \
StreamTokenizer st = new
 StreamTokenizer(in);
st.resetSyntax(); // удаляются правила
 по умолчанию st.wordChars(0,
Character.MAX_VALUE); // включаются все
st.whitespaceChars(0, ' '); //литеры, кроме
пробелов while (st.nextToken() != st.TT_EOF)
add(st.sval); add(NONWORD);
}
```

Функция add получает из хэш-таблицы вектор суффиксов для текущего префикса; если их не существует (вектор есть null), add создает новый вектор и новый префикс для сохранения их в таблице. В любом случае эта функция добавляет новое слово в вектор суффиксов и обновляет префикс, удаляя из него первое слово и добавляя в конец новое.

```
    // Chain add: добавляет
слово в список суффиксов,
обновляет префикс
void add(String word)
{

     Vector suf = (Vector)
statetab.get(prefix);
 if (suf == null) {
suf = new Vector();
statetab.put(new Prefix(prefix),
suf);

    }
suf.addElement(word);
 prefix.pref.removeElementAt(O);
```

```
prefix, pref .addElement( word);
}
```

Обратите внимание на то, что если suf равен null, то add добавляет в хэш-таблицу префикс как новый объект класса Pref ix, а не собственно pref ix. Это сделано потому, что класс Hashtable хранит объекты по ссылкам, и если мы не сделаем копию, то можем перезаписать данные в таблице. Собственно говоря, с этой проблемой мы уже встречались при написании программы на С.

Функция генерации похожа на аналогичную из программы на С, однако она получается несколько компактнее, поскольку может случайным образом выбирать индекс элемента вектора вместо того, чтобы в цикле обходить весь список.

```
    // Chain generate: генерирует
 выходной текст
 void generate(int nwords)
{
prefix = new Prefix(NPREF,
NONWORD); for
(int i = 0; i < nwords; i++)
{Vector s = (Vector)
statetab.get(prefix); int r =Math.abs
(rand.nextlnt()) % s.size();
String suf = (String) s.elementAt(r);
 if (suf.equals(NONWORD))break;
System.out.println
(suf);

    prefix.pref.removeElementAt(O);
prefix.pref.addElement(suf);
 }
 }
```

Два конструктора Prefix создают новые экземпляры класса в зависимости от передаваемых параметров. В первом случае копируется существующее значение типа Prefix, а во втором префикс создается из п копий строки; этот конструктор используется для создания NPREF копий NONWORD при инициализации префикса:

```
    // конструктор Prefix: создает
копию существующего префикса
Prefix(Prefix p)
{
pref = (Vector) p.pref.clone();
}

    // конструктор Prefix: n копий строки
 str Prefix(int n, String str)
 {
 pref = new Vector();
 for (int i = 0; i < n; i++)
 pref.addElement(str); }
```

Класс P refix имеет также два метода, hashCode и equals, которые неявно вызываются из Hashtable для индексации и поиска по таблице. Нам пришлось сделать Prefix полноценным классом как раз из-за этих двух методов, которых требует Hashtable, иначе для него можно было бы использовать Vector, как мы сделали с суффиксом.

Метод hashCode создает отдельно взятое хэш-значение, комбинируя набор значений hashCode для элементов вектора:

```
    static final int MULTIPLIER =31;
// для hashCode()
 // Prefix hashCode:
генерирует хэш-значение
// на основе всех слов префикса
public int hashCode()
 {
int h = 0;
 for
(int i = 0; i < pref.sizeO; i++)


h = MULTIPLIER *h + pref,elementAt(i).hashCode();
return h;
 }
```

Метод equals осуществляет поэлементное сравнение слов в двух префиксах:

```
    // Prefix equals:
сравнивает два префикса на
идентичность слов
public boolean equals(0bject o)

    {
Prefix p = (Prefix) o;
 for (int 1 = 0; i < pref.size(); i++)
 if
(!pref.elementAt(i).equals
(p.pref.elementAt(i)))
return false; return true;
 }
```

Программа на Java гораздо меньше, чем ее аналог на C, при этом больше деталей проработано в самом языке — очевидными примерами являются классы Vector и Hashtable. В общем и целом управление хранением данных получилось более простым, поскольку вектора растут, когда' нужно, а сборщик мусора (garbage collector — специальный автоматический механизм виртуальной машины Java) сам заботится об освобождении неиспользуемой памяти. Однако для того, чтобы использовать класс Hashtable, нам пришлось-таки самим писать функции hashCode и equals, так что нельзя сказать, что язык Java заботился бы обо всех деталях.

Сравнивая способы, которыми программы на C и Java представляют < и обрабатывают одни и те же структуры данных, следует отметить, что в версии на Java лучше разделены функциональные обязанности. При таком подходе нам, например, не составит большого труда перейти от использования класса Vector к

использованию массивов. В версии С каждый блок связан с другими блоками: хэш-таблица работает с массивами, которые обрабатываются в различных местах; функция lookup четко ориентирована на конкретное представление структур State и Suffix; размер массива префиксов вообще употребляется практически всюду.

Пропустив эту программу с исходным (химическим) текстом и форматируя сгенерированный текст с помощью процедуры f mt, мы получили следующее:

```
    % Java Markov <j rcheinistry. txt | fmt Wash the
blackboard.
Watch  it dry. The water goes into the air. When
water goes
 into the air it evaporates.  Tie a damp clotTf to
one end of a solid or liquid. Look around.
 What are   the solid things?
Chemical changes take place when something burns.
If the burning material has .liquids,
they are stable
and the sponge rise.
   It looked like dough, but it is burning. Break up the
    lump of sugar into
    small pieces and put them together again in the
bottom of a liquid.
```

Упражнение 3-4

Перепишите Java-версию markov так, чтобы использовать массив вместо класса Vector для префикса в классе Prefix.

C++

Третий вариант программы мы напишем на C++. Поскольку C+ + является почти что расширением С, на нем можно писать как на С (с некоторыми новыми удобствами в обозначениях), а наша изначальная С-версия будет вполне нормальной программой и для C++. Однако при использовании C++ было бы более естественно определить классы для объектов программы — что-то вроде того, что мы сделали на Java — это позволит скрыть некоторые детали реализации. Мы решили пойти даже дальше и использовать библиотеку стандартных шаблонов STL (Standard Template Library), поскольку в ней есть некоторые встроенные механизмы, которые могут выполнить значительную часть необходимой работы. ISO-стандарт C++ включает в себя STL как часть описания языка.

STL предоставляет такие контейнеры, как векторы, списки и множества, а также ряд основных алгоритмов для поиска, сортировки, добавления и удаления элементов данных. Благодаря использованию шаблонов C++ каждый алгоритм STL работает со всевозможными видами контейнеров, включая как типы, описанные пользователем, так и встроенные типы данных. Контейнеры реализованы как шаблоны C++, которые инстанцируются для конкретных типов данных; например, контейнер vector может использоваться для создания конкретных типов vector<int> или vector<string>. Все операции, описанные в библиотеке для vector, включая стандартные алгоритмы сортировки, можно использовать для таких "производных" типов данных.

В дополнение к контейнеру vector, который схож с Vector в Java, STL предоставляет контейнер deque (дек, гибрид очереди и стека). Дек — это двусторонняя очередь,

которая как раз подходит нам для работы с пре-_ фиксами: в ней содержится NPREF элементов, и мы можем выкидывать первый элемент и добавлять в конец новый, обе операции — за время O(1). Дек STL — более общая структура, чем требуется нам, поскольку она позволяет выкидывать и добавлять элементы с обоих концов, но характеристики производительности указывают на то, что нам следует использовать именно ее.

В STL существует также в явном виде и основанный на сбалансированных деревьях контейнер tap, который хранит пары ключ-значение и осуществляет поиск значения, ассоциированного с любым ключом, за 0(log n). Отображения, возможно, не столь эффективны, как O(1) кэш-таблицы, но приятно то, что для их использования не надо писать вообще никакого кода. (Некоторые библиотеки, не входящие в стандарт C++, содержат контейнеры hash или hash_map — они бы подошли просто идеально.)

Кроме всего прочего, мы будем использовать и встроенные функции сравнения, в данном случае они будут сравнивать строки, используя отдельные строки префикса (в которых мы храним отдельные слова!).

Имея в своем распоряжении все перечисленные компоненты, мы пишем код совсем гладко. Вот как выглядят объявления:

```
    typedef deque<string> Prefix;
map<Prefix, vector<string> > statetab;// prefix-> suffixes
```

Как мы уже говорили, STL предоставляет шаблон дека; запись deque<string> обозначает дек, элементами которого являются строки. Поскольку этот тип встретится в программе несколько раз, мы использовали typedef для присвоения ему осмысленного имени Prefix. А вот тип tap, хранящий префиксы и суффиксы, появится лишь единожды, так что мы не стали давать ему уникального имени; объявление tap описывает переменную statetab, которая является отображением префиксов на векторы строк. Это более удобно, чем в C или Java, поскольку нам не потребуется писать хэш-функцию или метод equals.

В основном блоке инициализируется префикс, считывается вводимый текст (из стандартного потока ввода, который в библиотеке C++ lost ream называется cin), добавляется метка конца ввода и генерируется выходной текст — совсем как в предыдущих версиях:

```
    // markov main: генерация случайного текста
// по алгоритму цепей Маркова
 int main(void)
 {
 int nwords = MAXGEN;
Prefix prefix; //
текущий вводимый префикс
 for (int i = 0; i < NPREF; i++)
// начальные префиксы
 add(prefix, NONWORD);
build(prefix, ciri>r"~
 add(prefix, NONWORD);
generate(nwords); return 0;
 }
```

Функция build использует библиотеку lost ream для ввода слов по одному:

```
    // build: читает слова из
входного потока,
// создает таблицу
состояний void build
(Prefix& prefix, istream& in)
{
 string buf;
 while (in » buf)
 add(prefix, buf); }
```

Строка buf будет расти по мере надобности, чтобы в ней помещались вводимые слова произвольной длины.

В функции add более явно видны преимущества использования STL:

```
    // add: добавляет слово в список суффиксов,
 обновляет п void add
(Prefix& prefix, const strings s)
{
if (prefix.size() == NPREF) {
statetab[prefix].push_back(s)
 ; prefix.pop_front();
  }
prefix.push_back(s);
 }
```

Как вы видите, выражения выглядят совсем не сложно; происходящее "за кулисами" тоже вполне доступно пониманию простого смертного. Контейнер тар переопределяет доступ по индексу (операцию[ ]) для того, чтобы он работал как операция поиска. Выражение statetab[prefix] осуществляет поиск в statetab по ключу prefix и возвращает ссылку на искомое значение; если вектора еще не существует, то создается новый. Функция push_back — такая функция-член класса имеется и в vector, и в deque — добавляет новую строку в конец вектора или дека; pop^f ront удаляет ("выталкивает") первый элемент из дека.

Генерация результирующего текста осуществляется практически так же, как и в предыдущих версиях:

```
    // generate: генерирует вывод -
по слову на строку
void generate(int nwords)
 {
Prefix prefix;
int i;
for (i = 0; i < NPREF; i++)
 // начальные префиксы add
(prefix, NONWORD);
for (i = 0; i < nwords; i++) {
vector<string>& suf =
 statetab[prefix]; const string& w =
suf[rand() % suf.size()]; if (W == NONWORD)
  break;
cout « w « "\n";
prefix.pop_front(); // обновляется
```

```
префикс prefix.push_back(w);

    }
}
```

Итак, в результате именно эта версия выглядит наиболее понятно и элегантно — код компактен, структура данных ясно видна, а алгоритм абсолютно прозрачен. Но, как и за все хорошее в жизни, за это надо платить — данная версия работает гораздо медленнее, чем версия, написанная на C; правда, это все же не самый медленный вариант. Вопросы измерения производительности мы вскоре обсудим.

Упражнение 3-5

Одно из главных преимуществ использования STL состоит в той простоте, благодаря которой можно экспериментировать с различными структурами данных. Попробуйте изменить эту версию, используя различные структуры данных для префиксов, списка суффиксов и таблицы состояний. Посмотрите, как при этом изменяется производительность.

Упражнение 3-6

Перепишите программу на C++ так, чтобы в ней использовались только классы и тип данных string — без каких-либо дополнительных библиотечных средств. Сравните то, что у вас получится, по стилю кода и скорости работы с нашей STL-версией.

AWK и PERL

Чтобы завершить наши упражнения, мы написали программу еще и на двух популярных языках скриптов — Awk и Perl. В них есть возможности, необходимые для нашего приложения, — ассоциативные массивы и методы обработки строк.

Ассоциативный массив (associative array) — это подходящий контейнер для кэш-таблицы; он выглядит как простой массив, но его индексами являются произвольные строки, или числа, или наборы таких элементов, разделенных запятыми. Это разновидность отображения данных одного типа в данные другого типа. В Awk все массивы являются ассоциативными; в Perl есть как массивы, индексируемые стандартным образом, целочисленными индексами, так и ассоциативные массивы, которые называются "хэшами" (hashes), — из их названия сразу становится ясен способ их внутренней реализации.

Предлагаемые версии на Awk и Perl рассчитаны только на работу с префиксами длиной в два слова.

```
    # markov.awk: алгоритм цепей
Маркова для префиксов из 2 слов
BEGIN { MAXGEN = 10000;
NONWORD = "\n"> w1 = w2 = NONWORD } .
{ for (i = 1; i <= NF; i++)
{ # читать все словаstatetab
[w1,w2,++nsuffix[w1,w2]] = $i
 wl = w2
w2 = $1 } } END {
Statetab[w1,w2,++nsuffix[w1,w2]]
 = NONWORD
```

```
 ft добавить метку конца ввода
w1 = w2 = NONWORD
for (i = 0; i < MAXGEN; i++)
 {  # генерируем
r = int(rand()*nsuffix[w1,w2])
 + 1 # nsuffix >= 1 p = statetab[w1,w2, r]
 if (p == NONWORD)
exit print p
w1 = w2  # идти дальше по цепочке w2 =.p
}
)
```

Awk — язык, функционирующий по принципу "образец-действие": входной поток читается по строке за раз, каждая строка сравнивается с образцом, для каждого совпадения выполняется соответствующее дей-Ствие. Существуют два специальных образца-— BEGIN и END, которые вызываются, соответственно, перед первой строкой ввода и после последней.

Действие — это блок выражений, заключенный в фигурные скобки. В Awk-версии в блоке BEGIN инициализируется префикс и пара других переменных.

Следующий блок не имеет образца, поэтому он по умолчанию вызывается для каждой новой строки ввода. Awk автоматически разделяет каждую вводимую строку на поля (ограниченные пробелами слова), имеющие имена от $1 до $NF; переменная NF — это количество полей. Выражение

```
    Statetab[w1,w2,++nsuffix[w1,w2]] = $i
```

создает отображение префиксов в суффиксы. Массив nsuffix считает суффиксы, а элемент nsuf f ix[w1, w2] считает количество суффиксов, ассоциированных с префиксом. Сами суффиксы хранятся в элементах массива statetab[w1,w2, 1], statetab[w1, w2, 2] и т. д.

Блок END вызывается на выполнение после тогсц как весь ввод был считан. К этому моменту для каждого префикса существует элемент nsuffix, содержащий количество суффиксов, и, соответственно, существует именно столько элементов statetab, содержащих сами суффиксы.

Версия Perl выглядит похожим образом, но в ней для хранения суффиксов используется безымянный массив, а не третий индекс; для обновления же префикса используется множественное присваивание. В Perl для обозначения типов переменных применяются специальные символы: $ обозначает скаляр, @ — индексированный массив, квадратные скобки [ ] используются для индексации массивов, а фигурные скобки { } — для индексации хэшей.

```
    # markov.pl: алгоритм цепей
 Маркова для префиксов из 2 слов
SMAXGEN = 1000;
$NONWORD = "\n";
 $w1 = $w2 = $NONWORD;
# начальное состояние
while (o) { n читать каждую строку ввода
 foreach (split)
{
```

```
 push((s>{$statetab{$w1}
 {$w2}
 }, $_);
 ($w1, $w2)
 = ($w2, $_);
 it множественное присваивание
}
 I-push(@{$statetab{$w1}
{$w2}



    }, $NONWORD);
 # добавить метку конца ввода
$w1 = $w2 = $NONWORD;
 for ($i =0; $i < $MAXGEN;
 $i++)
{
 $suf = $statetab{$w1}
 {$w2};
# ссылка на массив $r = int(rand @$suf);
 n @$suf - количество элементов
exit if (($t = $suf"->[$r])
eq $NONWORD);
print "$t\n";
($w1, $w2) =
($w2, $t);
 # обновить префикс
 }
```

Как и в предыдущей программе, отображение хранится при помощи переменной statetab. Центральным моментом программы является выражение

```
push(@{$statetab{$w1}{$w2}}, $_);
```

которое дописывает новый суффикс в конец (безымянного) массива, хранящегося в statetab{$w1}{$w2). На этапе генерации $statetab{$w1}{$w2} является ссылкой на массив суффиксов, а$suf->[$r] указывает на суффикс, хранящийся под номером r.

Программы и на Awk, и на Perl гораздо короче, чем любая из трех предыдущих версий, но их тяжелее адаптировать для обработки префиксов, состоящих из произвольного количества слов. Ядро программ на C и C++ (функции add и generate) имеет сравнимую длину, но выглядит более понятно. Тем не менее языки скриптов нередко являются хорошим выбором для экспериментального программирования, для создания прототипов и даже для производственного использования в случаях, когда время счета не играет большой роли.

Упражнение 3-7

Попробуйте преобразовать приложения на Awk и Perl так, чтобы они могли обрабатывать префиксы произвольной длины. Попробуйте определить, как это скажется на быстродействии программ.

Производительность

Теперь можно сравнить несколько вариантов программы. Мы засекали время счета на библейской Книге Псалмов (версия перевода King James Bible), в которой содержится 42 685 слов (5238 уникальных слов, 22 482 префикса). В тексте довольно много повторяющихся фраз ("Blessed is the..."), так что есть списки суффиксов, имеющие более 400 элементов, и несколько сотен цепей с десятками суффиксов, и это хороший набор данных для теста.

```
    Blessed is the man of the net. Turn thee unto me,
and raise me up, that
    I may tell all my fears. They looked unto him, he heard.
My praise shall
     be blessed. Wealth and riches shall be saved.
Thou hast dealt well with
    thy hid treasure: they are cast into a standing water,
the flint into
    a standing water, and dry ground into watersprings.
```

Приведенная таблица показывает время в секундах, затрачиваемое на j генерацию 10 000 слов; тестирование проводилось на машине 250 MHz MIPS R10000 с операционной системой Irix 6.4 и на машине Pentium II 400 MHz со 128 мегабайтами памяти под Windows NT. Время выполнения почти целиком определяется объемом вводимого текста; генерация происходит несравненно быстрее. В таблице приведен также примерный размер программы в строках исходного кода.

	250 MHz RIOOOO(c)	400 MHz Pentium II (c)	Строки исходного кода
C	0.36	0.30	150
Java	4.9	9.2	105
C++/STL/deque	2.6	11.2	70
C++/STL/list	1.7	1.5	70
Awk	2.2	2.1	20
Perl	1.8	1.0	18

Версии C и C++ компилировались с включенной оптимизацией, а программы Java запускались с включенным JIT-компилятором (Just-In-Time, "своевременная" компиляция). Время, указанное для версий C и C++ под Irix, — лучшее врейя из трех возможных компиляторов; сходные результаты были получены и для машин Sun SPARC и DEC Alpha.

Как видно из таблицы, версия на языке C лидирует с большим отрывом, следом за ней идет реализация на Perl. Не надо забывать, однако, что результаты, указанные в таблице, относятся к нашим экспериментам с определенным набором компиляторов и библиотек, поэтому в своей среде вы можете получить несколько другие характеристики.

Что-то явно не так с версией STL/deque под Windows. Эксперименты показали, что дек, представляющий префиксы, поедает практически все время выполнения, хотя в нем никогда не содержится более двух элементов; казалось бы, большая часть времени должна уходить на основную структуру — отображение. Переход с дека на

список (в STL это двухсвязный список) кардинальным образом улучшает время. Переход же с отображения (map) на (нестандартный) контейнер hash под Irix не приносит никаких изменений (на машине с Windows у нас не было реализации hash). В пользу замечательного принципа проектирования библиотеки STL говорит тот факт, что для внесения этих изменений нам было достаточно заменить слово list на слово deque, а слово hash на слово map в двух местах и скомпилировать программу заново. Мы пришли к выводу, что STL, которая является новым компонентом C++, все еще страдает некоторыми недоделками в реализации части своих элементов. Производительность при выборе различных версий STL или изменении используемых структур данных меняется непредсказуемым образом. То же самое можно отнести и к Java, где реализации также быстро меняются.

В тестировании программы, которая выдает в качестве результата что-то объемное, да к тому же выбранное случайным образом, есть доля творческого вызова. Как узнать, правильно ли программа работает? Всегд; ли она работает правильно? В главе 6, посвященной тестированию, м: дадим ряд советов на этот счет и расскажем, как мы тестировали наш; программу.

Уроки

Программа markov имеет длинную историю. Первая версия была на-1 писана Доном Митчелом, адаптирована Брюсом Эллисом и применялась для разнообразной забавной деконструктивистской деятельности на протяжении 1980-х годов. Она не имела никакого развития до тех пор, пока мы не решили использовать ее для иллюстрации университетского курса по проектированию программ. Однако и мы, вместо того чтобы сдуть пыль с оригинала, заново переписали ее на C, чтобы живее прочувствовать связанные с ней проблемы. После этого мы написали ее на ряде других языков, в каждом случае используя присущие тому или иному языку идиомы для выражения одной и той же основной идеи. После прочтения курса мы не раз еще переписывали программу, добиваясь предельной ясности и идеального вида.

Однако на протяжении всего этого времени основа проекта оставалась неизменной. Самые первые версии использовали такой же подход, что и версии, представленные здесь, разве что появилась вторая хэш-таблица для хранения отдельных слов. Если бы нам сейчас пришлось переписать все еще раз, мы бы вряд ли многое изменили. Архитектура программы коренится в принципиальной структуре данных. Структуры данных, конечно, не определяют каждую деталь, но формируют общее решение.

Переходы между некоторыми структурами данных вносят совсем немного различий, например переход от списков к расширяемым массивам. Некоторые реализации решают проблему в менее общем виде, например код на Awk или Perl может быть запросто изменен для обработки префиксов из одного или трех слов, но попытка реализации параметрического выбора размера префикса будет просто ужасна. Большое преимущество программ на объектно-ориентированных языках вроде C++ или Java состоит в том, что, внеся в них совсем незначительные изменения, можно создать структуры данных, подходящие для новых объектов: вместо английского текста можно использовать, например, программы (где пробелы будут уже значимыми символами), или музыкальные ноты, или даже щелчки мыши и выборы пунктов меню для генерации тестовых последовательностей.

Конечно, несмотря на то что структуры данных различаются слабо, программы на разных языках достаточно сильно отличаются друг от друга по размеру исходного кода и быстродействию. Грубо говоря, программы, написанные на языках высокого уровня, будут более медленными, чем программы на языках низкого уровня, хотя

оценку можно дать только качественную, но не количественную. Использование больших строительных блоков, таких как STL в C++ или ассоциативные массивы и обработка строк в скриптовых языках, приводит к более компактному коду и серьезно сокращает время на создание приложения. Как мы уже отмечали, за удобство приходится платить, хотя проблемы с быстродействием не имеют большого значения для программ типа рассмотренной, поскольку выполняется она всего несколько секунд.

Менее понятно, однако, как оценить ослабление контроля над программой и понимания происходящего со стороны программиста, когда размер "заимствованного" кода становится настолько большим, что отследить все нюансы становится невозможно. STL-версия — это как раз тот самый случай: ее производительность непредсказуема, и нет простых способов как-то с этим разобраться. Немногие из нас обладают достаточным запасом сил и энергии, чтобы разыскать источник трудностей и исправить ошибки.

Это всеобщая и все растущая проблема программного обеспечения — по мере того как библиотеки, интерфейсы и инструменты все более усложняются, они становятся все менее понятными и менее управляемыми. Когда все работает, среда программирования, имеющая много вспомогательных средств, может быть весьма продуктивна, но когда эти средства отказывают, трудно найти спасение. Действительно, если проблемы связаны с быстродействием или с трудноуловимыми логическими ошибками, то мы можем даже не понять, что что-то идет не так, как надо.

Проектирование и реализация этой программы дали нам несколько уроков, которые стоит усвоить перед тем, как браться за большие программы. Первый — это важность выбора простых алгоритмов и структур данных: самых простых из всех, что смогут выполнить задачу, уложившись во все ограничения (по времени, размеру и т. п.). Если кто-то уже описал эти структуры или алгоритмы и создал библиотеку, то нам это только на руку; наша C++ версия выигрывала как раз за счет этого.

Следуя совету Брукса, мы считаем, что лучше всего начинать детальное проектирование со структур данных, руководствуясь знаниями о том, какие алгоритмы могут быть использованы; после того как структуры данных определены, код проектируется и пишется гораздо проще.

Трудно сначала спроектировать программу целиком, а потом уже писать; создание реальных программ всегда включает в себя повторения и эксперименты. Процесс непосредственного написания программы вынуждает программиста уточнять решения, которые до этого существовали лишь в общем виде. Для нашей программы это весьма актуально — она претерпела множество изменений в деталях. Изо всех сил старайтесь начинать с чего-нибудь простого, внося улучшения, диктуемые опытом. Если бы нашей целью было просто реализовать алгоритм цепей Маркова для развлечения — тексты бывают довольно забавными, — мы практически наверняка написали бы его на Awk или Perl, причем даже не позаботившись навести глянец, который мы вам продемонстрировали, — и на этом успокоились бы.

Однако код для настоящего промышленного приложения требует го-'j раздо больших затрат, чем код прототипа. Если к представленным здесь программам относиться как к промышленной реализации (поскольку они были отлажены и тщательно оттестированы), то усилия на создание промышленной версии будут на один-два порядка больше, чем при написании программы для собственного использования.

Упражнение 3-8

Мы видели множество версий этой программы, написанных на различных языках программирования, включая Scheme, Tel, Prolog, Python, Generic Java, ML и Haskell; у каждой есть свои преимущества и свои недостатки. Реализуйте программу на своем любимом языке и оцените ее быстродействие и размеры кода.

Дополнительная литература

Библиотека стандартных шаблонов описана во множестве книг, включая "Генерацию программ и STL" Мэтью Остерна (Matthew Austern. Generic Programming and the STL. Addison-Wesley, 1998).

Для изучения самого языка C++ лучшим пособием является "Язык программирования C++" Бьёрна Страуструпа (Bjarne Stroustrup. The C++ Programming Language. 3rd ed. Addison-Wesley, 1997)2, для изучения Java — "Язык программирования Java" Кена Арнольда и Джеймса Гос-линга (Ken Arnold and James Gosling. The Java Programming Language. 2mt ed. Addison-Wesley, 1998). Лучшее, на наш взгляд, описание Perl при- • ведено в "Программировании на Perl" Ларри Уолла, Тома Кристиансена и Рэндала Шварца (Larry Wall, Tom Christiansen and Randal Schwartz. Programming Perl. 2nd ed. O'Reilly, 1996).

В принципе, можно говорить о существовании неких шаблонов проектирования (design patterns) — действительно, в большинстве программ используется лишь ограниченное количество принципиальных проектировочных решений — точно так же, как существует лишь несколько базовых структур данных; с некоторой натяжкой можно сказать, что это аналог идиоматического кода, о котором мы говорили в первой главе. Эта идея лежит в основе книги Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссайдеса "Разработка шаблонов: элементы повторноиспользуемого объектно-ориентированного программного обеспечения" (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995).

Авантюрные приключения программы markov (которая изначально называлась shaney) были описаны в разделе "Computing Recreations" журнала Scientific American за ию^ь 1989 года. Статья была перепечатана в книге Дьюдни "Магическая машина" (A. K. Dewdney. The Magic Machine. W. H. Freeman, 1990).

Интерфейсы

- Значения, разделенные запятой
- Прототип библиотеки
- Библиотека для распространения
- Реализация на C++
- Принципы интерфейса
- Управление ресурсами
- Abort, Retry, Fail?
- Пользовательские интерфейсы
- Дополнительная литература

И до того, как буду строить стену, я должен знать,
Что огораживаю я: внутри или снаружи, -
И что не нарушаю чьих-то прав.
Есть нечто, что не терпит ограждений,
Ломая их

Роберт Фрост. Починка стены

Суть проектирования — сбалансировать конфликтующие цели и ограничения. Когда вы пишете небольшую программу для собственного пользования, вы, конечна, можете сами выбирать конкретные решения, этот выбор не затронет ничего и никого, кроме вас. Но если ваш код будет использоваться кем-то еще, каждое решение имеет более широкие последствия.

Среди проблем, которые надо решить при проектировании, стоит выделить следующие.

- Интерфейсы: какой доступ и какой сервис предлагается? Интерфейс, в сущности, является соглашением между поставщиком (программистом) и потребителем. В идеале мы должны предоставлять унифицированные и удобные средства, имеющие достаточно возможностей для того, чтобы их было легко использовать, и в то же время не настолько большие, чтобы стать громоздкими.
- Сокрытие информации: какая информация доступна, а какая — нет? Интерфейс должен предоставлять прямой доступ к компонентам, скрывая при этом детали реализации, — с тем чтобы их можно было изменять, не затрагивая пользователя.
- Управление ресурсами: кто отвечает за управление памятью и другими ограниченными ресурсами? Здесь главными проблемами являются выделение и освобождение памяти и управление совместно используемой информацией.
- Обработка ошибок: кто обнаруживает ошибки, кто сообщает о них| и каким образом все это делается? Какие попытки восстановления предпринимаются при обнаружении ошибки?

В главе 2 мы рассмотрели составные части, из которых строится система, — структуры данных. В главе 3 мы узнали, как объединять их в небольшие программы. Теперь наше внимание сосредоточится на интерфейсах между компонентами, получаемыми из разных источников. В этой главе мы проиллюстрируем проектирование интерфейсов созданием библиотеки функций и структур данных для решения одной хорошо известной задачи. Попутно мы познакомим вас с некоторыми

принципами проектирования. Как правило, при проектировании приходится принимать огромное количество решений, но большинство из них делается почти бессознательно. Из-за незнания базовых принципов и возникают те малопонятные интерфейсы, которые ежедневно так досаждают программистам.

Значения, разделенные запятой

Значения, разделенные запятой (Comma-Separated Values — CSV), — так называется естественный и широко распространенный способ представления табличных данных. Каждая строка таблицы соответствует строке текста; поля в строке разделены запятыми. Таблица из главы 3, представленная в формате CSV, начиналась бы так:

```
    ,"250MHz","400MHz","Строки"
  , "R10000","Pentium II","исходного кода"
 C,0.36 sec,0.30 sec,150
 Java,4.9,9.2,105
```

Этот формат используется для чтения и записи различными программами, работающими с электронными таблицами. Используется он и на некоторых Web-страницах, например для выдачи справок о биржевых котировках. Одна из популярных страниц с биржевыми курсами представляет информацию примерно так:

Биржевой символ	Последние торги		Изменения		Объем
LU	2:19PM	86-1/4	+4-1/16	+4,94 %	5 804 800
T	2:19PM	60-11/16	-1-3/16	-1,92%	2468000
MSFT	2:24PM	106-9/16	+1-3/8	+ 1,31 %	11474900

Загружаемый табличный формат

Получать значения с помощью Web-браузера удобно, но долго. Вы запускаете браузер, ждете, потом на вас вываливается поток рекламы, вы выбираете список котировок, опять ждете, ждете, ждете, опять лицезрее-те рекламу и т. д. — и все для того, чтобы получить несколько чисел. Для дальнейшей обработки значений вам придется повторить весь процесс еще не один раз, а между тем, выбрав ссылку "Download Spreadsheet Format" (скачать в табличном формате), вы сразу получите файл, содержащий в основном ту же самую информацию в виде данных в формате CSV — примерно такого вида (здесь строки откорректированы нами по длине):

```
    "LU", 86. 25, "11/4/1998,",
"2:19PM",+4. 0625, 83.9375,86.875,83.625,
 5804800
"T",60.6875,"11/4/1998",
"2:19PM",-1.1875,
 62.375,62.625,60.4375,
2468000
"MSFT",106.5625,"11/4/1998","2:24PM",
 +1. 375, 105:8125,107.3125,
105.5625,11474900
```

Сразу ясно, что второй способ проще: за вас работает компьютер. Браузеры позволяют вашему компьютеру получать доступ к данным с удаленного сервера, но гораздо лучше получать данные без необходимости муторного личного участия. Надо отметить, что на самом деле все нажимания на кнопки — не более чем некая текстовая процедура: браузер читает некий HTML, вы вводите некий текст, и браузер отсылает его на сервер, получая в ответ какой-то новый HTML. Имея нормальные инструменты и язык программирования, нетрудно добиться получения информации в автоматическом режиме. Вот текст программы на языке Tel, обращающейся к Web-сайту биржевых курсов и возвращающей данные в формате CSV, предваренные несколькими заголовочными строками:

```
    8 getquotes.tcl: биржевые курсы
     для
Lucent, AT&T, Microsoft


set so [socket quote.yahoo.com 80] ;
 # соединение с сервером set q
Yd/quotes.csv?s=LU+T+MSFT&f=sl1d1t1c1ohgv"
puts $so "GET $q HTTP/1.0\r\n\r\n" ;
 # послать запрос
 flush $so
   puts [read $so]
```

Таинственная последовательность f =. . ., следующая за аббревиатурами биржевых сводок, — недокументированная управляющая строка (аналог первого аргумента printf), определяющая, какие данные требуется получить. Экспериментальным путем мы выяснили, что s задает код акций, 11 — последнюю цену, d — изменение цены по сравнению со вче-' рашним днем и т. п. Важны здесь не конкретные детали, которые могут всячески меняться, а открывающаяся возможность автоматизации получения нужной информации и преобразования ее в нужный вид без участия пользователя. Пусть работает железный агрегат.

Для того чтобы запустить getquotes, вам потребуются какие-то доли секунды, — это несравненно быстрее, чем возиться с браузером.

Получив данные, мы, естественно, захотим подвергнуть их дальнейшей обработке. С форматами данных вроде CSV лучше всего работать, если есть удобные библиотеки, осуществляющие преобразования из формата в формат и, возможно, соединенные с вспомогательными операциями типа преобразования чисел. Однако мы не знаем ни одной доступной бесплатной библиотеки для обработки CSV, поэтому напишем свою.

В нескольких последующих разделах мы создадим три версии библиотеки для чтения и преобразования данных CSV. Попутно мы обсу-! дим аспекты, неизбежно возникающие при проектировании программ, взаимодействующих с другими программами. Так, например, оказалось, что стандартного определения CSV не существует, поэтому наша реализация не может базироваться на точной спецификации, — это обычная ситуация при проектировании интерфейсов.

Прототип библиотеки

Вряд ли нам удастся получить удачный проект библиотеки интерфейса с первой попытки. Как написал однажды Фред Брукс (Fred Brooks), "заранее планируйте выкинуть первую версию — все равно придется". Брукс писал о больших системах,

но суть остается той же и для любой нормальной программы. Как правило, до тех пор пока вы не создали первой версии и не поработали с ней, трудно представить себе все аспекты работы программы настолько хорошо, чтобы спроектировать достойный продукт.

Исходя из этих соображений, мы начнем создавать библиотеку CSV с версии "на выброс", с прототипа. В первой версии мы проигнорируем многие проблемы, которые должны быть решены в грамотной библиотеке, однако она будет достаточно полной, чтобы ее можно было использовать, и с помощью этой версии мы поближе познакомимся с задачей.

Начнем с функции csvgetline, которая считывает одну строку данных CSV из файла в буфер, разделяет ее на поля массива, удаляет кавычки и возвращает количество полей. В течение многих лет мы уже не раз писали что-то подобное на различных языках, так что задание нам знакомо. Вот версия-прототип на C; мы пометили код вопросами, потому что это всего-навсего прототип:

```
    ? char buf[200]; /* буфер вводимой строки */
? char *field[20]; /* поля */
?
 ? /* csvgetline: читает и разбирает строку, */
? /* возвращает количество полей */
 ? /* пример ввода: "LU", 86. 25,
"11/4/1998", "2:19PM", +4. 0625 */
? int csvgetline(FILE *fin)
? {
? int nfield; «
? char *p, *q;
?
? if (fgets(buf, sizeof(buf), fin) == NULL)
 ? return -1;
 ? nfield = 0;
? for (q = buf; (p=strtok(q, ",\n\r")) != NULL;
 q = NULL)
? field[nfield++] = unquote(p);
? return nfield;
 ? }
```

Комментарий в начале функции включает в себя пример формата ввода, воспринимаемого функцией; такие комментарии очень полезны в программах, разбирающих беспорядочный ввод.

Формат CSV слишком сложен, чтобы разбирать его с помощью scanf, поэтому мы использовали функцию strtok из стандартной библиотеки C. Каждый вызов strtok(p, s) возвращает указатель на первую лексему (token) из строки p, состоящую из символов, не входящих в s; st rtok обрывает эту лексему, заменяя следующий символ исходной строки нулевым байтом. При первом вызове strtok первым аргументом является сканируемая строка; при следующих вызовах для обозначения того, что сканирование должно продолжиться с той точки, где закончился предыдущий вызов, в этом месте стоит NULL. Интерфейс получился убогим. Поскольку между вызовами strtok хранит переменную в некоем неизвестном месте, в каждый момент может исполняться только одна последовательность вызовов; несвязанные перемежающиеся вызовы будут конкурировать и мешать друг другу.

Использованная нами функция unquote удаляет открывающие и закрывающие кавычки, которые могут содержаться во вводимой строке. Она не обрабатывает, однако, вложенных кавычек, поэтому для прототипа ее еще можно использовать, но в общем случае она непригодна:

```
    ? /* unquote: удаляет открывающие и
 закрывающие кавычки */
? char *unquote(char *p)
 ? {
? if (p[0] == '')
{
 ? if (p[strlen(p)-1] == )
? p[strlen(p)-1] = '\O';
? p++;
?

    }
? return p;
   ?
 }
```

При выводе printf заключает поля в парные простые кавычки; это зрительно разделяет поля и помогает выявить ошибки некорректной обработки пробелов.

Мы можем прогнать через этот тест результаты работы getquotes. tcl:

```
    % getquotes.tcl | csvtest
поле[0]   =   'LIT
поле[1]   =   '86.375'
поле[2]   =   '11/5/1998'
поле[3]   =   'V.01PM'
поле[4]   =   '-0.125'
поле[5]   =   '86'
поле[6]   =   '86.375'
поле[7]   =   '85.0625'
поле[8]   =   '2888600'
поле[0]   =   'T'       v
поле[1]   =   '61..0625'
...
```

(Заголовки .HTTP мы убрали.)

Итак, у нас'есть прототип, который, кажется, в состоянии работать с данными вроде приведенных выше. Однако теперь было бы логично опробовать его на чем-то еще (особенно если мы планируем распространять эту библиотеку). Мы нашли еще один Web-сайт, позволяющий скачать биржевые котировки и получить файл с той же, собственно, информацией, но представленной в несколько иной форме: для разделения записей вместо символа йеревода строки используется символ возврата каретки (\r), а в конце файла завершающего возврата каретки нет. Выглядят новые данные так (мы отформатировали их, чтобы они умещались на странице):

```
    "Ticker","Price","Change","Open",
"Prev Close","Day High", "Day Low",
"52 Week High","52 Week Low","Dividend",
```

```
    "Yield","Volume","Average Volume","P/E"
"LIT,86.313,-0.188,86.000,86.500,
 86.438,85.063,108.50, 36.18,0.16,0.1,2946700,
 9675000, N/A
"T",61.125,0.938,60.375,60.188,61.125,
60.000, 68. 50, 46.50,1.32,2.1,3061000,
4777000,17.0
"MSFT",107.000,1.500,105.313,105.500,
107.188,105.250, 119.62,59.00,
 N/A,N/A,7977300,16965000,51.0
```

При таком вводе наш прототип позорно провалился.

Мы спроектировали наш прототип, изучив только один источник данных, тестирование провели на данных из того же источника. Стало быть, нечего удивляться тому, что первое же столкновение с данными из другого источника привело к гибельным последствиям.

Длинные строки на вводе, большое количество полей, непредусмотренные или пропущенные разделители — все это вызывает проблемы. Наш ненадежный прототип подходит только для индивидуального использования или в целях демонстрации принципиальной пригодности выбранного подхода, но не более того. Что ж, пришло время переработать проект.

При создании прототипа мы сделали ряд предположений — явных и неявных. Ниже перечислены некоторые из наших решений, зачастую не самых подходящих для универсальной библиотеки. Каждое поднимает вопрос, требующий более тщательной проработки.

- Прототип не способен обработать длинные строки или большое количество полей. Он может выдавать неправильные ответы или вообще зависать, потому что в нем нет даже проверки на переполнение, не говоря уже о возвращении каких-то разумных значений при возникновении ошибок.
- Предполагается, что ввод состоит из строк, оканчивающихся символом перевода строки.
- Поля разделены запятыми; если поле заключено в кавычки, последние удаляются. Не предусмотрен случай вложенных кавычек или запятых.
- Вводимая строка не сохраняется; в процессе генерации полей она переписывается.
- При переходе от одной строки к следующей никакие данные не сохраняются; если что-то надо запоминать, то следует создавать копию.
- Доступ к полям осуществляется через глобальную переменную — массив field, который используется совместно функцией csvget-line и функцией, которая ее вызвала; нет никакого контроля доступа к содержимому полей или указателям. Не предусмотрено никаких средств для предотвращения доступа за последнее поле.
- Использование глобальных переменных делает проект непригодным для многонитевой среды или даже для одновременного исполнения двух параллельных вызовов.
- Функция csvgetline читает только из уже открытых файлов; открытие лежит целиком на совести вызывающего кода.
- Ввод и разделение полей прочно связаны друг с другом: каждый вызов читает строку и сразу же разбивает ее на поля вне зависимости от того, есть ли в этом необходимость.

- Возвращаемое значение есть количество полей в строке; для подсчета этого значения строка должна быть разделена на поля. Не предусмотрено никакого механизма для определения ошибок конца файла.
- Нет никакой возможности изменить ни одно из перечисленных свойств, не внося изменений в код.

В этом длинном, но далеко не полном списке приведены те решения, которые мы приняли на этапе проектирования, — и каждое решение навсегда вплетено в код. Это приемлемо для временной версии, разбирающей файлы известного формата, поступающие из одного конкретного источника. Но что будет, если формат изменится, или запятая появится внутри кавычек, или сервер выдаст длинную строку или много полей?

Может показаться, что со всем этим нетрудно справиться, ведь "библиотека" мала и, в конце концов, является всего лишь прототипом. Представьте, однако, что этот код, пролежав в забвении месяцы или годы, в какой-то момент станет частью большой программы, спецификации которой будут/ меняться. Как адаптируется csvgetline? Если программу будут использовать другие люди, то скороспелые решения в ее проектировании могут вызвать проблемы, которые проявятся через долгое время. К сожалению, история многих интерфейсов подтверждает это заявление: большое количество временного, чернового кода просачивается в большие программные системы, в которых этот код так и остается "грязным" и зачастую слишком медленным.

Библиотека для распространения

Теперь с учетом того, чему мы научились на опыте прототипа, попробуем создать библиотеку общего назначения. Наиболее очевидные требования такие: csvgetline должна быть более устойчива, то есть уметь обрабатывать и длинные строки, и большое количество полей; более осторожно надо подойти и к синтаксическому разбору полей.

Для создания интерфейса, который могли бы использовать и другие люди, мы должны выработать решения по аспектам, перечисленным в начале главы: интерфейсы, сокрытие деталей, управление ресурсами и обработка ошибок; их взаимосвязь оказывает сильнейшее влияние на проект. Наше разделение этих проблем было несколько произвольно, так как они сильно взаимосвязаны.

Интерфейс. Мы выработали решение о трех базовых операциях:

char *csvgetline(FILE *): читает новую CSV строку;
char *csvf ield(int n): возвращает n-е поле текущей строки;
int csvnf leld(void): возвращает число полей в текущей строке.

Какое значение должна возвращать csvgetline? Желательно, чтобы она возвращала побольше полезной информации; тогда напрашиваете* возвращение того же количества полей, как и в прототипе. Но тогда количество полей будет подсчитываться даже в случае, если поля эти больше использоваться не будут. Еще один вариант возврата — длина вводимой строки, но это значение зависит от того, включать ли в длину завершающий символ перевода строки. После ряда экспериментов мы пришли к выводу, что csvgetline должна возвращать указатель на оригинальную строку ввода или NULL, если был достигнут конец файла.

Мы будем удалять символ перевода строки из конца строки, возвращаемой csvgetline, так как, если понадобится, его можно без труда вписать обратно.

С определением поля дело обстоит довольно сложно; мы попробовали собрать воедино варианты, которые встречались в электронных таблицах и других программах. Получилось примерно следующее.

Поле является последовательностью из нуля или более символов. Поля разделены запятыми. Открывающие и завершающие пробелы (пропуски) сохраняются. Поле может быть заключено в двойные кавычки, в этом случае оно может содержать запятые. Поля, заключенные в двойные кавычки, могут содержать символы двойных кавычек, представляемые парой последовательных двойных кавычек, — то есть CSV поле "x""y" определяет строку x"y. Поля могут быть пустыми; поле, определяемое как "", считается пустым и эквивалентно полю, определяемому двумя смежными запятыми.

Поля нумеруются с нуля. Как быть, если пользователь запросит несуществующее поле, вызвав csvfield(-l) или csvfield(100000)? Мы могли бы возвращать "" (пустую строку), поскольку это значение можно выводить или сравнивать; программам, которые работают с различным количеством полей, не пришлось бы принимать специальных предосторожностей на случай обращения к несуществующему полю. Однако этот способ не предоставляет возможности отличить пустое поле от несуществующего. Второй вариант — выводить сообщение об ошибке или даже прерывать работу; несколько позже мы объясним, почему так делать нежелательно. Мы решили возвращать NULL — общепринятое в C значение для несуществующей строки.

Сокрытие деталей. Библиотека не будет накладывать никаких ограничений ни на длину вводимой строки, ни на количество полей. Чтобы осуществить это, либо вызывающая сторона должна предоставить память, либо вызываемая сторона (то есть библиотека) должна ее зарезервировать. Посмотрим, как это организовано в сходных библиотеках: при вызове функции f gets ей передается массив и максимальный размер; если строка оказывается больше буфера, она разбивается на части. Для работы с CSV такое поведение абсолютно неприемлемо, поэтому наша библиотека будет сама выделять память по мере необходимости.

Только функция csvgetline занимается управлением памятью; вне ее ничего о методах организации памяти не известно. Лучше всего осуществлять такую изоляцию через интерфейс функции: получается (то есть видно снаружи), что csvgetline читает следующую строку — вне зависимости от ее размера, csvfield(n) возвращает указатель на байты n-го поля текущей строки, а csvnf ields возвращает количество полей в текущей строке.

Мы должны будем наращивать память по мере появления длинных строк или большого количества полей. Детали того, как это происходит, спрятаны в функциях csv; никакие другие части программы не знают, как это делается: использует ли библиотека маленькие массивы, наращивая их при необходимости, или, наоборот, очень большие массивы, или вообще какой-то совершенно другой подход. Точно так же интерфейс не раскрывает и того, когда же память высвобождается.

Если пользователь вызывает только csvgetline, то нет надобности разделять строку на поля; это можно сделать по специальному требованию. Происходит ли разделение полей ретиво (eager, непосредственно при чтении строки), лениво (lazy, только когда нужно посчитать количество полей) или очень ленива (very lazy,

выделяется только запрошенное поле) — еще одна деталь реализации, скрытая от пользователя.

Управление ресурсами. Мы должны решить, кто отвечает за совместно используемую информацию.

Возвращает ли csvgetline исходные данные или делает копию? Мы решили, что csvgetline возвращает указатель на исходные данные, которые будут перезаписаны при чтении следующей строки. Поля будут созданы в копии введенной строки, и csvf ield будет возвращать указатель на поле в копии строки. При таких соглашениях пользователь должен сам создавать дополнительную копию, если какая-то конкретная строка или поле должны быть сохранены или изменены, и пользователь же отвечает за высвобождение этой памяти после того, как необходимость в ней отпадет.

Кто открывает и закрывает файл ввода? Кто бы ни открывал вводимый файл, он же должен его закрыть; парные действия должны выполняться на одном уровне или в одном и том же месте. Мы будем исходить из предположения, что csvgetline вызывается с указателем FILE, определяющим уже открытый файл; по окончании обработки файл будет закрыт вызывающей стороной.

Управление ресурсами, используемыми совместно или передающимися между библиотекой и вызывающими ее программами, — сложная задача; часто существуют веские, но противоречивые доводы в пользу различных решений. Ошибки и недопонимание при разделении ответственности за управление совместно используемыми ресурсами — характерный источник ошибок.

Обработка ошибок. Когда csvgetline возвращает NULL, не существует способа отличить выход на конец файла от ошибки вроде нехватки памяти; точно так же и доступ к несуществующему полю не вызовет ошиб: ки. По аналогии с terror мы могли бы добавить в интерфейс еще одну функцию, csvgete г го г, которая сообщала бы нам о последней ошибке, но для простоты мы не будем включать ее в данную версию.

Надо принять за постулат, что функции библиотеки не должны просто прерывать исполнение при возникновении ошибки; статус ошибки должен быть возвращен вызывающей стороне. Также не следует печатать сообщения или выводить окна диалога, поскольку функции библиотеки могут исполняться в системах, где такие сообщения будут мешать чему-то еще. Обработка ошибок — тема, достойная отдельного обсуждения, и мы еще вернемся к ней далее в этой главе.

Спецификация. Все описанные выше решения и допущения должны быть собраны воедино в спецификацию, описывающую средства, предоставляемые csvgetline, и методы ее использования. В больших проектах спецификация должна предшествовать реализации, при этом, как правило, разные люди и даже разные организации создают спецификацию и пишут код. Однако на практике эти работы часто производят параллельно — тогда спецификация и код эволюционируют совместно; иногда же "спецификация" пишется уже после разработки программы, чтобы приблизительно описать, что же делает код.

Самый грамотный подход — писать спецификацию как можно раньше и (как делали мы) пересматривать ее по мере реализации кода. Чем тщательнее и вдумчивее будет написана спецификация, тем больше вероятность создать хороший продукт. Даже при создании программ для собственного пользования важно подготовить

достаточно осмысленную спецификацию, поскольку она требует анализа существующих проблем и четкого фиксирования принятых решений.

В нашем случае спецификация будет включать в себя прототипы функций и детальное описание их поведения, сделанных допущений и распределения ответственности:

Поля разделены запятыми.
Поле может быть заключено в двойные кавычки: "...".
Поле, заключенное в кавычки, может содержать
запятые,но не символы перевода строки.
Поле, заключенное в кавычки, может содержать
символыдвойных кавычек, представляемые парой
двойных кавычек Поле может быть пустым;
"" и пустая строка равно представляют пустое
поле. Предваряющие и заключительные
пробелы сохраняются.

char *csvgetline(FILE *f);
читает одну строку из файла ввода f;
подразумевается, что строки во вводе
оканчиваются символами \r, \n, \r\пилиEOE.
возвращает указатель на строку
(символы конца строки
удаляются) или NULL, если достигнут EOF.
строки могут иметь произвольную
длину; возвращается
NULL, если превышен резерв памяти, строки
рассматриваются как память,
доступная только для чтения;
для сохранения или изменения содержимого
вызывающая сторона должна сделать
копию.

char *csvfield(int n);
поля нумеруются начиная с 0. возвращает n-е поле
из последней строки, прочитанной csvgetline;
возвращает NULL, если n отрицательно или лежит
за последним полем, поля разделяются запятыми.
поля могут быть заключены в двойные кавычки,
эти кавычки убираются;внутри двойных кавычек
запятая не являетсяразделителем, а пара
символов "" заменяется на ". в полях,
не ограниченных кавычками, кавычки
рассматриваютсякак обычные символы, может
быть произвольное количество полей любой
длины;возвращает NULL, если превышается
резерв памяти, поля рассматриваются
как память, доступная только для чтения;
для сохранения или изменения содержимого
вызывающая сторона должна сделать копию,
при вызове до csvgetline поведение не определено.

```
int csvnfield(void);
возвращает количество полей в последней строке,
прочитанной csvgetli ne. при вызове до csvget line
 поведение не определено.
```

Представленная спецификация все еще оставляет некоторые вопро- < сы открытыми. Например, какие значения должны возвращать csvf ield и csvnf ield, если они вызваны после того, как csvgetline натолкнулась на EOF? Разрешить подобные головоломки непросто даже для маленькой I программы, а для больших систем — исключительно трудно, но очень важно хотя бы попробовать с ними справиться. В противном случае вы рискуете обнаружить пробелы и недочеты уже в ходе реализации проекта.

Остаток параграфа посвящен новой реализации csvgetl i ne, которая COOT- | ветствует нашей спецификации. Библиотека разбита на два файла — заголовочный csv. h и файл воплощения csv. c. В заголовочном файле содержатся объявления функций, то есть представлена общедоступная часть интерфейса. В csv. c содержится собственно рабочий код библиотеки — реализации функций. Пользователи включают csv. h в свой исходный код и компонуют свой скомпилированный код со скомпилированной версией csv. c; таким образом, исходный код библиотеки никогда не должен быть видим.

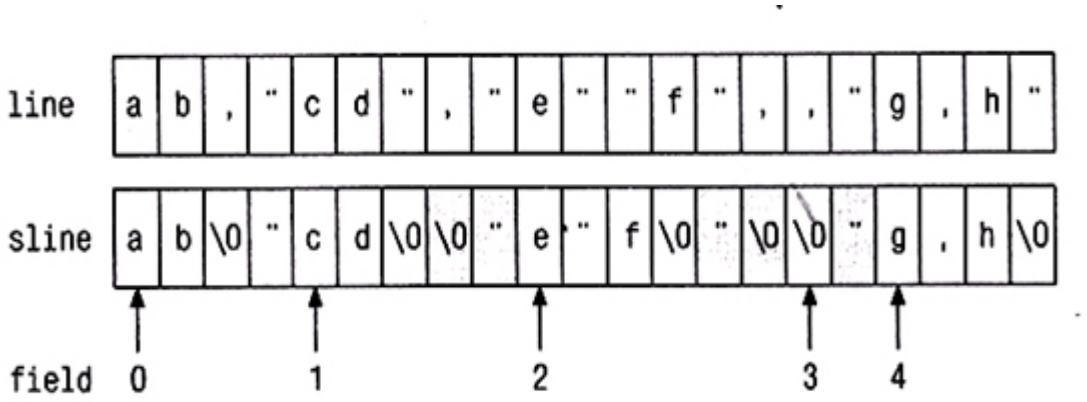
```
    /* csv.h: интерфейс библиотеки csv */
extern char *csvgetline(FILE *f);
 /* читает очередную строку */
 extern char *csvfield(int n);
/* возвращает поле номер n
 */ extern int csvnfield(void);
 /* возвращает количество полей */
```

Внутренние переменные, в которых хранится текст, и внутренние функции вроде split объявлены статическими (static), так что они видны только внутри содержащего их файла. Это простейший способ сокрытия информации в программе на C.

```
    enum { NOMEM = -2 };
 /* сигнал о нехватке памяти */
static char *line = NULL;
 /* вводимые символы */
static char *sline = NULL;
 /* копия строки для split */
static int maxline = 0;
/* размер line[] и sline[] */
 static char **field = NULL;
 /* указатели полей */
static int maxfield = 0;
/* размер fieldf] */
 static int nfield = 0:
 /* количество полей в field[] */>
    static char fieldsep[] = ",-";
 /* символы - разделители полей       */
```

Переменные инициализируются также статически. Эти начальные значения используются для проверки необходимости создания или наращивания массивов.

Эти объявления описывают простую структуру данных. Массив line содержит вводимую строку; массив sline создается путем копирования символов из line и вставки разделителя после каждого поля. Массив field указывает на значения в sline. На диаграмме показано состояние этих трех массивов после того, как была обработана строка ab, "cd", "e""f",, "g, h". Заштрихованные элементы в sline не являются частью какого-либо поля.



А вот как выглядит сама функция csvgetline:

```
    /* csvgetline: получаем одну строку, */
/* при необходимости наращиваем память.
*/ /* Пример ввода: "LU", 86.25,"11/4/1998",
"2:19PM",+4.0625 */ char *csvgetline(FILE *fin)
int i, c;
 char *newl, *news;
 if (line == NULL1)
{ /*при первом вызове выделяется память*/
maxline = maxfield = 1;
 line = (char *) malloc(maxline);
 sline = (char *) malloc(maxline);
 field = (char **) malloc(maxfield*
 sizeof(field[0]));
 if (line == NULL || sline == NULL | 'field == NULL)
{ reset (); return NULL; /* нехватка памяти */
 for (i=0; (c=getc(fin))!=EOF &
& !endofline(fin,c); i++) { if (i >= maxline-1)
{ /* увеличение строки */
 maxline *= 2;
 /* удвоение текущего размера */
 if (newl == NULL) { reset();
 return NULL; /* нехватка памяти */ }
 line = newl;
 news = (char *) realloc(sline, maxline);
 if (news == NULL) { reset();
 return NULL; /* нехватка памяти */ }
sline = news; }
 line[i] = c; }
 line[i] = ДО'; if (splitO == NOMEM)
 { reset();
return NULL;
 /* нехватка памяти */ }
```

```
 return
 (c == EOF && i == 0) ? NULL : line;
   }
```

Поступающая строка накапливается в строке line, которая при необходимости наращивается, вызывая realloc; при каждом увеличении размер удваивается, как в параграфе 2.6. Массив sline всегда увеличивается до размера line; csvgetline вызывает split для создания в отдельном массиве field указателей на поля — этот массив также при необходимости наращивается.

Мы привыкли начинать с очень маленьких массивов и увеличивать их по потребности, чтобы иметь гарантию, что код увеличения массива был выполнен. Если выделения памяти не происходит, мы вызываем reset для восстановления глобальных значений в их первоначальное состояние, чтобы дать шанс на успех последующему вызову csvgetline:

```
    /* reset: возвращает переменные в

    начальные значения */
static void reset(void)
 {
 free(line); /* free(NULL)
разрешено в ANSI C */
 free(sline);
free(field);
line = NULL;
 sline = NULL;
 field = NULL;
 maxline = maxfield = nfield = 0;
 }
```

Функция endof line нужна для выявления и обработки ситуаций, когда вводимая строка заканчивается символами возврата каретки, перевода строки, ими обоими вместе или даже EOF:

```
    /* endofline: выявляет и
 уничтожает \r, \n, \r\n, EOF */
static int endofline(FILE *fin, int c)
 {
 int eol;
 /eol = (c=='\r' I c=='\n'); if (c == Af)
 { c = getc(fin); if (c != Дп' && c != EOF)
ungetc(c, fin); /* прочитано слишком далеко; */
/* c возвращается обратно */ }
return eol;
 }
```

Здесь необходима отдельная функция, поскольку стандартные функции ввода не обрабатывают все многообразие нетривиальных форматов, встречающихся в реальных условиях.

Наш прототип использовал st rtok для определения следующего поля поиском символа-разделителя, которым в принципе является запятая. Однако при таком подходе невозможно обрабатывать запятые, находящиеся внутри кавычек. В split

необходимо внести глобальные изменения (хотя ее интерфейс и не изменится). Представьте себе такие строки ввода:

```
" " " " "
' '
" "
' '
' '
```

Каждая строка содержит по три пустых поля. Для того чтобы split была в состоянии корректно интерпретировать такие строки и им подобные, ее реализацию придется глобально усложнить, — это классический пример того, как особые случаи и граничные условия оказываются доминирующими в программе.

```
    /* split: разделяет строки на поля */
 static int spiit(void)
{
char *p, **newf;
 char *sepp; /* указатель на временный
 символ-разделитель */
 int sepc; /* временный
символ-разделитель */
nfield = 0;
 if (line[0] == ДО')
 return 0;
 strcpy(sline, line); p = sline;
do
{
 if (nfield >= maxfield)
 {
 maxfield *= 2; /* удвоить текущий размер
массива */ newf = (char **) realloc(field,
  - maxfield * sizeof(field[0]));
if (newf == NULL)
return NOMEM; field = newf;
if (*p == "')
sepp = advquoted(++p); /* пропуск
 первой кавычки */ else
sepp = p + strcspn(p, fieldsep);
sepc = sepp[0];
sepp[0] = '\0'; /* окончить поле
 */ field[nfield++] = p; p = sepp + 1; }
 while (sepc == ',.');
return nfield;
  }
```

В теле цикла массив указателей на поля при необходимости увеличивается, после этого вызывается очередная функция, осуществляющая поиск и обработку следующего поля. Если поле начинается с двойной кавычки, advquoted находит поле и возвращает указатель на разделитель, которым поле заканчивается. В противном случае для поиска следующей запятой мы используем библиотечную функцию strcspn(p, s), которая ищет в строке p следующее вхождение любого из символов строки s; возвращает эта функция количество пропущенных символов.

Двойные кавычки внутри поля представляются парой смежных двойных кавычек; функция advquoted сжимает такую комбинацию до одной кавычки, а также удаляет кавычки, обрамляющие поле. Дополнительный код добавлен в попытке справиться с правдоподобным вводом, не подходящим под спецификацию, — таким, например, как "abc"def. В подобных случаях мы добавляем в конец поля все, что заключено между второй кавычкой и следующим разделителем. Оказывается, Microsoft Excel использует схожий алгоритм.

```
    /* a'dvquoted: для поля, обрамленного кавычками;
*/ /*/ возвращает указатель на следующий
 разделитель */ s/atic char *advquoted(char *p)
  /       int i, j;
 for (i = j = 0; p[j] != ДО'; i++, j++)
  {
if (p[j] == "" && p[++j] != "")

    {
  /* копировать до следующего разделителя

    или

    \0 */ int k = strcspn(p+j, fieldsep);
memmove(p+i, p+j, k); i += k; j += k; break; }
p[i] = Ptj];

    }
 P[i] = ЛО'; return p + j';
 }
```

Поскольку входная строка уже разделена, то реализация csvfield и csvnfield становится тривиальной:

```
     /* csvfield: возвращает указатель
 на n-е поле */
char *csvfield(int n)
{
if (n < 0 | n >= nfield) return NULL;
return fieldfn];
 }
/* csvnfield: возвращает количество полей */
int csvnfield(void)
{
return nfield;
 }
```

Наконец мы можем модифицировать тестирующую программу и проверить эту версию библиотеки: поскольку копия строки ввода сохраняется (чего в прототипе не было), появилась возможность распечатывать сначала исходную строку, а потом уже полученные поля:

```
    /* csvtest main: тестирует
CSV библиотеку */
 int main(void)
{
```

```
int i;
 char *line;
 while ((line = csvgetline(stdin)) != NULL)
 { printf("line = '%s'\n", line); for
 (i = 0; i < csvnfield(); i++)
printf("field[%d]
= '%s'\n", i, csvfield(i))
 }
return 0;
 }
```

На этом версия на C завершена. Она обрабатывает строки произвольной длины и делает нечто осмысленное даже при некорректном вводе. В результате нам пришлось написать в четыре раза больше кода, чем в прототипе, при этом некоторые фрагменты получились довольно запутанными. Подобное увеличение объема и сложности кода — совершенно типично для перехода от прототипа к полноценному продукту.

Упражнение 4-1

При разделении полей возможно несколько уровней "ленивости" - -разделять сразу все поля, но только после получения запроса на конкретное поле, выделять только запрошенное поле и, наконец, разделять все поля до запрошенного. Рассмотрите потенциальные преимущества и недостатки перечисленных способов; реализуйте их и замерьте скорость работы каждого.

Упражнение 4-2

Добавьте новые возможности в библиотеку. Сделайте так, чтобы:
а) разделитель мог принадлежать к произвольному классу символов;
б) для разных полей существовали разные разделители;
в) разделитель был заменен на регулярное выражение (см. главу 9). На что будет похож получившийся интерфейс?

Упражнение 4-3

В нашей реализации библиотеки мы применили статическую инициализацию, используемую в C в качестве основы для одноразового переключения: если на входе указатель есть NULL, то выполняется инициализация. Можно, однако, сделать и по-другому: заставить пользователя вызывать некотору/ю специальную функцию инициализации — в ней, кстати, могут содержаться некоторые рекомендованные начальные значения для массивов. Попробуйте написать версию, которая объединяла бы все достоинства обоих подходов. Какую роль в вашей версии будет играть reset?

Упражнение 4-4

Спроектируйте и реализуйте библиотеку для записи данных в формате CSV. Простейшая версия может просто брать массив строк и печатать их с кавычками и запятыми. Более интересный вариант — использовать форматные строки как printf. В главе 9 вы найдете некоторые полезные советы.

Реализация на C++

В этом разделе мы напишем версию библиотеки CSV на C++, в которой постараемся преодолеть некоторые ограничения, имеющиеся в C~версии. Нам придется внести некоторые изменения в спецификацию, главным из которых станет то, что функции будут теперь обрабатывать строки C++ вместо массивов символов C. Использование строк C++ автоматически решит некоторые проблемы, связанные с хранением данных, поскольку управлением памятью вместо нас займутся библиотечные функции. Так, в частности, функции работы с полями будут возвращать строки, которые затем могут изменяться вызывающей стороной, — проект получится более гибким, чем в предыдущей версии.

Класс Csv определяет внешние спецификации, изящно скрывая при этом переменные и функции реализации. Поскольку объект класса исчерпывающе описывает все состояния экземпляра, мы можем обрабатывать сразу несколько переменных Csv, при этом каждая из них будет абсолютно независима, так что одновременно могут обрабатываться сразу несколько входных потоков CSV.

```
    class Csv { // читает и разбирает CSV
// пример ввода: "LIT, 86. 25, "11/4/1998",
"2:19PM", +4. 0625
 public:
Csv(istream& fin = cin, string sep = ",") :
fin(fin), fieldsep(sep) {}
 int getline(string&);
string getfield(int n);
 int getnfieldO const { return nfield; }
private:
 istream& fin; // указатель на файл ввода
 string line; // вводимая строка
vector<string> field; // строки полей
 int nfield; // количество полей
string fieldsep; // символы разделителей
 int split();
 int endofline(char);
 int advplain(const string& line,
string& fid, int);
 int advquoted(const string& line,
 string& fid, int); };
```

Для конструктора определены параметры, принимаемые по умолчанию, — такой объект Csv будет читать из стандартного входного потока и использовать обычный символ-разделитель; эти параметры можно изменить, задав другие значения в явном виде.

Для работы со строками класс использует не строки C, а стандартные C++-классы st ring и vector. Для типа st ring не существует невозможного состояния — "пустая" строка означает всего лишь строку нулевой длины, и нет никакого значения, эквивалентного по своему смыслу NULL, который бы мы использовали как сигнал достижения конца файла. Таким образом, Csv: :getline возвращает введенную строку через аргумент, передаваемый по ссылке, используя возвращаемое значение для сообщений о конце файла или ошибках.

```
    // getline: получает одну строку,
// по мере необходимости
наращивает размер
 int Csv::getline(string& str)
 {
 char c;
 for (line = ""; fin.get(c) && !endofline(c); )
line += c; split(); str = line; return !fin.eof();
}
```

Операция += здесь переопределяется, чтобы добавлять символ в строку.

Несколько меньше изменений потребуется вносить в endof line. Нам точно так же придется считывать ввод посимвольно, поскольку ни одна из стандартных функций ввода не может обработать все многообразие вариантов.

```
    // endofline: ищет и удаляет  \r', \n, \r\n или EOF
 int Csv::endofline(char c)
{
 int eol;
eol = (c==Ar' || c=='\n');
 if (c == V) { , fin.get(c); if
 (!fin.eof() && c != '\n',)
 fin.putback(c);
// слишком много прочитали >
 return eol;
}
```

А вот как выглядит новая версия функции split:

```
     // split: разделяет строку на поля
 int Csv::split()
 {
 string fid;
 int i, j;
nfield = 0;
  if (line.lengthO == 0)
return 0; i = 0;
 do
{
if (i < line.length()
&& line[i] == '".')
 j = advquoted(line, fid, ++i);
 // пропуск кавычки else
 j = advplain(line, fid, i);
if (nfield >= field.size())
 field.push^back(fld);
 else
 field[nfield] = fid; nfield++; i = j + 1;
 }
 while (j < line.length());
 return nfield;
 }
```

Поскольку strcspn не работает со строками C++, нам надо изменить и split, и advquoted. Новая версия advquoted для поиска следующего вхождения символа-разделителя использует стандартную C++-функ-цию find_first_of. Вызов s. find_first_of (fieldsep, j) ищет в строке s первое вхождение любого символа из fieldsep, начиная с позиции]. Если вхождение найдено не было, возвращается индекс, лежащий за концом строки, так что нам надо будет вернуть его обратно в должный диапазон. Внутренний цикл for в advquoted добавляет в поле fid все символы, расположенные до ближайшего разделителя.

```
    // advquoted: для полей,  заключенных в кавычки;
 // возвращает индекс следующего разделителя
 int Csv::advquoted(const string& s, string& fid, int i)
 {
 int j;
 fid = "";
 for (j = i; j < s.length(); j++)
 {
 if (S[j] == "&& s[++j] != ")
 {
 int k = s.find_first_of(fieldsep, j);
 if (k > s.length())
 // разделитель не найден
 k = s.length(); for (k -= j; k-- > 0; )
 fid += s[j++];
 break;
 }
```

Функция find_first_of используется также и в новой функции advplain, которая обрабатывает обычные, не заключенные в кавычки поля. Еще раз подчеркнем, что необходимость в этом обусловлена тем, что функции языка C вроде strcspn не могут быть применены к строкам C++, которые представляют собой совершенно особый тип данных.

```
    // advplain: для полей, не заключенных в кавычки,
// возвращает индекс следующего разделителя
int Csv::advplain(const strings s, strings fid, int i)

    {
  int j;
 j = s.find_first_of(fieldsep, i);
 // поиск разделителя if (j > s.lengthO)
 // разделитель не найден
 j = s.lengthO; fid = string(s, i, j-i); return j;
 }
```

И снова, как и в предыдущей версии, Csv::getfield абсолютно тривиальна, а Csv: :getnfield настолько коротка, что воплощена прямо в описании класса.

```
    // getfield: возвращает n-е поле string
 Csv::getfield(int n) {
 if (n < 0 n >= nfield)
 return ""; else
 return field[n];
}
```

Тестовая программа представляет собой несколько упрощенный йа-риант предыдущей версии:

```
    // Csvtest main: тестирует класс Csv int main(void)
{
string line; Csv csv;
 while (csv.getline(line) != 0)
 {
cout « "Строка = '"
« line <<"'\n"; for (int i = 0; i < csv.getnfieldQ;
 i++)
cout « "Поле[" « i « "] = '"
« csv.getfield(i) « "'\n";
 }
    return 0;
  }
```

Использование библиотеки в C++ незначительно отличается от версии на C. В зависимости от компилятора новая версия в сравнении с C-версией дает замедление от 40 % до четырех раз на большом файле из 30 000 строк примерно по 25 полей на строку. Как мы уже выясняли при оценке быстродействия программы markov, подобный разброс зависит от степени проработанности используемых библиотек. Последнее, что остается добавить: исходный код версии C++ получился примерно на 20 % короче.

Упражнение 4-5

Введите в версию C++ оператор [ ], чтобы к полям можно было обращаться как к csv[i].

Упражнение 4-6

Напишите библиотеку CSV на Java, а затем сравните все три версии с точки зрения простоты и ясности, надежности и скорости.

Упражнение 4-7

Перепишите C++ версию кода CSV с использованием класса STL iterator.

Упражнение 4-8

Версия на C++ предоставляет возможность нескольким независимым экземплярам Csv работать одновременно, никак не мешая друг другу, — в этом выразилось важное достоинство инкапсуляции всего состояния объекта, экземпляры которого можно порождать многократно. Измените версию на C так, чтобы добиться подобного эффекта; для этого замените глобальные структуры данных структурами, выделение памяти для которых и инициализация осуществляются явным образом с помощью отдельной функции csvnew.

Принципы интерфейса

В предыдущих параграфах мы прорабатывали детали некоего интерфейса. Сформулируем теперь в общих чертах, что же такое интерфейс. Интерфейс -- это детализированная, описанная граница взаимодействия между кодом, предоставляющим некоторые возможности, и кодом, который эти возможности использует. Интерфейс определяет, что именно предоставляет своему пользователю некоторый законченный блок кода, каким образом функции (а может быть, и какие-то элементы данных) из этого блока могут быть использованы в остальной части программы. Интерфейс CSV предоставляет пользователю три функции: чтение строки, получение поля и возврат количества полей. Кроме них, пользователь не может получить от нашего кода ничего.

Для того чтобы оказаться удобным, интерфейс должен отвечать некоторым базовым требованиям: быть простым, общим, стандартным, предсказуемым, надежным, а также нести в себе возможность без потерь адаптироваться к изменениям запросов пользователей и своей внутренней реализации. В основе хороших интерфейсов лежат несколько принципов. Принципы эти тесно взаимосвязаны, а иногда даже противоречивы, но они помогут нам описать, что же происходит при пересечении границы между двумя частями программы.

Прячьте детали реализации. Реализация, которая стоит за интерфейсом, должна быть скрыта от остальной части программы — с тем чтобы ее можно было изменять, не затронув при этом ничего снаружи. Для этого принципа существует несколько терминов: сокрытие информации (hiding), инкапсуляция, абстракция, модульность и т. п.; все они описывают в общем одни и те же идеи. Интерфейс должен скрывать те детали реализации, которые не имеют отношения непосредственно к клиенту (пользователю интерфейса). Скрытые детали можно изменять, никак не затрагивая этим клиента: таким образом, можно постепенно улучшать интерфейс, наращивать его возможности и даже целиком заменить всю реализацию.

Базовые библиотеки большинства языков программирования дают хорошо известные примеры реализации этого принципа, хотя и не всегда удачно разработанные. Одна из наиболее известных среди них — это стандартная библиотека ввода-вывода в C, в ней содержится несколько десятков функций для открытия, закрытия, чтения, записи и другой обработки файлов. Реализация файлового ввода-вывода скрыта в типе данных FILE*; на самом деле его свойства можно даже посмотреть (они нередко высказаны в <stdio. h>), но использовать не стоит.

Если заголовочный файл содержит только название структуры, а не полное ее описание, то такой тип называют иногда непрозрачным типом, поскольку свойства его неизвестны, а все операции осуществляются через указатель.

Избегайте глобальных переменных; всюду, где это возможно, лучше передавать ссылки на данные через аргументы функций.

Мы настоятельно рекомендуем не делать видимыми никаких данных ни в каком виде, — если пользователи смогут по своему желанию менять значения переменных, то чересчур сложно будет сохранять целостность и непротиворечивость данных. С помощью интерфейсов функций достаточно просто задавать жесткие правила доступа, однако этот прин- | цип часто нарушается. Предопределенные потоки ввода-вывода вроде | stdin и stdout практически всегда определяются как элементы глобального массива структур FILE:

```
    extern FILE __iob[_NFILE];
# define stdin (&__iob[0])
# define stdout (&__iob[1])
 # define stderr (&__iob[2])
```

Таким образом, реализация получается абсолютно прозрачной, при ; этом, несмотря на то что stdin, stdout и stderr выглядят как переменные, присваивать им никаких значений нельзя. Специфическое имя__iob основано на соглашении ANSI C, гласящем, что два подчеркивания используются в начале тех имен служебных переменных, которые должны | быть видны. Таким образом, выбранное нами имя, скорее всего, не будет I конфликтовать с именами внутри самой программы.

Классы в C++ и Java — еще более удачные механизмы для сокрытия информации; их можно считать центральными средствами правильного I использования этих языков. Классы-контейнеры из ЗТЬдля C++, которые мы использовали в главе 3, заходят еще дальше: за исключением некоторых данных о производительности, никакой информации о деталях | реализации не имеется, — следовательно, разработчики библиотеки могут использовать абсолютно любые механизмы.

Ограничьтесь небольшим набором независимых примитивов. Интерфейс должен предоставлять все необходимые возможности, но не более того; части интерфейса по возможности не должны перекрывать друг друга в плане функциональности. С одной стороны, лучше иметь большое количество функций в библиотеке —.тогда можно подобрать любую необходимую комбинацию. С другой стороны, чем больше интерфейс, тем труднее его написать и поддерживать в дальнейшем, а кроме того, неоправданно большие размеры могут привести к тому, что его будет трудно изучить и, следовательно, использовать оптимально. "Интерфейсы прикладных программ" (Application Program Interfaces, или API) зачастую настолько велики, что ни один смертный, похоже, не в состоянии освоить их целиком.

Для удобства использования некоторые интерфейсы предоставляют несколько способов для выполнения той или иной операции; с этой тенденцией надо бороться. Стандартная библиотека ввода-вывода C предоставляет как минимум четыре разные функции для вывода символа в выходной поток:

```
    char c;
putc(c, fp);
fputc(c, fp); fprintf(fp, "%c", e);
fwrite(&c, sizeof(char), 1, fp);
```

Если потоком является stdout, то существует еще несколько возможностей. В принципе, удобно, но не все эти возможности так уж необходимы.

Узкие, специализированные интерфейсы предпочтительнее, чем глобальные, расширенные. Делайте что-то конкретное, и делайте это хорошо. Не добавляйте что-либо в интерфейс только потому, что это несложно сделать; не исправляйте интерфейс из-за ошибок в реализации. Например, вместо того чтобы использовать memcpy как скоростной вариант и memmove как вариант надежный, удобнее было бы иметь одну функцию, которая всегда была бы безопасна, а также быстра — когда это возможно.

Не делайте ничего "за спиной" у пользователя. Библиотечная функция не должна создавать никаких таинственных файлов и переменных или без предупреждения

менять глобальные данные. Весьма аккуратно и обдуманно надо относиться к изменению вообще любых данных в вызывающей программе. Наша функция strtok не отвечает некоторым из перечисленных критериев. Например, сюрпризом для пользователя явится вписывание пустых байтов в середину введенной строки. Использование пустого указателя для обозначения места окончания предыдущего захода является потенциальным источником ошибок, а кроме того, исключает возможность одновременного использования нескольких экземпляров функции. Более логичным было бы создание одной функции, которая делила бы на лексемы исходную строку. Кстати, по аналогичным причинам наша вторая версия на C не может быть использована для работы с двумя входными потоками (вернитесь к упражнению 4-8).

Использование одного интерфейса не должно повлечь за собой применение других интерфейсов только для удобства разработчика интерфейса или реализации. Наоборот, интерфейс должен быть по возможности самодостаточным; если же такой у вас не получается, вы должны абсолютно явно описать все необходимые внешние услуги. В противном случае окажется, что вы взвалили бремя ответственности за поддержку интерфейса на своего клиента (пользователя). В качестве характерного примера можно вспомнить муки управления огромными списками заголовочных файлов в программах на C и C++ — заголовочные файлы могут содержать тысячи строк и содержать ссылки на десятки других заголовочных файлов.

Всегда делайте одинаковое одинаково. Очень важно обеспечить последовательность и систематичность интерфейса. Схожие действия должны выполняться схожими способами. Основные функции str. . . в библиотеке C нетрудно использовать даже без описания, поскольку все они ведут себя практически одинаково: поток данных идет справа налево, так же, как и в операции присваивания, и все они возвращают результирующую строку. Однако в стандартной библиотеке ввода-вывода C предсказать порядок аргументов в функциях трудно. В одних из них аргумент FILE* расположен первым, в других— последним; различается также порядок задания размера и количества элементов. А вот правила интерфейса алгоритмов для контейнеров STL хорошо унифицированы, так что предсказать, как будет вести себя незнакомая функция, совсем просто.

Надо стремиться и к внешнему согласованию интерфейса, то есть к сходству с другими, уже известными интерфейсами. Например, функции mem. . . в библиотеках C проектировались позднее, чем функции str. . ., и следуют их стилю. А стандартные функции ввода-вывода, f read и fwrite было бы куда проще использовать, если бы они больше походили на свои прообразы — read и write. В Unix ключи командной строки предваряются символом "минус", однако один и тот же ключ может иметь совершенно различный смысл — даже в родственных программах.

Если командный интерпретатор операционной системы всегда под-1 ставляет в текст шаблоны поиска вроде * в *. exe, поведение будет единообразным. Но если эту подстановку будут делать отдельные программы, то единообразия ожидать трудно. Web-браузеру для перехода на ссылку достаточно однократного щелчка мыши, а во многих других приложениях для вызова программы и для перехода на ссылку применяется двои-ной щелчок; в результате многие пользователи совершенно автоматически используют двойные щелчки и в web-браузерах.

В одних программных средах изложенных принципов придерживаться проще, чем в других, однако стремиться к их претворению в жизнь надо всегда. Так, например, в C довольно трудно скрыть все детали реализации, но хороший программист не станет злоупотреблять открытостью деталей, поскольку интерфейс не должен быть

привязан к частностям — это противоречит принципу сокрытия информации. Комментарии в заголовочных файлах, имена особого вида (вроде__iob) и тому подобные вещи помогут максимально приблизиться к достойному поведению вашего интерфейса в тех случаях, когда вы не можете сделать этого строгими методами.

Очевидно, что и любой проект интерфейса может быть хорош только до какого-то предела. Даже самый прекрасный интерфейс, используемый сегодня, может стать причиной проблем завтра; однако чем лучше он спроектирован, тем дальше отодвинуто это самое завтра.

Управление ресурсами

Одна из наиболее серьезных проблем, требующих решения при проектировании интерфейса библиотеки (а также класса или пакета), — это управление ресурсами, которыми библиотека распоряжается самостоятельно или совместно с вызывающим ее окружением. Наиболее важным из таких ресурсов является память: кто должен ее выделять и высвобождать? Кроме того, среди других ресурсов есть открытые файлы, а также переменные, значения которых представляют общий интерес. Грубо говоря, проблемы с ресурсами можно разделить на инициализацию, поддержание заданного состояния, совместное использование и копирование, а также высвобождение.

В прототипе нашего пакета CSV для задания начальных значений указателей, счетчиков и прочих подобных вещей применялась статическая инициализация. Однако подобный подход довольно ограничен: мы не можем вернуть библиотеку в начальное состояние после того, как были вызваны какие-либо функции этой библиотеки. Альтернативный способ инициализации — создание отдельной специальной функции, которая бы устанавливала все внутренние переменные в корректные начальные значения. При таком подходе возврат в стартовое состояние возможен в любой момент, даже после вызова функций библиотеки, однако пользователь должен будет сам вызывать эту функцию явным образом. Для этой цели функция reset из второй версии библиотеки могла бы быть сделана видимой (то есть public).

В C++ и Java для инициализации данных внутри класса используются конструкторы. Должным образом определенные конструкторы дают нам гарантию, что все данные класса инициализированы и способа создать неинициализированный объект не существует. Набор конструкторов может поддерживать различные виды инициализации. Так, мы могли бы снабдить Csv конструктором, получающим имя файла, или конструктором, получающим входной поток.

А как насчет копирования информации, обрабатываемой библиотекой, — такой, как вводимые строки и поля? Наша C-программа csvgetline предоставляет прямой доступ к вводимым данным (строкам и полям), возвращая указатели на них. У такого свободного доступа существует ряд недостатков. Пользователь может перезаписать память, так что информация окажется некорректной. Например, выражение вроде

```
strcpy(csvfield(1), csvfield(2));
```

может в целом ряде случаев сработать некорректно, — скорее всего, перезаписав начало второго поля, если оно окажется длиннее первого. Пользователь библиотеки должен сделать копию всей информации, которую нужно будет сохранить после очередного вызова csvgetline. Так, после выполнения вот такого фрагмента кода,

указатель вполне может оказаться неверным, если второй вызов csvgetline приведет к новому выделению памяти для буфера строк:

```
    char *p;
csvgetline(fin);
p = csvfield(1);
csvgetline(fin);
 /* здесь p может оказаться неверным */
```

Версия на C++ безопаснее, поскольку строки в ней являются всего лишь копиями, которые можно менять как заблагорассудится.

Java использует ссылки для обращения к объектам, то есть ко всему, кроме базовых типов вроде int. Это более эффективно, чем создание копий, однако пользователь может быть введен в заблуждение, считая, что ссылка является копией; ошибка подобного рода имела место в ранней Java-версии программы markov. Надо сказать, что данная проблема является вечным источником ошибок при работе со строками C. Не стоит забывать, что при необходимости создания копии методы клонирования позволяют вам сделать и это.

Обратной стороной инициализации или конструирования чего-либо, является его финализация (finalization), или деструкция, — то есть очистка и высвобождение ресурсов после того, как они больше не нужны. Особенно важно высвобождение памяти. Очевидно, что программе, которая не высвобождает неиспользуемую память, этой самой памяти в какой-то момент не хватит. Как ни странно, большая часть современных программ страдает этим недостатком. Схожая проблема возникает и в ситуации, когда приходит время закрывать открытые файлы: если данные были буферизованы, этот буфер нередко надо уничтожить (а память, занимаемую им, очистить). Для функций стандартной библиотеки C высвобождение происходит автоматически после нормального окончания работы программы, все остальные случаи должны обрабатываться программой. В C и C++ стандартная функция atexit предоставляет способ получить управление непосредственно перед тем, как программа будет завершена нормально; создателям интерфейсов не стоит пренебрегать такой возможностью для высвобождения ресурсов.

Высвобождайте ресурсы на том же уровне, где выделяли их. Хороший способ управления выделением и высвобождением ресурсов —возложить ответственность за освобождение ресурса на ту же библиотеку, пакет или интерфейс, которые выделяют этот ресурс. Можно выразить эту мысль и другими словами: состояние ресурса не должно меняться в пределах интерфейса. Все функции наших библиотек CSV считывали данные из уже открытых файлов, и по окончании работы они оставляли файлы открытыми. Закрытием файлов должны были заниматься те, кто их открывал, то есть пользователи библиотеки.

Конструкторы и деструкторы C++ помогают строго выполнять это правило. Когда экземпляр класса выходит из области видимости или явным образом уничтожается, вызывается деструктор. В этом деструкторе можно уничтожать буферы, освобождать память, возвращать значения в исходное состояние и делать вообще все, что необходимо. В Java подобного механизма нет. Можно определить для класса метод финали-зации, однако нельзя быть уверенными, что он будет выполнен вообще, не говоря уже о том, чтобы выполниться в какое-то конкретное время. Таким образом, нельзя дать гарантий, что действия по высвобождению ресурсов будут выполнены, хотя зачастую можно предполагать, что это все же произойдет.

В Java, однако, существует механизм, оказывающий огромную помощь в управлении ресурсами, — встроенная сборка мусора (garbage collection). При запуске программы выделяется память под новые объекты. Способа удалить их явным образом просто нет, однако некая система времени исполнения отслеживает, какие объекты все еще используются, а какие нет, и периодически удаляет неиспользуемые.

Существуют различные способы реализации сборки мусора. В некоторых схемах отслеживается счетчик ссылок (reference count) — некоторое число, показывающее, сколькими объектами используется интересующий нас объект. Объект высвобождается, как только счетчик ссылок становится равным нулю. Эту технологию можно реализовать явным образом в C и C++ для управления совместно используемыми объектами. Другой алгоритм периодически ищет связи между выделенной областью памяти и всеми объектами, на которые имеются ссылки. Объекты, обнаруживаемые при этом, кем-то используются, объекты же, на которые никто не ссылается, соответственно, не используются и могут быть уничтожены.

Наличие автоматической сборки мусора не означает, что при проектировании можно оставить вопросы управления ресурсами без внимания. Нам все равно надо определить, возвращает ли интерфейс ссылки на совместно используемые объекты или их копии, а это оказывает большое влияние на всю программу. И вообще, бесплатной сборки мусора не бывает, за нее приходится платить дополнительными расходами на поддержание информации и высвобождение неиспользуемой памяти; кроме того, невозможно предсказать моменты, когда эта сборка мусора заработает.

Все описанные проблемы становятся еще более запутанными, если библиотека должна использоваться в среде, где ее функции могут исполняться одновременно в нескольких нитях управления — как, например, в многонитевой программе на Java.

Чтобы избежать лишних проблем, необходимо писать реентерабельный (reentrant, повторно вызываемый) код, то есть код, который бы работал вне зависимости от количества одновременных его вызовов. В реентерабельном коде не должно быть глобальных переменных, статических локальных переменных, а также любых других переменных, которые могут быть изменены в то время, как их использует другая нить. Основой хорошего проекта многонитевой программы является такое разделение компонентов, при котором они не могут ничего использовать совместно иначе, чем через должным образом описанный интерфейс. Библиотеки, в которых по небрежности переменные доступны для совместного использования, способны разрушить многонитевую модель. (В многонитевой программе использование st rtok может привести к ужасным последствиям, поскольку существуют другие функции из библиотеки C, которые хранят значения во внутренней статической памяти.) Если переменная может быть использована несколькими процессами, то необходимо предусмотреть некий блокирующий механизм, который бы давал гарантию, что в любой момент времени с ними может работать только одна нить. Здесь очень полезны классы, поскольку они создают основу для обсуждения моделей совместного использования и блокировки. Синхронизированные методы в Java предоставляют нити управления способ заблокировать целый класс или его экземпляр от одновременного изменения другой нитью; синхронизированные блоки разрешают только одной нити за раз выполнять фрагмент кода.

Многонитевое управление добавляет немало новых сложностей во многие аспекты проектирования и программирования; тема эта чересчур обширна, чтобы обсуждать ее в деталях на страницах этой книги.

Abort, Retry, Fail?

В предыдущих главах мы использовали для обработки ошибок функции вроде eprintf и estrdup — просто выводили некие сообщения перед тем, как прервать выполнение программы. Например, функция eprintf ведет себя так же, как fprintf (stderr, . . .), но после вывода сообщения выходит из программы с некоторым статусом ошибки. Она использует заголовочный файл <stdarg. h> и библиотечную функцию vfprintf для вывода аргументов, представленных в прототипе многоточием (...). Использование библиотеки stdarg должно быть начато вызовом va_start и завершено вызовом va__end. Мы еще вернемся к этому интерфейсу в главе 9.

```
    # Include <stdarg.h>
 # ifinclude <string.h>
# Ifinclude <errno. h>
  /* eprintf: печать сообщения об ошибке и выход */
 void eprintf(char *fmt, ...)
{
va_list args;
fflush(stdout);
if (progname() != NULL)
fprintf(stderr, "%s: ", prognameO);
 va_start(args, fmt); vfpnntf(stderr, fmt, args);
 va_end(args);

    if (fmt[0] != ДО' && fmt[strlen(fmt)-1] == ':')
fprintf (stderr, " %s", strerror(errno));
 fprintf(stderr, "\n");
exit(2); /* общепринятое значение
                          */
/* для ненормального завершения работы */
 }
```

Если аргумент формата оканчивается двоеточием (:), то eprintf вызывает стандартную функцию st re г го г, которая возвращает строку, содержащую всю доступную дополнительную системную информацию об ошибке. Мы написали еще функцию weprintf, сходную с eprintf, которая выводит предупреждение, но не завершает программу. Интерфейс, схожий с printf, удобен для создания строк, которые могут быть напечатаны или выданы в окне диалога.

Сходным образом работает est rdup: она пытается создать копию строки и, если памяти для этого не хватает, завершает программу с сообщением об ошибке (с помощью eprintf):

```
    /* estrdup: дублирует строку; */
 /* при возникновении ошибки сообщает об этом */
char *estrdup(char *s)
 {
char *t;
     t = (char *) malloc(strlen(s)+1);
if (t == NULL)
 eprintf("estrdup(\"%.20s\") failed:", s);
 strcpy(t, s);
 return t;
 }
```

Функция emalloc предоставляет аналогичные возможности для вызова malloc:

```
     /* emalloc: выполняет malloc; */
/* при возникновении ошибки сообщает об этом */
 void *emalloc(size__t n)
 {
 void *p;
 p = malloc(n); if (p == NULL)
epnntf("malloc of %u bytes failed:", n);
return p;
 }
```

Эти функции описаны в заголовочном файле eprintf. h:

```
    /* eprintf.h: функции, сообщающие об ошибках */
extern void eprintf(char *, ...);
extern void weprintf(char *, ...);
 extern char *estrdup(char *);
 extern void *emalloc(size_t);
extern void *erealloc(void *', size_t);
extern char *progname(void);
 extern void setprogname(char *);
```

Он включается в любой файл, вызывающий одну из функций, которые сообщают об ошибке. Каждое сообщение об ошибке содержит имя программы, определенное вызывающим кодом, — оно устанавливается и извлекается простейшими функциями set prog name и prog name, описанными в том же заголовочном файле и определенными в исходном файле вместе с eprintf:

```
    static char *name = tfllLL; /* имя программы для сообщений */
 /* setprogname: устанавливает хранимое имя программы */
 void setprogname(char *str)
 {
 name = estrdup(str);
 }
/* progname: возвращает хранимое имя программы */
 char *progname(void)
 {
 return name;
}
```

Типичный пример использования выглядит примерно так:

```
     int main(int argc, char *argv[])
 {
 setprogname("markov");
 f = fopen(argv[i], "r");
 if (f == NULL)
 eprintf("can't open %s:", argv[i]);
 }
```

что приводит к появлению сообщений вроде

```
markov: can't open psalm.txt: No such file or directory
```

Мы считаем эти "оберточные" функции вполне подходящими для наших собственных программ, поскольку они унифицируют обработку ошибок; кроме того, само их присутствие вдохновляет на поиск ошибок. Ничего сложного или особо выдающегося в них нет, так что вы можете запросто придумать для себя какие-то более подходящие варианты.

Представим теперь, что вместо создания функций для собственного использования нам надо разработать библиотеку, с которой будут работать другие программисты. Что должна делать функция из этой библиотеки при возникновении ошибки? Те функции, что мы только что написали, выводят сообщение и умирают. Для многих программ, особенно для небольших самостоятельных утилит, такое поведение может'быть вполне приемлемым. Для других же программ простой выход не годится, поскольку при этом другие части программы лишаются возможности хотя бы попытаться вернуться в нормальное состояние; характерным примером являются текстовые редакторы, — в них стоит приложить максимум усилий для сохранения редактируемого документа. В некоторых ситуациях библиотечные функции не должны даже выдавать никакого сообщения, поскольку существуют системы, где такое сообщение будет мешать отображению полезной информации или же, наоборот, просто сгинет бесследно. Для подобных случаев полезно записывать сообщения в некий отдельный журнальный файл (log file), который можно просматривать независимо.

Обнаруживайте ошибки на низком уровне, обрабатывайте на высоком. Существует общий принцип: ошибки должны обнаруживаться на самом низком уровне, какой только возможен; обрабатывать же их надо на высоком уровне. В большинстве случаев определять способ обработки ошибки должен вызывающий код, а не вызываемый. Библиотечные функции могут помочь в этом, обеспечивая приемлемую реакцию при сбоях, — например, при получении несуществующего поля в качестве аргумента не прерывать работу всей программы, а возвращать NULL. Или, как в csvgetline, возвращать NULL вне зависимости от того, сколько раз эта функция была вызвана после достижения конца файла.

Не всегда очевидно, какие же значения должны возвращаться при ошибках; мы уже сталкивались с проблемой возвращаемого значения у функции csvgetline. Хотелось бы, конечно, возвращать как можно более содержательную информацию, но при этом в такой форме, чтобы остальная часть программы могла использовать ее без труда. В C, C++ и Java это значит, что информация должна возвращаться в качестве результата функции и, возможно, в значениях параметров-ссылок (указателей). Многие библиотечные функции умеют различать обычные значения и специальные значения ошибок. Функции ввода типа getcha r возвращают значение, конвертируемое в char для нормальных данных, и некоторое неконвертируемое в char значение, например EOF, для обозначения конца файла или ошибки.

Этот механизм, однако, не работает, если функция может возвращать любые значения из возможного диапазона. Например, математические функции вроде log могут возвращать любое число с плавающей точкой. В стандарте IEEE для чисел с плавающей точкой предусмотрено специальное значение NaN ("not a number" — не число), означающее ошибку, — это значение и возвращается функциями в случае ошибки.

Некоторые языки, такие как Perl и Tel, предоставляют несложный способ группировки двух и 0олее значений в кортеж (tuple). В таких языках значение функции и код

ошибки можно без проблем передавать совместно. В C++ STL имеется тип данных pai r, который можно использовать таким же образом.

Хотелось бы, по возможности, уметь различать исключительные значения типа конца файла или кода ошибок, а не запихивать их все в какое-то одно значение. Если значения нельзя разделить сразу же, можно поступить таким образом: возвращать одно значение для всех видов исключительных ситуаций и создать дополнительную функцию, которая бы возвращала дополнительную информацию об ошибке.

Именно такой подход используется в Unix и стандартной библиотеке C: многие системные вызовы и библиотечные функции возвращают в случае ошибки -1 и при этом устанавливают глобальную переменную errno; функция strerror возвращает строку, соответствующую номеру ошибки. В нашей системе программа

```
   # include <stdio.h>
# ((include <string. h>
# include <errno.h>
# include <math.h>


/* errno main: тестирование библиотеки */
int main(void)
 ; \
double f;
errno = 0;
/* очищаем переменную кода ошибки */
f = log(:-;1.23);
printf("%f %d %s\n", f, errno, strerror(errno));

    return 0;
}
```

напечатает

```
    nanOxЮОООООО 33 Domain error
```

Обратите внимание на то, что errno должна быть предварительно очищена (как в приведенной программе), тогда при возникновении ошибки она установится в некоторое ненулевое значение.

Используйте исключения только для исключительных ситуаций. В некоторых языках для отлова нестандартных ситуаций и восстановления после них имеется специальный механизм исключений, или исключительных ситуаций (exceptions); таким образом предоставляется альтернативный способ управления работой программы при возникновении каких-либо проблем. Исключения не следует использовать для обработки обычных возвращаемых значений. Так, при чтении файла рано или поздно будет достигнут его конец; это должно обрабатываться посредством возвращаемого значения, а не исключения.

Рассмотрим такой фрагмент, написанный на Java:

```
   String fname = "someFileName"; try {
 FileInputStream in = new FileInputStream
(fname);
```

```
int c;
while ((c = in. read()) != -1)
System.out.print((char) c);
 in.close();
}
catch (FileNotFoundException e)
 {
System.err.println(fname + " not found");
}
catch (IOException e)
{
 System.err.println("IOException: " + e);
e.printStackTrace();
 }
```

Этот цикл считывает символы, пока не будет достигнут конец файла — ожидаемое событие, которое функция read отмечает возвратом значения -1. Однако, если файл не может быть открыт, возникает (или, как принято говорить, возбуждается) исключение, а не установка переменной in в null, как это было бы сделано в C или C++. Наконец, если в блоке t ry происходит какая-то другая ошибка ввода, также возбуждается исключение, обрабатываемое в блоке IOException.

Не стоит злоупотреблять исключениями: они сильно видоизменяют управляющую логику, что ведет к появлению достаточно сложных логических конструкций — потенциальных слабых мест программы. Вряд ли при неудачной попытке открыть файл, например, стоит возбуждать исключение. Последние лучпф оставить для действительно непредвиденных случаев вроде отсутствия свободного места на диске или.ошибок арифметики с плавающей точкой.

В C пара функций — setjmp и longjmp — предоставляет возможность реализовать механизм исключений на гораздо более низком уровне, но это настолько сложно, что мы не будем описывать, как это сделать.

Как насчет восстановления ресурсов при возникновении ошибки? Должна ли библиотека предпринимать попытки такого восстановления, если что-то идет не так, как надр? Как правило, нет, однако очень неплохо предусмотреть какой-то механизм, позволяющий удостовериться, что информация сохранилась в максимально корректной форме. Естественно, неиспользуемое пространство памяти должно быть высвобождено. Если же к каким-то, переменным еще возможен доступ, они должны быть установлены в осмысленные значения. Распространенной причиной ошибок является использование указателя на уже освобожденную память. Чтобы не попасться на эту удочку, достаточно в коде обработки ошибки, который высвобождает что-то, установить указатель, адресующийся к этому чему-то, в ноль. Функция reset во второй версии библиотеки CSV как раз и являлась нашей попыткой преодолеть некоторые из описанных проблем. Обобщая же все вышесказанное, отметим: надо добиваться того, чтобы библиотека оставалась пригодна к использованию даже после возникновения ошибки.

Пользовательские интерфейсы

До сих пор мы говорили главным образом об интерфейсах между компонентами программы или несколькими программами. Но есть же и еще один, очень важный, вид интерфейса — между программой и ее пользователями-людьми.

Большинство примеров программ в этой книге основаны на работе с текстом, так что их пользовательский интерфейс представляется более-менее очевидным. В предыдущем разделе мы выяснили, что ошибки надо отслеживать и сообщать о них; при необходимости должны предприниматься попытки восстановления. Сообщение об ошибке должно включать в себя всю доступную информацию и быть максимально информативным для каждого конкретного контекста; незачем выводить

```
estrdup failed
```

когда можно сообщить

```
markov: estrdup("Derrida") неудача: мало места в памяти
```

Нам ничего не стоит включить дополнительную информацию (вспомните, как мы это делали в estrdup), а пользователю это может помочь идентифицировать проблему или хотя бы просто подобрать корректные входные данные.

Если пользователь допустил ошибку, программа должна показать ему пример правильного ввода, как это сделано в функциях типа

```
   /* usage: печатает подсказку и выходит */
 void usage(void)
  {
 fprintf(stderr, "usage: %s [-d] [-n nwords]"
" [-s seed] [files ...]\n", prognameO);
 exit(2);
 }
```

Имя программы, вырабатываемое функцией prog name, идентифицирует источник сообщения. Это особенно важно в случае, если программа является частью какого-то большого процесса. Если программа будет выводить сообщения вроде syntax error или estrdup failed, то пользователь может просто не понять, откуда пришло сообщение.

Текст сообщений об ошибке, подсказок и окон диалога должен обязательно четко описывать допустимые значения: не утверждайте, что параметр слишком велик, а приведите диапазон допустимых значений для этого параметра. Когда это возможно, текст должен сам по себе являться корректным вводом, например полной командной строкой с правильно заданным параметром. Это не только даст возможность пользователю понять, чего же от него ждут, но и позволит сохранить такой выводимый текст в файле (или "вырезать" его с помощью мыши) и потом использовать для запуска какого-то следующего процесса. Здесь, кстати, сразу становится виден один из недостатков окон диалога: их содержимое довольно трудно запомнить для дальнейшего использования.

Эффективный способ создать хороший пользовательский интерфейс для ввода — спроектировать специализированный язык для установки параметров, контролирования действий и т. п. Интерфейсы, основанные на языках, мы подробно обсудим в главе 9.

Защитное программирование, то есть такое программирование, при котором можно быть уверенным, что программе не страшен никакой некорректный ввод, не только защитит пользователя от самого себя и своих ошибок, но и предохранит всю систему в целом. Об этом речь пойдет в главе 6, посвященной тестированию программ.

Большинство людей пользуется сейчас графическими интерфейсами. Графические пользовательские интерфейсы — отдельная большая тема, поэтому мы упомянем лишь о нескольких связанных с ними моментах. Во-первых, графический интерфейс трудно сделать "правильным", поскольку его пригодность и удобство оцениваются пользователями субъективно. Во-вторых, с чисто практической точки зрения в системе с графическим пользовательским интерфейсом размер кода, обрабатывающего взаимодействие с пользователем, как правило, гораздо больше; чем код для любого самого сложного алгоритма.

Тем не менее в проектировании как внутренней реализации, так и наружного дизайна пользовательского интерфейса действуют одни и те же принципы. С точки зрения пользователя, хорошая проработка вопросов стиля — простоты, прозрачности, стандартности, предсказуемости, привычности и строгости — является синонимом хорошего интерфейса; отсутствие же перечисленных качеств наверняка приведет к зачислению интерфейса в разряд неудобных.

Стандартность и привычность интерфейса крайне желательны; это требование включает в себя последовательное использование терминов, модулей, форматов, шрифтов, цветов, размеров и всех остальных составляющих графическую среду элементов. Сколько различных английских слов используется для выхода из программы или закрытия окна? С десяток — от Abandon до control-Z; подобная непоследовательность может слегка запутать даже пользователя, для которого английский является родным языком, иностранца же она просто заводит в тупик.

Внутри кода, работающего с графикой, интерфейсам следует уделить особое внимание, поскольку эти системы, как правило, велики и сложны, а процесс ввода данных (и вообще получение реакции пользователя) весьма нетривиален. В разработке графических пользовательских интерфейсов большим преимуществом обладает объектно-ориентированная модель программирования, поскольку она предоставляет способ инкапсуляции состояний и поведения окон. При этом используется наследование для объединения одинаковых моментов в базовые классы и вынесения различий в классы-наследники.

Дополнительная литература

Несмотря на то что ряд технических деталей, описанных в книге "Мистический человекомесяц" Фредерика Брукса (Frederick P. Brooks, Jr. The Mythical Man Month. Addison-Wesley, 1975; Anniversary Edition, 1995)' уже устарел, она не перестала быть захватывающе интересной . и во многом столь же актуальной сегодня, как и двадцать лет назад.

Практически в каждой книге по программированию есть что-то интересное о проектировании интерфейсов. Практическим пособием, созданным на основе большого, потом и кровью добытого опыта, является книга "Разработка крупномасштабных программ на C++" Джона Лакоса (John Lakos. Large-Scale C++ Software Design. Addison-Wesley, 1996). В этой книге обсуждаются проблемы создания и управления действительно большими программами на C++. В создании программ на C поможет труд Дэвида Хэнсона "Си: интерфейс и реализация" (David Hanson. C Inter/aces and Implementations. Addison-Wesley, 1997).

Отличным рассказом о том, как писать программы в команде, является книга Стива Мак-Коннелла "Быстрая разработка" (Steve McCon-nell's. Rapid Development. Microsoft Press, 1996). В ней, кстати, особое внимание уделяется роли прототипа программы.

О проектировании графических пользовательских интерфейсов написано немало книг, авторы которых затрагивают различные аспекты этого процесса. Мы советуем:

- Kevin Mullet, Darrell Sano. Designing Visual Inter/aces: Communication Oriented Techniques. Prentice Hall, 1995;
- Ben Shneiderman. Designing the User Inter/ace: Strategies/or Effective Human-Computer Interaction. 3rd ed. Addison-Wesley, 1997;
- Alan Cooper. About Face: The Essentials of User Interface Design. IDG, 1995;
- Harold Thimbleby. User Interface Design. Addison-Wesley, 1990.
- Брукс-мл. Ф. П. Как проектируются и создаются программные комплексы. М.: Наука, 1979; новое издание перевода: Мистический человекомесяц. СПб.: СИМБОЛ+, 1999.

Отладка

- Отладчики
- Хорошие подсказки, простые ошибки
- Трудные ошибки, нет зацепок
- Последняя надежда
- Невоспроизводимые ошибки
- Средства отладки
- Чужие ошибки
- Заключение
- Дополнительная литература

bug ("жучок", "баг").
b. Дефект или/неполадка в машине, плане и т. п. Происх. — США.
"Пэл Мэл Газет", 1889, 11 марта, 1/1. Мистер Эдисон, как я слышал, провел две бессонных ночи, отыскивая "жучка" в своем фонографе, — это выражение означает решение сложной проблемы, его использование подразумевает, что где-то внутри спряталось какое-то воображаемое насекомое, которое и вызывает все проблемы.

Oxford English Dictionary, 2"d Edition

В предыдущих четырех главах мы продемонстрировали много различного кода и при этом притворялись, что весь этот код работал должным образом с первого раза. Естественно, это было не так: на самом деле было множество "багов". Слово "баг" появилось вовсе не среди программистов, но считается одним из самых распространенных терминов в программировании. Почему программирование столь сложно?

Одна из причин сложности программы заключается в большом количестве способов, с помощью которых могут взаимодействовать ее компоненты, а уж программы полны и компонентами, и взаимосвязями между ними. Многие технологии пытаются сократить связи между компонентами, чтобы уменьшить количество взаимодействий: например, используется сокрытие информации, абстрагирование и интерфейсы, а также все возможности языков, способствующие этим технологиям. Существуют также технологии для проверки целостности архитектуры программы: доказательства корректности программ, моделирование, анализ требований, формальные проверки. Ни одна из перечисленных технологий не изменила радикально способа создания программ: они работают лишь на небольших задачах. В реальности всегда будут ошибки, которые мы находим с помощью тестирования и устраняем с помощью отладки (debugging).

Хорошие программисты знают, что они проведут столько же времени, отлаживая программу, сколько они ее и писали, и поэтому стараются учиться на своих ошибках. Каждая найденная ошибка сможет научить вас, как предотвратить появление подобной ошибки в будущем и как справиться с ней, если она все же появится.

Отладка сложна и может занимать непредсказуемо долгое время, поэтому цель в том, чтобы миновать большую ее часть. Технические приемы, которые помогут

уменьшить время отладки, включают хороший дизайн, хороший стиль, проверку граничных условий, проверку правильности (исходных) утверждений и разумности кода, защитное программирование, хорошо разработанные интерфейсы-, ограниченное использование глобальных данных, средства контроля и проверки. Грамм профилактики стоит тонны лечения.

Какова роль языка? Основной движущей силой в эволюции языков программирования была попытка предотвратить ошибки с помощью возможностей языка. Некоторые такие возможности уменьшают шанс появления целых классов ошибок: проверка диапазонов индексов, ограничение использования указателей или полный отказ от них, сборка мусора, строковые типы данных, типизированный ввод-вывод, строгая проверка типов. Однако некоторые возможности языка напрашиваются на ошибку, например оператор goto, глобальные переменные, свободно используемые указатели, автоматические преобразования типов. Программистам следует знать зоны повышенного риска в своих языках и быть особенно осторожными при их использовании. Следует также включить все проверки компилятора и слушаться его предупреждений.

Каждая возможность в языке предотвращает одну проблему, но при этом имеет свою цену. Если в языке высокого уровня простые ошибки исчезают автоматически, то ценой этого станет большая вероятность совершать ошибки высокого уровня. Никакой язык не защитит вас от ошибок полностью.

Как бы нам ни хотелось обратного, но основное время при программировании тратится на тестирование и отладку. В этой главе мы обсудим, как сократить время, которое вы тратите на отладку, и как использовать это время наиболее продуктивно; к вопросам тестирования мы вернемся в главе 6.

Отладчики

Компиляторы основных языков программирования обычно поставляются со сложными отладчиками, часто входящими в состав среды программирования, которая объединяет в себе создание и редактирование исходного кода, компиляцию, выполнение и отладку. Отладчики включают в себя графический интерфейс для пошагового выполнения программы, оператор за оператором или функция за функцией, с остановками на конкретных строках программы или при достижении какого-то условия. Они также предоставляют возможность форматирования и отображения значений переменных.

Отладчик можно использовать непосредственно, если существующая проблема точно известна. Некоторые отладчики включаются автоматически, если во время выполнения программы что-то происходит не так, как следует. Обычно довольно легко обнаружить, в каком месте выполнялась программа, если она неожиданно аварийно завершилась, при этом можно рассмотреть последовательность функций, выполнявшихся в тот момент (это называется "просмотр стека вызовов"), а также отобразить значения локад.ьных и глобальных переменных. Этой информации бывает достаточно,/чтобы выявить ошибку. В противном случае можно повторно запустить программу в пошаговом режиме, чтобы обнаружить, где именно начинается неверное поведение.

В правильной среде программирования, в руках опытного пользователя, хороший отладчик делает отладку эффективной и быстрой, чуть ли не безболезненной. Почему, имея в распоряжении столь мощные инструменты, кто-то будет заниматься отладкой без них? Почему мы отводим отладке целую главу?

Тому есть несколько причин, некоторые — вполне объективные, а другие основаны на личном опыте. Часть менее распространенных языков программирования не имеет отладчиков или обеспечивает лишь рудиментарные возможности отладки. Отладчики системно-зависимы, так что вы можете оказаться в системе, в которой нет привычного вам отладчика. Некоторые программы не очень хорошо поддаются отладке: многопроцессные или многонитевые программы, операционные системы, распределенные системы зачастую должны отлаживаться более низкоуровневыми средствами. В таких ситуациях вы можете полагаться только на себя, и немногие вещи могут вам помочь: операторы выдачи сообщений на экран, личный опыт и способность рассуждать, глядя на код.

Наш личный выбор — стараться не использовать отладчики, кроме как для просмотра стека вызовов или же значений пары переменных. Одна из причин этого заключается в том, что очень легко потеряться в деталях сложных структур данных и путей исполнения программы; мы считаем пошаговый проход по программе менее продуктивным, чем усиленные размышления и код, проверяющий сам себя в критических точках. Щелканье по операторам занимает больше времени, чем просмотр сообщений операторов отладочной выдачи, расставленных в критических местах. Быстрее решить, куда поместить оператор отладочной выдачи, чем проходить шаг за шагом критические участки кода, даже предполагая, что мы знаем, где находятся такие участки. Более важно то, что отладочные операторы сохраняются в программе, а сессии отладчика преходящи.

Слепое блуждание в отладчике, скорее всего, непродуктивно. Полезнее использовать отладчик, чтобы выяснить состояние программы, в котором она совершает ошибку, а затем подумать о том, как такая ошибка могла возникнуть. Отладчики могут быть запутанными и сложными программами, особенно для новичков, которым они принесут больше недоумения, чем помощи. Если задать отладчику неправильный вопрос, то он, скорее всего, даст вам ответ, и вы не догадаетесь, куда этот ответ заведет вас.

Отладчик, однако же, может иметь невероятное значение, и вам обязательно надо включить его в свой набор инструментов; скорее всего, отладчик — первое, к чему вы прибегнете. Но даже если отладчика у вас нет или вы застряли на особенно сложной проблеме, все равно технические приемы, рассмотренные в этой главе, позволят вам отлаживаться эффективно и быстро. Они также помогут увеличить продуктивность использования отладчика, потому что в основном эти приемы связаны с рассуждениями об ошибках и вероятных причинах их появления.

Хорошие подсказки, простые ошибки

Ой! Что-то случилось. Моя программа "свалилась", напечатала какой-то мусор или, кажется, "зависла". Что мне делать?

Начинающие обычно винят в происшедшем компилятор, библиотеку или еще что-нибудь, но только не свой код. Опытные программисты были бы счастливы сделать то же самое, но они-то знают, что проблема, скорее всего, заключается в их собственной ошибке.

К счастью, в большинстве своем ошибки просты, и их можно обнаружить с помощью простых приемов. Изучите улики — неверные результаты работы и попытайтесь догадаться, как такие результаты могли возникнуть. Посмотрите на отладочную выдачу перед аварийным завершением; если возможно, получите у отладчика стек вызовов. Теперь вы уже кое-что знаете о том, что именно произошло и где.

Остановитесь, подумайте. Как такое могло случиться? Рассуждайте, исходя из состояния "свалившейся" программы, чтобы определить причину.

Процесс отладки включает в себя обратную трассировку (backward reasoning) — прослеживание событий в обратном порядке, как в детективе. Случилось что-то невозможное, и единственное, что известно точно, — невозможное случилось. Для того чтобы раскрыть причины, нужно мысленно проходить обратный путь от результата к возможной причине. Когда у нас имеется полное объяснение, мы знаем, что именно исправлять и, по ходу дела, скорее всего, обнаружим несколько других вещей, которых мы не ожидали.

Ищите знакомые ситуации. Спросите себя, известна ли уже вам эта ситуация. "Я уже видел это" — с этой фразы часто начинается понимание, а иногда даже и возникает ответ. Обычные ошибки имеют четко различимые признаки. Например, начинающие программисты на C часто пишут

```
? int n
```

вместо

```
  int n;
scant ("$ &п);
```

При такой попытке ввода значения обычно возникает ошибка обращения за пределы доступной памяти. Преподаватели языка C немедленно узнают этот симптом.

Несовпадающие типы и преобразования при вызове printf и scant рождают бесконечный поток тривиальных ошибок:

```
? int n = 1;
? double d = PI;
? pnntf("%d %f\n", d, n);
```

Признаком этой ошибки иногда бывают абсурдные значения переменных: огромные целые, невероятно большие или невероятно маленькие значения с плавающей точкой. На Sun SPARC эта программа выводит огромное целое и астрономическое число с плавающей точкой (выдача отформатирована, чтобы не выходить за поля страницы):

```
1074340347 2681561585988520015341087942602333963 50\
19365859717932180477149637953077886114805641 40\
07968212895947435371511635241011754740847641 56\
42277140832383962343014 4.000000
```

Другой обычной ошибкой является использование %f вместо %lf, когда значение типа double читается с помощью scanf. Некоторые компиляторы ловят такие ошибки, проверяя, соответствуют ли типы аргументов scanf и printf параметрам форматной строки; если вывод всех предупреждений компилятора разрешен, то относительно приведенного выше обращения к printf компилятор GNU gcc сообщит

```
  x.c:9: warning: int format, double arg (arg 2)
x.c:9: warning: double format, different type arg (arg 3)
```

Неинициализированные локальные переменные — еще один источник четко отличимых ошибок. Результатом часто являются слишком большие значения, возникшие из-за мусора, оставшегося в этом месте памяти от другой переменной. Некоторые компиляторы предупредят вас, если вы включите это предупреждение, но часть случаев они отследить все же не могут. Память, выделенная функциями типа malloc, realloc и new, скорее всего, также содержит мусор; обязательно инициализируйте ее.

Проверьте самое последнее изменение. В чем оно заключалось? Если в процессе разработки вы изменяете только один участок за раз, то ошибка, как правило, находится в новом коде или же в участке старого кода, который используется из нового кода. Тщательно посмотрите на последние изменения, это поможет локализовать проблему. Если ошибка появляется в новой версии, а в старой ее нет, следовательно, новый код является частью проблемы. Это означает, что вам следует сохранять как минимум предыдущую версию программы, ту, которую вы считаете правильной, чтобы можно было сравнить поведение версий. Это также означает, что вам следует делать записи об изменениях и исправленных ошибках, чтобы не пришлось переоткрывать эту информацию при попытках исправления ошибок. Здесь будут полезны системы контроля исходных текстов и другие механизмы хранения истории.

Не повторяйте дважды туже самую ошибку. После того как вы исправите ошибку, спросите себя, не совершали ли вы подобной ошибки когда-то раньше. Такая история случилась с нами буквально за несколько дней до того, как мы писали эту главу. Для нашего коллеги была написана программа-прототип, которая включала в себя стереотипную конструкцию для разборки опций:

```
    ? for (i = 1; i < argc; i++) {
? if (argv[i][0] != '-') /* аргументы кончились */
? break;
? switch (argv[i][1]) {
? case 'o1: /* имя выходного файла */
? outname = argv[i];
? break;
 ? case T :
 ? from = atoi(argv[i]);
? break;
? case 't' :
? to = atoi(argv[i]);
? break;
  ...
```

Довольно скоро nocjje опробования программы наш коллега сообщил, что имя выходного файла всегда начиналось с -o. Это было обидно, но, как оказалось, легко исправимо: код следовало читать так:

```
    outname = &argv[i][2];
```

Программа была исправлена и отослана обратно, а затем пришла опять с сообщением, что программа не обрабатывала должным образом аргументы типа - f 123: преобразованное числовое значение всегда содержало ноль. Это та же самая ошибка: следующая часть оператора выбора должна была звучать так:

```
    from = atoi(&argv[i][2]);
```

Из-за того, что автор торопился, он не заметил, что тот же самый промах произошел еще в двух местах, и понадобился еще один круг, чтобы полностью исправить все практически одинаковые ошибки.

В простом коде могут быть ошибки, если привычность этого кода такова, что заставляет нас ослабить внимание. Даже если код столь прост, что вы можете написать его во сне, не засыпайте, пока его пишете.

Не откладывайте отладку на потом. Чрезмерная торопливость может повредить и в других ситуациях. Не игнорируйте проявившуюся ошибку: отследите ее прямо сейчас, потому что потом она может и не возникнуть. Пример — знаменитая история, случившаяся при запуске космической станции "Mars Pathfinder". После безупречного "приземления" в июле 1997 года компьютеры станции имели обыкновение перезагружаться в среднем один раз в день, и это поставило инженеров в тупик. Когда они отследили ошибку, то поняли, что уже встречались с ней. Во время предпусковых проверок такие перезагрузки случались, но были проигнорированы, потому что инженеры работали над другими вопросами. Теперь они оказались вынуждены решать проблему, когда машина находится на расстоянии десятков миллионов километров, и исправить ошибку стало значительно труднее.

Пользуйтесь стеком вызовов. Хотя отладчики умеют обращаться с программами и в процессе их работы, все же одним из основных их применений является исследование "посмертного" состояния программы. Номер строки исходного текста, в котором произошла ошибка, или, зачастую, кусок стека вызовов — это самая полезная отладочная информация. Хорошей подсказкой также бывают невероятные значения аргументов (нулевые указатели, огромные целые, тогда как они должны быть небольшими, или отрицательные, когда они должны быть положительными, строки, состоящие из неалфавитных символов).

Вот типичный пример, основанный на обсуждении сортировки из главы 2. Для того чтобы отсортировать массив целых, нужно вызвать qsort с функцией сравнения целых чисел icmp:

```
    nt arr[N];
qsort(arr, N, sizeof (arr[0]), icmp);i
```

Предположим, что мы по недосмотру передаем вместо icmp функцию сравнения строк scmp:

```
    ?int arr[N];
? qsort(arr, sizeof(arr[0]), scmp);
```

Компилятор не может обнаружить несовпадения типов, поэтому неприятность ожидает своего часа. Когда мы запускаем программу, она "валится", пытаясь обратиться к неразрешенному адресу. Отладчик dbx выдает такую трассировку стека вызовов:

```
    0 strcmp(0x1a2, Ox1c2)
["strcmp.s":31]
1 scmp(p1 = 0x10001048, p2 = Ox1000105c)
["badqs.c": 13]
```

```
 2 qst(0x10001048, 0x10001074, Ox400b20, 0x4)
 ["qsort.c":147]
3 qsort(0x10001048, Ox1c2, 0x4, Ox400b20)
 [ "qsort.c" :63]
4 main() ["badqs.c":45]
 5 __iatart() ["crt1tinit.s":13]
```

Это означает, что программа "погибла" в функции st rcmp; при изучении ситуации становится ясно, что два указателя, переданных этой функции, слишком малы — явное указание на проблему. Строка 13 в нашел тестовом файле badqs. с содержит вызов который обнаруживает загубивший вызов и указывает на ошибку.

Отладчик можно использовать также для отображения значений локальных и глобальных переменных, которые могут дать дополнительную информацию об ошибочном месте.

Читайте код перед тем, как исправлять. Один из эффективных, но недооцененных приеморхУгладки — тщательное чтение и обдумывание кода перед внесением в него исправлений. Порою хочется добраться до клавиатуры и начать редактировать программу, чтобы посмотреть, не исчезнет ли ошибка сама собой. Но все же, скорее всего, вы не знаете, что именно сломано, и измените что-нибудь не то, может быть сломав при этом что-нибудь еще. Распечатанный на бумаге критический участок кода выглядит совсем не так, как на экране, и поощряет потратить больше времени на обдумывание. Однако не печатайте листинги постоянно. На распечатку целой программы вы изведете уйму деревьев, а структуру программы, разбросанной по множеству страниц, гораздо сложнее увидеть. Кроме того, распечатка устареет в тот момент, когда вы начнете вносить изменения.

Сделайте перерыв. Иногда вы видите в исходном тексте то, что вы имели в виду, а не то, что вы на самом деле написали. Небольшое отвлечение от текста смягчит ваше недопонимание и поможет коду сказать самому за себя, когда вы к нему вернетесь.

Боритесь с желанием начать исправлять немедленно: подумать — хорошая альтернатива.

Объясните свой код кому-нибудь еще. Другой эффективный способ — объяснить свой код кому-нибудь еще. Такое объяснение часто помогает самому увидеть свою ошибку. Иногда требуется буквально несколько предложений — и звучит смущенная фраза: "Ой, я вижу, где ошибка, извини, что побеспокоил". Это просто замечательный метод, причем в качестве слушателей можно использовать даже непрограммистов.1 В одном университетском компьютерном центре рядом с центром поддержки сидел плюшевый медвежонок. Студенты, встретившиеся с таинственными ошибками, должны были сначала объяснить их этому медвежонку и только затем могли обратиться к консультанту.

Трудные ошибки, нет зацепок

"Не за что зацепиться. Что происходит?" Если у вас действительно нет ни малейшей догадки о том, что же происходит, жизнь становится сложнее.

Сделайте ошибку воспроизводимой. Первый шаг — убедиться, что вы можете заставить ошибку проявляться по вашему желанию. Довольно угнетающе искать ошибку, которая появляется только время от времени. Потратьте время и найдите

такую комбинацию входных данных и настроек, которые гарантированно приводят к ошибке, затем сделайте так, что эту ошибку можно было бы вызвать несколькими нажатиями клавиш. Если ошибка сложна, то при поиске проблемы вам придется повторять ее снова и снова, поэтому, упростив воспроизведение ошибки, вы сэкономите свое время.

Если ошибка появляется от случая к случаю, попытайтесь понять причину этого. Может быть, при каких-то условиях она появляется чаще? Даже если вы не в состоянии повторить ее каждый раз, то, сократив время ее ожидания, вы найдете ее быстрее.

Если программа способна выдавать отладочную информацию, включите ее. Программа случайного моделирования, например программа markov из третьей главы, должна иметь ключ командной строки, выдающий такую отладочную информацию, как, например, стартовое число генератора случайных чисел — это нужно для того, чтобы выдачу программы можно было воспроизвести; другой ключ должен позволять устанавливать это стартовое значение. Многие программы имеют подобные ключи командной строки, неплохо и вам сделать так же.

Разделяй и властвуй. Можно ли уменьшить объем входных данных, приводящих к "падению" программы? Сужайте диапазон возможностей, создавая наименьший набор данных, при котором ошибка все еще проявляется. При каких изменениях ошибка исчезает? Попытайтесь обнаружить важные тестовые случаи, специально фокусирующиеся на ошибке. Каждый тест должен быть нацелен на получение определенного результата, который подтверждает или опровергает какую-нибудь гипотезу о происходящем.

Попробуйте двоичный поиск. Отбросьте половину входных данных и посмотрите, осталась ли ошибка в выходных данных; если нет, то вер-j нитесь к предыдущему состоянию и отбросьте другую половину входных данных. Тот же самый процесс двоичного поиска можно применять и к тексту программы: удалите участок кода, который, по идее, не относится к ошибке, и посмотрите, не исчезла ли она. При сокращении данных для тестирования и больших программ полезен текстовый редактор с возможностью отмены редактирования.

Изучайте нумерологию ошибок. Иногда определенная регулярность чисел, сопровождающих ошибку, подсказывает, на что нужно обратить внимание. Однажды в новой главе этой книги мы обнаружили серию опечаток, заключавшихся в пропадании случайных букв. Ситуация выглядела таинственной. Текст]$ыл создан посредством вырезания и вставки кусков другого файла,лшэтому казалось, что проблемы были в этих самых командах вырезйния и вставки. Откуда начать поиск? Мы взглянули на данные и заметили, что пропавшие символы были равномерно распределены по тексту. При измерении интервалов оказалось, что расстояние между пропавшими буквами было равно 1023 байтам — подозрительно круглое значение. Поиск в исходном тексте редактора нашел несколько кандидатов — чисел в районе 1024. Одно из этих чисел находилось в новом коде, поэтому мы исследовали именно его и немедленно обнаружили классическую "ошибку на единицу", где нулевой байт перезаписывал последний символ в 1024-байтовом буфере.

Изучение структуры чисел, связанных с ошибкой, указало прямо на нее. А затраченное время? Пара минут озадаченности, пять минут рассмотрения данных, чтобы обнаружить закономерность в пропадании символов, минута на поиск вероятных мест ошибки и еще одна минута, чтобы устранить ее. Такую ошибку

совершенно безнадежно было бы искать в отладчике, потому что в ней участвовали две многопроцессных программы, управлявшихся мышью и сообщавшихся друг с другом через файловую систему.

Выводите информацию, локализующую место ошибки. Если вы не понимаете, что именно делает программа, добавьте в нее операторы, отображающие дополнительную информацию, — зачастую это самый простой и недорогой способ выяснения. Например, выведите в каком-нибудь месте кода "сюда нельзя добраться", если вы считаете, что это так; теперь, если вы увидите это сообщение, переместите операторы вывода назад, ближе к началу, чтобы выяснить, в каком месте начинается неправильное поведение программы. Или же отображайте сообщение "добрались сюда", чтобы найти последнюю точку, в которой все еще было хорошо. Сообщения должны отличаться друг от друга, чтобы можно было понять, куда именно вы смотрите.

Отображайте сообщения в компактной фиксированной форме, чтобы их можно было легко просматривать глазами или с помощью программ типа дгер. (Такие программы просто бесценны при поиске текста. В девятой главе приведена простая реализация такой программы.) Если вы отображаете значение переменных, форматируйте их одинаково. В C и C++ показывайте указатели в виде шестнадцатеричных чисел, например %x или %p; это поможет вам увидеть, равны ли два указателя, взаимосвязаны ли они. Научитесь читать значения указателей и распознавать возможные и невозможные значения, например ноль, отрицательные или нечетные числа, а также маленькие числа. Хорошее знакомство с видами адресов поможет также при использовании отладчика.

Если выводимые результаты могут быть очень объемными, то может , быть достаточно отображать лишь одиночные буквы, например A, B, . . . , в качестве компактного отображения потока выполнения программы.

Пишите код, который проверяет сам себя. Если требуется дополнительная информация, напишите собственную функцию, которая проверяет условия, отображает содержимое соответствующих переменных и завершает программу:

```
    /* check: проверить условие, напечатать сообщение */
/* и закончить работу */
  void check(char *s)
 {
if -(varl > var2) {
 printf("%s: varl %d var2 %d\n", s, varl, var2);
 fflush(stdout);
/* для гарантии выполнения вывода */
 abortQ; /* аварийное завершение */
 }
 }
```

Мы сделали так, что check вызывает a bo rt, стандартную функцию библиотеки языка C, которая приводит к аварийному завершению работы программы, чтобы затем можно было проанализировать ее с отладчиком. В каком-нибудь другом случае можно просто продолжить выполнение.

Теперь добавьте вызовы функции check везде, где она может быть полезна:

```
    check("flo подозрительного места");
 /* ... подозрительный код... */
check("после подозрительного места");
```

После исправления ошибки не выбрасывайте функцию check. Оставь-1 те ее в исходном тексте, закомментируйте или запретите с помощью отладочного флага, чтобы ее можно было включить опять, если возникнет другая сложная проблема.

В более запутанных случаях функция check может проводить проверку и отображать структуры данных. Этот подход можно обобщить, используя процедуры, проводящие постоянную проверку целостности структур данных и другой информации. В программе со сложными структурами данных полезно написать такие проверки, поместив их в саму программу до того, как возникнут какие-нибудь проблемы, чтобы их можно было просто включить в случае чего. Используйте их не только для отладки; пусть они будут включ'ены на всех стадиях разработки программы. Если они не сильно влияют на производительность, будет разумно оставить их включенными навсегда. Большие программы типа систем телефонной коммутации часто отводят значительные куски кода "аудитным" подсистемам, которые регулярно анализируют информацию и оборудование, сообщают о встреченных ошибках или даже исправляют их.

Ведите журнальный файл. Другая тактика — ведение журнального файла (log file), содержащего отладочную выдачу фиксированного формата. Когда случается "падение", журнал хранит записи, показывающие, что случилось непосредственно перед этим, web-серверы и другие сетевые программы ведут обширные журналы учета трафика, чтобы собирать информацию о клиентах и о работе программы. Вот такой фрагмент журнального файла можно было встретить на одной из наших машин:

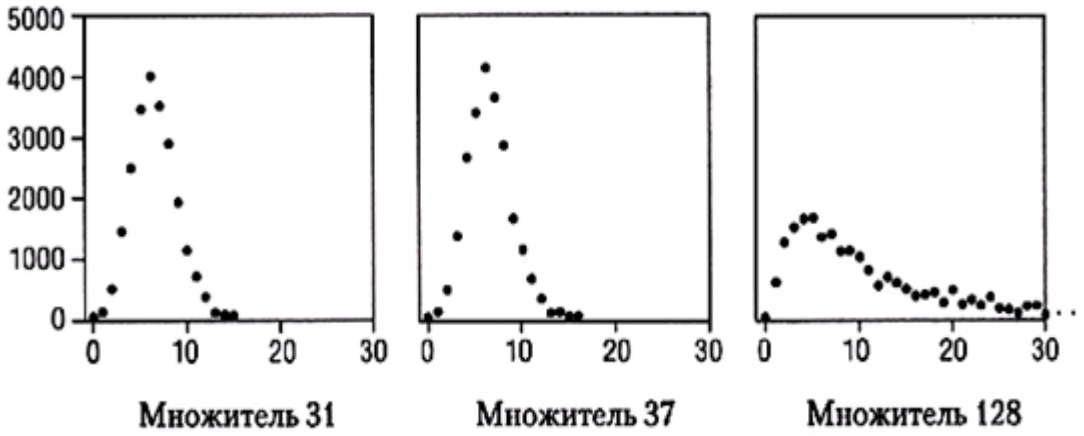
```
    [Sun Dec 27 16:19:24 1998*]
HTTPd: access to
 /usr/local/httpd/cgi-bin/test.html failed for

    m1.cs.bell-labs.com,
reason: client denied by server
 (CGI non-executable) from
 http://m2.cs.bell-labs.com/cgi-bin/test.pi
```

Убедитесь, что вы сбрасываете буферы ввода-вывода, чтобы последние сообщения остались в журнальном файле. Функции вывода типа printf обычно буферизуют выводимые данные, чтобы делать вывод более эффективным; аварийное завершение приведет к потере этих буферизованных данных. В языке С вызов функции fflush гарантирует, что все выводимые данные будут записаны до внезапного завершения программы; в C++ и Java существуют аналогичные функции для выходных потоков. Если вы хотите избежать лишней работы, используйте для журнальных файлов небуферизованный ввод-вывод. Стандартные функции setbuf и setvbuf управляют буферизацией; setbuf (fp, NULL) отключает буферизацию потока fp. Стандартные потоки сообщений об ошибке (stderr, cerr, System, err) обычно небуферизованы.

Постройте график. Иногда, при тестировании и отладке, картинки эффективнее, чем текст. Как мы увидели во второй главе, картинки особенно полезны для понимания структур данных и, конечно же, при написании программ работы с графикой, но они

также могут использоваться для любых программ. Диаграммы разброса данных демонстрируют неверные значения гораздо лучше, чем столбцы чисел. Гистограммы отражают странные места в экзаменационных оценках, случайных числах, размерах "корзин" операторов захвата памяти и хэш-таблиц и т. п.

Если вы не понимаете, что происходит в вашей программе, попробуйте собрать статистику о структуре данных в ней и представить результаты графически. На приводимых графиках изображена статистика цепочек в программе markov из главы 3: по оси x показана длина цепочек, а по оси y — количество элементов в цепочках этой длины. Входные данные — наш стандартный текст, Псалмы (42 685 слов, 22 482 префикса). Первые два графика соответствуют хорошим мультипликаторам, 31 и 37, а третий — кошмарному мультипликатору 128. В первых двух случаях нет ни одной цепочки длиною больше 15 или 16, а большинство элементов хранится в цепочках из 5 или 6 элементов. В третьем случае область распределения много больше, самая длинная цепочка содержит 187 элементов, а в цепочках длиною больше 20 содержатся тысячи элементов.



Используйте различные инструменты. Используйте возможности среды, в которой ведете отладку. Например, программа сравнения файлов, вроде (Jiff, может сравнить результаты успешного и неуспешного запусков, чтобы вы сфокусировали внимание на том, что именно изменилось. Если отладочная выдача очень длинна, используйте g rep для поиска в ней или текстовый редактор для ее исследования. Боритесь с желанием отправить отладочную выдачу на принтер: компьютеры обрабатывают объемистые данные гораздо лучше людей. Используйте языки скриптов и другие средства для автоматизации обработки вывода при отладочных запусках.

Пишите тривиальные программки для проверки гипотез или для подтверждения того, что вы действительно понимаете, как работает та или иная возможность. Например, проверяйте, можно ли освобождать нулевой указатель, такой программой:

```
    int main(void) / '
{
 free(NULL)/ return 0;
}
```

Программы контроля версий исходных текстов типа RCS1 помогут понять, что изменилось, и вернуться к предыдущим версиям, выдающим проверенные результаты. Помимо указания на недавние изменения эти программы могут также

обозначить участки кода, имеющие длинную историю частых модификаций; в таких участках нередко скрываются ошибки.

Ведите записи. Если поиск ошибок продолжается довольно долго, вы можете позабыть, что именно вы пробовали, а что — еще нет. Если вы записываете результаты ваших тестов, то имеете меньше шансов упустить что-нибудь или же посчитать, что вы проверили что-нибудь, тогда как вы этого не сделали. Регистрация поможет вам вспомнить о старой проблеме, когда всплывет что-нибудь аналогичное, а также поможет, если вы захотите объяснить эту проблему кому-нибудь еще.

Последняя надежда

Что делать, если вы все перепробовали, но ничего не помогает? Может быть, как раз наступило время взять хороший отладчик и пройтись по программе. Если ваша мысленная модель работы программы по какой-то причине попросту не соответствует действительности и вы смотрите в совершенно другом направлении, чем нужно, или же смотрите в правильном направлении, но в упор не видите проблему, то отладчик заставит вас изменить ход мыслей. Такие ошибки в "мысленной модели" наиболее сложны, и помощь со стороны машины здесь бесценна.

Иногда источник непонимания очень прост: неверный приоритет операторов, неверный оператор, выравнивание, не соответствующее действительной структуре программы, или же ошибка области видимости, когда локальная переменная прячет под собой глобальную или же глобальная переменная вторгается в локальную область видимости. Например, программисты часто забывают, что & и | имеют меньший приоритет, чем == и ! =. Они пишут так:

```
?if (x & 1 == 0)

? ....
```

и не могут понять - почему результат этого выражения — всегда "ложь". Иногда неверное движение пальца при наборе превращает одиночный символ = в двойной и наоборот:

```
    ? while ((c == getcharO)
 != EOF)
? if (c = An')
? break;
```

Или после редактирования случайно остается лишний код:

```
? for (i=0; i < n; i++);

 ? a[i++] = 0;
```

Или проблему создает спешка при наборе текста кода:

```
    ? switch (c) {
? case '<•:
? mode = LESS;
 ? break;
```

```
? case '>'.:
? mode = GREATER;
? break;
? defualt:
? mode = EQUAL;
 ? break;
 ?
}
```

Иногда ошибка случается из-за неправильного порядка аргументов в вызове процедуры. Если проверка типов не может помочь, например:

```
 memset(p, n, 0); /* записать п нулей в р */?
```

вместо

```
memset(p, 0, п); /* записать n нулей в р */
```

то транслятор такой ошибки не обнаружит.

Иногда незаметно/для вас что-то изменяется, например глобальные или общие переменные, а вы об этом ничего не знаете, пока какая-нибудь функция не обратится к ним.

Иногда в алгоритме или структуре данных есть фатальная ошибка, которую вы просто не замечаете. Во время подготовки материала по цепным спискам мы написали набор функций, создающих новые элементы, вставляющих эти элементы в начало и конец списка, и т. п.; эти функции приведены во второй главе. Конечно же, мы написали тестовые программы, чтобы убедиться, что все правильно. Первые несколько тестов работали, тогда как один эффектно "валился". В сущности, тестовая программа была такой:

```
 ? while (scanf("%s %d","name, Svalue) != EOF) {
? p = newitem(name, value);
 ? listl = addfront(list1, p);
? list2 = addend(list2, p);
 ?
 }
? for (p = listl; p != NULL; p = p->next)
? printf("%s %d\n", p->name, p->value);
```

Как выяснилось, поразительно трудно заметить, что в первом цикле один и тот же узел р добавлялся в два списка сразу, поэтому к тому времени, когда надо было печатать, указатели оказывались безнадежно испорченными.

Такие ошибки искать довольно трудно, потому что мозг просто обходит их. Отладчик помогает здесь, вынуждая вас двигаться в другом направлении, именно в том, в котором движется программа, а не в том, в котором вы думаете. Часто проблема заключается в структуре всей программы, и для того, чтобы увидеть ошибку, требуется вернуться к исходным предпосылкам.

Заметьте, кстати, что в примере со списками ошибка была в тестовом коде, и поэтому ее гораздо сложнее обнаружить. К сожалению, бывает поразительно легко потерять кучу времени, отыскивая несуществующие ошибки, потому что тестовая

программа была ошибочной, или же тестировалась не та версия программы, или перед тестированием не были проведены обновление программы и ее перекомпиляция.

Если вы, приложив массу усилий, не смогли найти ошибку, передохните. Проветрите голову, займитесь чем-нибудь посторонним. Поговорите с приятелем, попросите помощи. Ответ может появиться сам собой, из ниоткуда, но даже если это и не так, вы хотя бы не застрянете в той же самой колее во время следующей отладочной сессии.

Крайне редко проблема действительно заключается в компиляторе, библиотеке, операционной системе или даже в "железе", особенно если что-нибудь изменилось в конфигурации непосредственно перед тем, как появилась ошибка. Никогда нельзя сразу начинать винить все перечисленное, но если все остальные причины устранены, то можно начать думать в этом направлении. Однажды мы переносили большую программу форматирования текста из Unix-среды на PC. Программа отлично ском-пилировалась, но вела себя очень странно: теряла почти каждый второй символ входного текста. Нашей первой мыслью было, что это как-то связано с использованием 16-битовых целых вместо 32-битовых или, может быть, с другим порядком байтов в слове. Печатая символы, полученные во входном потоке, мы наконец нашли ошибку в стандартном заголовочном файле ctype.h, поставлявшемся вместе с компилятором. В этом файле функция isprint была реализована в виде макроса:

```
? «define isprint(c) ((c) >= 040 && (c) < 0177)
```

а в главном цикле было написано так:

```
while (isprint(c = getchar()))
```

Каждый раз, когда входной символ был пробелом (восьмеричное 040, плохой способ записи ' ') или стоял в кодировке еще дальше, а это почти всегда так, функция getchar вызывалась еще раз, потому что макрос вычислял свой аргумент дважды, и первый входной символ пропадал. Исходный код был не столь чистым, как следовало бы, — слишком сложное условие цикла, — но заголовочный файл был непростительно неверен.

Сейчас все еще можно встретиться с такой проблемой: вот этот макрос можно найти в заголовочных файлах одного современного производителя:

```
? Odefine__iscsym(c) (isalrujm(c) |[ ((c) == ' „'))
```

Обильным источником ошибок являются "утечки" памяти, когда забывают освободить неиспользуемую память. Другая проблема —• забывают закрывать файлы до тех пор, пока не переполнится таблица открытых файлов и программа больше не сможет открыть ни одного. Программы с утечками таинственным образом отказываются работать, потому что у них заканчивается тот или иной ресурс, но конкретную ошибку бывает невозможно воспроизвести.

Иногда отказывает само "железо". Ошибка в вычислениях с плавающей точкой в процессоре Pentium в 1994 году, которая приводила к неверным ответам при некоторых'вычислениях, была обширно освещена в печати и довольно дорого обошлась. После того как она была обнаружена, ее, конечно же, мржно было легко воспроизвести. Одна из самых странных ошибок, которую мы когда-либо видели,

содержалась в программе-калькуляторе, некогда работавшем на двухпроцессорной машине. Иногда выражение 1/2 выдавало результат 0. 5, а иногда — постоянно появляющееся, но совершенно неправильное значение 0.7432; никаких закономерностей в появлении правильного или неправильного значений не было. В конце концов проблему обнаружили в модуле вычислений с плавающей точкой в одном из процессоров. Программа-калькулятор случайным образом выполнялась то на одном из них, то на другом, и в зависимости от этого ответы были либо верными, либо совершенно бессмысленными.

Много лет назад мы использовали машину, температуру которой можно было оценить, исходя из количества младших битов, бывших неправильными при вычислениях с плавающей точкой. Одна из ее плат была плохо закреплена, и, когда машина нагревалась, эта плата сильнее выдвигалась из своего разъема и больше битов данных отключалось от основной платы.

Невоспроизводимые ошибки

С нестабильными ошибками сложнее всего иметь дело, и обычно проблема не столь очевидна, как неисправное "железо". Однако сам факт, что проблема недетерминирована, содержит в себе информацию. Это означает, что ошибка, скорее всего, не в вашем алгоритме, а в том, как ваш код использует информацию, которая изменяется при каждом выполнении программы.

Проверьте, что все переменные инициализированы. Может быть, вы просто используете случайное значение, оставшееся в повторно используемой ячейке памяти. При написании программ на C и C++ в этом чаще всего виновны локальные переменные функций и выделяемая память. Установите все переменные в заранее известное значение; напрмер, если стартовое значение генератора случайных чисел обычно вычисляется исходя из времени суток, то присвойте ему нулевое значение.

Если ошибка изменяет свое поведение или вообще исчезает при добавлении отладочного кода, то это может быть связано с выделением памяти: где-то вы пишете за пределы выделенной памяти, а добавление отладочного кода изменяет расположение данных в памяти, так что ошибка начинает проявляться по-другому. Большинство функций вывода, от printf до диалоговых окон, захватывают для себя память сами, еще больше "мутя воду".

Если место ошибки, казалось бы, находится далеко от любого места, в котором она могла бы появиться, значит, происходит запись за пределы доступной памяти, причем затирается значение, которое используется лишь гораздо позже. Иногда случается проблема "висящих указателей", когда указатель на локальную переменную случайно передается за пределы функции, а затем используется. Возврат адреса локальной переменной — лучший способ создания ошибки замедленного действия:

```
    ? char *msg(int n, char *s)
?
 {
 ? char buf[100];
?
 ? sprintf(buf, "error %d: %s\n", n, s);
? return but;
 ?
}
```

К тому моменту, когда указатель, возвращаемый функцией msg, используется, он уже не указывает на осмысленное место. Память нужно выделять с помощью функции malloc, использовать массив, объявленный как static, или требовать предоставления памяти вызывающей программой.

Использование динамически выделяемого значения после того, как оно было освобождено, имеет подобные симптомы. Мы уже упоминали об этом во второй главе, когда написали функцию f reeall. Вот этот код — неверен:

```
?for (p = listp; p != NULL; p = p->next)

? free(p);
```

После того как память была освобождена, она не должна использоваться, потому что ее содержимое могло измениться и нет гарантии, что p->next все еще указывает на правильное значение.

В некоторых реализациях malloc и free повторное освобождение участка памяти портит внутренние структуры, но не вызывает никаких проблем до тех пор, пока гораздо позже какая-нибудь другая операция выделения памяти не поскользнется на испорченном участке памяти. Некоторые реализации динамического выделения памяти имеют отладочные возможности для проверки корректности области динамической памяти при каждом вызове. Включите такую отладку, если вы столкнулись с недетерминированной ошибкой. В крайнем случае вы можете написать собственную реализацию динамической памяти, которая проверяет каждую операцию или просто заносит эти операции в журнал для дальнейшего изучения. Если ситуация удручает, достаточно написать простую, не очень скоростную реализацию. Есть также великолепные коммерческие продукты, проверяющие работу с памятью и отлавливающие ошибки и утечки; если у вас таких нет, собственная версия malloc и f гее может в какой-то мере их заменить.

Когда программа работает у одного пользователя и не работает у другого, значит, дело во внешней среде, в которой выполняется программа. Несоответствие может оказаться в файлах, которые читает программа, в правах доступа к файлам, в переменных окружения, в путях поиска команд, в значениях по умолчанию и в стартовых файлах. Сложно сказать что-либо в такой ситуации, потому что вам придется постараться полностью сдублировать среду выполнения неверной программы, практически вжиться в шкуру ее пользователя.

Упражнение 5-1

Напишите версии malloc и free, которыми можно пользоваться для отладки проблем с выделением памяти. Одним из возможных подходов является проверка всего используемого пространства при каждом вызове malloc и free; другой подход — записывать журнальную информацию, которую можно обрабатывать специальной программой. В любом случае добавьте в начало и конец выделяемой памяти специальные маркеры, чтобы отследить запись, выходящую за ее пределы.

Средства отладки

Отладчики — не единственные средства нахождения ошибок. Самые различные программы помогают нам обрабатывать объемистый вывод для того, чтобы отыскивать интересующие участки, находить аномалии и представлять выходные

данные в наиболее простой и понятной форме. Многие из таких программ входят в стандартный набор утилит, другие пишутся специально, чтобы обнаружить конкретную ошибку или проанализировать определенную программу.

В этой главе мы опишем простую программу strings, очень полезную для просмотра файлов, состоящих в основном из непечатаемых символов, например исполняемых файлов или таинственных двоичных форматов, столь любимых некоторыми текстовыми процессорами. В таких файлах часто спрятана полезная информация, например текст документа, сообщения об ошибках, недокументированные опции программы, имена файлов и каталогов или имена функций, которые могут вызываться программой.

Программа st rings полезна и для нахождения текста в других двоичных файлах. Файлы с изображениями часто содержат ASCII-строки, сообщающие, какая программа создала этот файл, а сжатые файлы и архивы (например, zip-файлы) могут содержать имена файлов: strings обнаружит и их.

Unix-системы обычно уже содержат реализацию программы strings, хоть она и отличается от той, которую запрограммируем мы. Unix-версия в случае, если обрабатываемый файл — программа, просматривает только сегменты кода и данных, игнорируя таблицу символов. Ключ -a заставляет ее читать весь файл.

В сущности, strings извлекает ASCII-строку из двоичного файла, чтобы ее можно было прочитать или обработать с помощью другой программы. Если в тексте сообщения об ошибке не говорится, какая именно программа выдала данное сообщение, то узнать это, не говоря уж о том, почему именно она его выдала, будет довольно сложно. В этом случае установить источник можно поиском в подозрительных каталогах; этот поиск выполняется с помощью такой команды:

```
% strings *.exe *.dll | grep 'mystery message'
```

Функция st rings читает файл и печатает каждую последовательность из как минимум MINLEN = 6 печатных символов.

```
    /* strings: извлечь из потока читабельные строки */
void strings(char *name, FILE *fin)
 {
 int c, i;
char buf[BUFSIZ];
 do { /* один раз для каждой строки */
 for (i =0; (c = getc(fin)) != EOF; )
{
if (! isprint(c))
 break;
buf[i++] = c; if (i >= BUFSIZ)
break;
}
 if (i >= MINLEN) /*
 если строка слишком длинная */
printf("%s:%.*s\n", name, i, but); } while (c != EOF);
 }
```

Форматная строка %. *s в функции printf берет длину строки из следующего аргумента (i), потому что buf не завершается нулем.

Цикл do-whi|le находит и печатает каждую строку, заканчивая работу при обнаружении EOF. Проверка конца файла после тела цикла позволяет функции getc и циклу по строке иметь одинаковое условие завершения, а также c\ помощью единственного обращения к printf обрабатывать конец строки, конец файла и слишком длинные строки.

Стандартный внешний цикл с проверкой при входе или единственный цикл с getc и более сложным телом заставил бы использовать printf дважды. Эта функция сначала так и работала, но потом мы нашли ошибку в операторе printf. Исправив в одном месте, мы забыли исправить ее в двух других. ("А не делал ли я ту же самую ошибку где-нибудь еще?") Здесь нам стало ясно, что программу нужно переписать, чтобы дублирующегося кода было меньше; так появился цикл do-while.

Основная процедура программы strings вызывает функцию strings для каждого файла- аргумента:

```
    /* strings main: искать в файлах читабельные строки */
int main(int argc, char *argv[])
 {
 int i;
FILE *fin;
 set progname(" strings");
if (argc == 1)
 eprintf("использование: strings имена_файлов")';
 else
{
for (i = 1; i < argc; i++)
 {
 if ((fin = fopen(argv[i], "rb")) == NULL)
 weprintfC'ne могу открыть %s:", argv[i]);
 else
 {
 strings(argv[i], fin); fclose(fin);
} } }
 return 0;
  }
```

Вы, наверное, удивлены, что strings не читает стандартный ввод, если не было дано ни одного имени файла. Сначала именно так и было. Для того чтобы объяснить, почему теперь это изменилось, требуется рассказать историю об отладке.

Очевидный тест программы st rings — пропустить ее через саму себя. Это сработало отлично под Unix, но под Windows 95 команда

```
C:\> strings <strings.exe
```

выдала ровно пять строк:

```
   IThis program cannot be run in DOS mode.
  '. rdata
@.data
```

.idata
.reloc

Первая строка "Эта программа не может исполняться под DOS" выглядела как сообщение об ошибке, и мы потеряли некоторое время, пока не I поняли, что это на самом деле строка из файла с программой, так что результат был правилен, по крайней мере до какого-то момента. Не секрет, что некоторые отладочные сессии терпели крушение из-за неверного понимания источника сообщения.

Но в любом случае должны быть еще строки! Где они? Однажды поздно ночью наконец забрезжил свет. ("Я где-то уже видел это!") Это — проблема с переносимостью, описанная подробнее в восьмой главе. Изначально мы написали программу так, чтобы она читала только из стандартного ввода, используя функцию getchar. Под Windows, однако, getchar возвращает EOF, когда она встречает определенный байт (0x1 A или Control-Z) в текстовом режиме ввода,4 и именно это и приводило к преждевременному завершению.

Это абсолютно законное поведение, но совсем не то, что ожидали мы, с нашим опытом работы с Unix. Было решено открывать файл в двоичном режиме, используя "rb". Но stdin уже открыт, а стандартного способа изменить режим его работы не существует. (Можно использовать функции fdopen или setmode, но они не являются частью стандарта.) Таким образом, мы столкнулись с набором неприятных альтернатив: заставить пользователя всегда задавать имя файла, чтобы программа работала под Windows за счет неудобства для пользователей Unix; без пред-] упреждения выдавать неправильный ответ, если пользователь Windows пытается задействовать стандартный ввод; использовать условную компиляцию, чтобы адаптировать поведение к различным системам ценой пониженной переносимости. Мы выбрали первый вариант, чтобы программа везде работала одинаково.

Упражнение 5-2

Программа strings печатает строки длиной MINLEN или более символов, и иногда при этом обнаруживается гораздо больше строк, чем надо. Реализуйте необязательный аргумент, устанавливающий минимальную длину строки.

Упражнение 5-3

Напишите программу vis, которая копирует стандартный ввод на стандартный вывод, отображая непечатаемые символы типа "забоя", контрольных символов и не-АЗСП-символов в виде\Хпп, где hh — шест-надцатеричное представление непечатаемого байта. В отличие от st ri ngs программа vis полезна при обработке файлов, содержащих лишь несколько непечатаемых символов.

Упражнение 5-4

Что выдает vis, если во входном потоке попадается строка \XOA? Можете ли вы устранить двусмысленность результатов работы этой программы?

Упражнение 5-5

Расширьте функциональность программы vis, чтобы она могла обрабатывать набор файлов, разбивать слишком длинные строки на части и полностью удалять

непечатаемые символы. Какие еще возможности, хорошо совместимые с назначением этой программы, можно реализовать?

Чужие ошибки

По правде говоря, большинству программистов не достается удовольствие разработки совершенно новой системы с нуля. Вместо этого большую часть времени они проводят, используя, поддерживая, изменяя и неизбежно отлаживая код, написанный другими людьми.

При отладке чужого кода остается в силе все, что мы сказали об отладке своего собственного кода. Перед тем как начать, однако, вы сначала должны добиться определенного понимания организации программы и хода мысли ее создателей. Термин "открытие", использованный в одном очень большом программном проекте, является не такой уж плохой метафорой. Задачей является "открыть", что происходит в коде, который писали не вы.

Здесь очень сильно могут помочь инструментальные средства. Программы текстового поиска типа g rep помогут найти все места использования какого-нибудь имени. Перекрестные ссылки дают определенное представление о структуре программы. Граф, показывающий взаимные вызовы функций, ценен, если только он не очень велик. Пошаговый проход по программе с помощью отладчика поможет увидеть последовательность событий. Из истории ревизий программы можно узнать, что происходило в ней с течением времени. Частые изменения являются знаком того, что код был плохо понят или подвергался частой смене требований и поэтому потенциально содержит ошибки.

Иногда вам нужно найти ошибку в программе, которой вы не писали и исходного текста которой вы даже не имеете. В этом случае задачей является обнаружение и описание ошибки, причем так, чтобы вы смогли аккуратно сообщить о ней разработчику и в то же время, возможно, найти ее обходное решение.

Если вам кажется, что вы нашли ошибку в чужой программе, первым шагом следует убедиться, что это настоящая ошибка, чтобы не терять ни времени автора, ни собственного авторитета.

Если вы нашли ошибку в компиляторе, убедитесь, что это действительно ошибка компилятора, а не ошибка в вашем коде. Например, операция побитового сдвига вправо заполняет освобождающиеся биты нулем (логический сдвиг) или знаковым битом (арифметический сдвиг), а чем именно — в языках С и С++ не указано, поэтому новички иногда считают, что если конструкция типа

```
? i = -1; printfO

 ? i » 1);
```

выдает неожиданный результат, то ошибка — в компиляторе. На самом деле это — вопрос переносимости, потому что данный оператор имеет право действовать по-разному на разных системах. Попробуйте проверить различные системы и посмотрите, что произойдет; обратитесь к описанию языка, чтобы удостовериться в правильной интерпретации результатов.

Убедитесь, что ошибка не нова. Используете ли вы последнюю версию программы. Есть ли список исправленных в ней ошибок? Большая часть программного

обеспечения проходит серию выпусков; если вы нашли ошибку в версии 4.0Ы, она может быть уже исправлена или заменена новой в версии 4.0Ь2. В любом случае немногие программисты испытывают достаточно энтузиазма, чтобы исправлять ошибки где-либо, кроме текущей версии программы.

Наконец, представьте самого себя в роли получателя вашего сообщения об ошибке. Вам следует предоставить автору максимально удобный тестовый пример, какой только сможете сделать. Будет не слишком хорошо, если ошибку можно показать только при больших объемах входных данных, или в очень сложных условиях, или при наличии множества дополнительных файлов. Сократите тест до минимального самодостаточного случая. Сообщите также дополнительную информацию, которая, возможно, связана с ошибкой, например версию самой программы, версию компилятора, операционной системы, состав оборудования на машине. Для ошибочной версии isprint, упомянутой в параграфе 5.4, мы предоставили такой тестовый случай:

```
    /* тестовая программа для ошибки в isprint */
  int main(void)
{
 int c;
 while (isprint(c = getchar()) || c != EOF)
 printf("%c", c); return 0; }
```

Входными данными служила любая строка печатного текста, потому что на выходе появлялась только половина входа:

```
    % echo 1234567890 isprint_jtest   .
 24680
 %
```

Лучшие сообщения об ошибке — те, что требуют для демонстрации ошибки только пары строк входных данных на стандартной конфигурации, а также те, что содержат и исправление соответствующей ошибки. Шлите такие сообщения об ошибке, какие вам хотелось бы получать самим.

Заключение

При правильном подходе отладка может превратиться в удовольствие типа разгадывания головоломки. Впрочем, неважно, нравится нам это или нет, отладка является искусством, которое нам придется демонстрировать регулярно. Было бы неплохо, если бы ошибок не возникало вовсе, поэтому мы с самого начала пытаемся избежать их, стараясь писать правильный код. Хорошо написанный код сразу содержит меньше ошибок, а те, что все же имеются, гораздо легче обнаружить.

После того как вы увидите ошибку, первое, что нужно сделать, — понять, на что "намекает" эта ошибка. Откуда она могла взяться? Есть ли в ней что-нибудь знакомое? Не менялось ли что-нибудь в программе буквально только что? Есть ли какие-нибудь особенности у входных ? данных, которые привели к ошибке? Нескольких хорошо отобранных тестовых случаев и нескольких операторов печати в коде может быть достаточно.

Если четких намеков нет, все равно, хорошо подумать — лучший пер- і вый шаг, за которым должны следовать систематические попытки локализовать местонахождение проблемы. Одним из возможных шагов будет сокращение входных

данных до минимальных размеров, при которых программа все еще отказывается работать. Другой возможный шаг — удаление кода, чтобы устранить те его участки, что не связаны с проблемой. Можно добавить проверяющий код, который включается только через определенное количество шагов в программе, чтобы опять же попытаться локализовать проблему. Все эти шаги делаются в рамках одной стратегии "разделяй и властвуй", при отладке эффективной столь же, сколь в политике и войне.

Используйте другие вспомогательные средства. Объясните свой код кому-нибудь еще (хотя бы плюшевому медведю) — это восхитительно эффективно. Используйте отладчик, чтобы увидеть стек вызовов. Используйте коммерческие средства обнаружения утечек памяти, нарушения границ массивов, подозрительного кода и т. п. Пройдитесь по программе, если станет ясно, что вы не очень понимаете, как она работает.

Познайте себя и то, какие ошибки вы совершаете. После того как вы нашли и обнаружили ошибку, убедитесь, что вы устранили и другие подобные ошибки. Подумайте о происшедшем, чтобы избежать повторения той же самой ошибки.

Дополнительная литература

Много полезных советов по отладке содержится в книгах Стива Ма-гьюира "Создание надежного,кода" (Steve Maguire. Writing Solid Code. Microsoft Press, 1993) и Стива Мак-Коннелла "Все о коде" (Steve McConnell. Code Complete. Microsoft Press, 1993).

Тестирование

- Тестируйте при написании кода
- Систематическое тестирование
- Автоматизация тестирования
- Тестовые оснастки
- Стрессовое тестирование
- Полезные советы
- Кто осуществляет тестирование?
- Тестирование программы markov
- Заключение
- Дополнительная литература

> В практике вычислений вручную или с помощью настольной машины, надо, взять за правило проверять каждый шаг вычисления и, при нахождении ошибки, локализовать ее, повторив процесс в обратном порядке с той точки, где ошибка была обнаружена впервые.
>
> Норберт Винер. Кибернетика

Тестирование и отладка часто упоминаются вместе, однако это две разные вещи. Сильно упрощая, можно сказать, что отладкой называется то, что вы делаете, когда знаете, что программа не работает. Тестирование же — это последовательные, систематические попытки добиться ошибки от программы, которая считается работающей.

Эдсгеру Дейкстре (Edsger Dijkstra) принадлежит известное высказывание о том, что тестирование может показать лишь наличие ошибок, но не их отсутствие. Он надеется на то, что создатели программ смогут писать их корректно, то есть без ошибок вообще, и, следовательно, в тестировании не будет никакой необходимости. Это, конечно, отличная цель, и к ее достижению стоит стремиться, но для настоящих (коммерческих) программ это пока нереально. Так что в данной главе мы остановимся на том, как тестировать программы с целью находить ошибки быстро, рационально и эффективно.

Задумываться о потенциальных проблемах вашего кода полезно всегда. Систематическое тестирование, от простейших до самых хитроумных тестов, позволит удостовериться в том, что программа является корректной с самого начала и остается таковой по мере усовершенствования. Автоматизация позволяет во многом избежать нудного ручного тестирования, заменив его экстенсивным автоматическим тестированием. Существует великое множество приемов и ухищрений, которым опыт научил программистов.

Один из способов написания кода, не содержащего ошибок, — генерировать его программно. Если некоторое задание на программирование понятно настолько, что работа по написанию кода кажется механической, ее следует механизировать. Так бывает, когда программу можно сгенерировать из спецификации, написанной на специализированном языке. Например, мы компилируем код на языке высокого уровня в ассемблерный код, используем регулярные выражения для задания

шаблонов текста, используем нотации типа SUM(A1: A50) для представления операций в некотором диапазоне ячеек электронной таблицы. В подобных случаях при наличии корректного генератора или транслятора и корректной спецификации результирующая программа будет также корректна. Более детально эту обширную тему мы обсудим в главе 9, в этой же главе мы в общих чертах осветим способы создания тестов из компактных спецификаций.

Тестируйте при написании кода

Чем раньше обнаружена проблема, тем лучше. Если вы будете постоянно задумываться о том, что вы пишете, еще когда вы пишете, то сможете проконтролировать простейшие свойства программы прямо на этапе ее создания. В результате ваш код как бы пройдет один круг тестирования еще до того, как будет скомпилирован. Ошибки определенных видов даже никогда не появятся.

Тестируйте граничные условия кода. Одним из важнейших методов тестирования является тестирование граничных условий: каждый раз, написав небольшой кусок кода, например цикл или условное выражение, проверьте, что тело цикла повторится нужное количество раз, а условное выражение правильно разветвляет вычисление. Этот процесс называется тестированием граничных условий потому, что вы проверяете крайние, экстремальные значения алгоритма или данных, такие как пустой ввод, единственный введенный элемент, полностью заполненный массив и т. п. Основная идея состоит в том, что большинство ошибок возникает как раз на границах — при каких-то экстремальных значениях. Если какой-то блок кода содержит ошибку, то, скорее всего, эта ошибка происходит на границе, и наоборот — если при экстремальных значениях код работает корректно, то он практически наверняка будет работать корректно и повсюду.

Приводимый фрагмент кода, моделирующий f gets, считывает симво-'j лы, пока не найдет символ перевода строки или не заполнит буфер:

```
    ? int i;
 ? char s[MAX];
?
 for (i = 0; (s[i] = getchar()
1= '\n' && i < MAX-1; ++i)
 9
s[--i] = '\01'
```

Представьте себе, что вы только что написали этот цикл. Теперь мысленно выполните за него обработку строки. Первое граничное условие, которое надо проверить, очевидно: пустая строка. Если представить строку, содержащую единственный символ перевода строки, то нетрудно убедиться, что цикл остановится на пертой итерации со значением i, равным 0, так что в последней строке i будет уменьшено до -1 и, следовательно, запишет нулевой байт в элемент s[-1], который находится вне границ массива. Итак, проверка первого же граничного условия обнаружила ошибку.

Если переписать цикл так, чтобы он использовал идиоматическую форму заполнения массива вводимыми символами, он будет выглядеть следующим образом:

```
    ? for (1=0; i < MAX-1; i++)
? if ((s[i] = getchar()) == '\n')
```

```
 ? break;
 ? s[i] = '\O';
```

Повторив в уме первый reef, мы удостоверимся, что теперь строка, содержащая только символ перевода строки, обрабатывается корректно: i равно 0, первый же введенный символ прерывает работу цикла, а '\0- сохраняется в s[0]. Проверив схожим образом варианты с вводом одного и двух символов, замыкаемых символом перевода строки, мы убедимся, что цикл работает корректно вблизи нижней границы ввода.

Теперь надо проверить и другие граничные условия. Ситуации, когда во вводе содержится очень длинная строка или не содержится символов перевода строки, предусмотрены кодом — на этот случай существует ограничение i значением MAX-1. Однако что будет, если ввод абсолютно пуст (в нем нет вообще ни одного символа) и первый же вызов getchar возвратит значение EOF? Надо добавить проверку и для такого случая:

```
    for (i = 0; i < MAX-1; i++)
if ((s[i] = getchar()) == '\n' | s[i] == EOF)
 break; s[i] = '\O';
```

Проверка граничных условий может обнаружить много ошибок, но, конечно, не все. Мы еще вернемся к рассмотренному примеру в главе 8, где покажем, что в нем осталась еще ошибка переносимости.

Следующим шагом будет проверка ввода около другой границы, когда массив почти заполнен, полностью заполнен и наконец переполнен, особенно если как раз в этот момент и встречается символ перевода строки. Мы не будем расписывать здесь все детали этих тестов; выполните их самостоятельно, — это очень хорошее упражнение. Задумавшись о всевозможных граничных условиях, нам придется решить, что делать в случае, если буфер заполнится до того, как во вводе встретится ' \n '; этот пробел в спецификации должен быть ликвидирован на ранней стадии написания программы.

Тестирование граничных условий особенно эффективно для поиска ошибок выхода за границы массива на 1 (off-by-one errors). Попрактиковавшись, вы сделаете такую проверку своей второй натурой, и множество тривиальных ошибок будет устранено в самый момент возникновения.

Тестируйте пред- и постусловия. Еще один способ предотвратить возникновение проблем — удостовериться в том, что ожидаемые или необходимые условия удовлетворяются до (предусловие) или после (постусловие) выполнения некоторого блока кода. Проверяя вводимые значения на соответствие допустимому диапазону, мы встретились как раз с примером тестирования предусловий. Ниже приведена функция для вычисления среднего из п элементов массива. При значениях л, меньших или равных 0, функция работает некорректно:

```
    ? double avg(double a[], int n)
 ? { /* average - среднее */
 ? int i;
  ? double sum;
  ?
  ? sum = 0.0;
 ? for (i =0; i < n; i++)
```

```
? . sum += a[i];
? return sum / n;
 ?

    }
```

Что будет делать эта функция, если n будет равно 0? Массив, не содержащий элементов, — вполне осмысленный элемент программы, а вот среднее значение его элементов не имеет никакого смысла. Должна ли функция позволять системе отлавливать деление на 0? Прерывать функцию? Сообщать об ошибке? Без предупреждения возвращать какое-нибудь нейтральное значение? А что если n вообще отрицательно, что абсолютно бессмысленно, но не невозможно? Как мы уже говорили в главе 4, нам представляется правильным в случае, если n меньше либо равно нулю, возвращать 0 в качестве среднего значения:

```
return n <= 0 ? 0.0
```

но однозначно правильного ответа здесь не*существует.

Имеется, правда, гарантированно неверное мнение — игнорировать проблему. В ноябре 1998 в журнале Scientific American был описан инцидент, произошедший на борту американского ракетного крейсера Yorktown. Член команды по ошибке вместо значимого числа ввел 0, что привело к ошибке деления на нуль; ошибка разрослась и в конце концов силовая установка корабля оказалась выведена из строя. Несколько часов Yorktown дрейфовал по воле волн — а все из-за того, что в программе не была осуществлена проверка диапазона вводимых значений.

Используйте утверждения. В C и C++ существует возможность использования специального щ механизма утверждений (assertions) (в <assert. h>), который позволяет включать в программу проверку пред- и постусловий. Поскольку невыполненное утверждение прерывает работу программы, используют их, как правило, в ситуациях, когда сбой на самом деле не ожидается, а при его возникновении нет возможности продолжить работу нормально. Только что рассмотренный пример можно было бы дополнить утверждением, вставленным перед началом цикла:

```
assert(n > 0);
```

Если утверждение не выполнено, программа прерывается; сопровождается это выдачей стандартного сообщения:

```
Assertion failed: n > O,
 file avgtest.c, line 7
Abort(crash)
```

Утверждения особенно полезны при проверке свойств интерфейса, поскольку они привлекают внимание к несовместимости вызывающего и вызываемого блоков системы и зачастую помогают найти виновника этой несовместимости. Так, если наше утверждение, что n больше O, не проходит при вызове функции, это сразу указывает на ошибку в вызывающей функции, а не в самой функции avg. Если интерфейс изменился, а внести соответствующие коррективы мы забыли, утверждения помогут выловить ошибку до того, как она приведет к возникновению серьезных проблем.

Используйте подход защитного программирования. Полезно вставлять некоторый код для обработки (хотя бы просто предупреждения пользователю) случаев, которых "не может быть никогда", то есть ситуаций, которые теоретически не должны случиться, но все же имеют место (например, из-за сбоя где-то в другом участке программы). Хороший пример — добавление проверки на нулевой или отрицательный размер массива в функцию avg. Еще одним примером может стать программа, выставляющая оценки по американской системе; очевидно, что отрицательных или очень больших значений появиться в ней не может, но лучше все же это проверить:

```
    if (grade < 0 || grade > 100) /* этого не может быть */
 letter = '?'; else if (grade >= 90)
  letter = 'A'; else
  ...
```

Это пример защитного программирования (defensive programming), при котором вы убеждаетесь в том, что программа защищена от неправильного использования или некорректных данных. Пустые указатели, индексы вне диапазона, деление на ноль и другие ошибки можно обнаружить на ранних стадиях жизни программы или нейтрализовать. Если бы все программисты применяли принципы защитного программирования, с Yorktown ничего бы не произошло, что бы там ни вводил оператор.

Проверяйте коды возврата функций. Одним из приемов защиты, которым программисты почему-то незаслуженно пренебрегают, является проверка возвращаемого значения библиотечных функций и системных вызовов. Значения, возвращаемые функциями, обслуживающими ввод, такими как f read и fscant, надо всегда проверять. Также обязательно надо проверять и возвращаемые значения вызовов открытий файлов типа f open. Если чтение или открытие файла по каким-то причи- і нам не выполняется, не может быть и речи о нормальном продолжении работы программы.

Проверка возвращаемого значения функций вывода типа f p rintf или fwrite поможет поймать ошибки, происходящие при попытке записи в файл, когда свободного места на диске не осталось. Также полезно на всякий случай проверить значение, возвращаемое fclose, — если при выполнении произошла какая-нибудь ошибка, эта функция возвратит EOF, в противном случае возвращается ноль.

```
    fp = fopen(6utfile, "w");
  while (...) /* вывод в outfile */
 fprintf(fp, ...); if (fclose(fp) == EOF)
   { /* нет ли ошибок? */
  /* произошла какая-то ошибка вывода */ }
  }
```

Последствия ошибок вывода могут быть серьезными. Если записываемый файл является обновленной версией уже существующего файла, такая проверка спасет вас от удаления старой версии при отсутствии сформировавшейся новой.

Усилия, потраченные на тестирование кода при написании, минимальны и окупаются сторицей. Мысли о тестировании в момент написания улучшают код, так как в, этот момент вы яснее всего отдаете себе отчет в том, что он должен делать. Если же вы начинаете проверку только после того, как что-нибудь сломается, то к этому моменту вы можете забыть, как код работает. Работая под давлением, вам будет трудно

восстановить всю картину заново, что отнимет много времени, а внесенные исправления окажутся менее продуманными и, следовательно, менее эффективными, поскольку ваше понимание, скорее всего, окажется неполным.

Упражнение 6-1

Проверьте приводимые фрагменты на граничные условия и при необходимости исправьте их, руководствуясь принципами хорошего стиля, изложенными в главе 1, и советами из этой главы.

1. Этот код должен вычислять факториалы:

```
    ? int factorial(int n)
 ? {
 ? int fac;
 ? fac = 1;
? while (n--)
 ? fac *= n;
 ? . return fac;
 ?
```

2. Этот отрывок должен распечатывать символы строки, каждый в отдельной строке:

```
   ? 1 = 0;
? do {
? putchar(s[i++]);
? putchar('\n');
? } while (s[i] != \'O');
```

3. Предполагается, что эта функция будет копировать строку из одного места в другое (из источника s гс в приемник dest):

```
   ? void strcpy(char
 *dest, char *src)
 ? {
 ? int i;
?
? for (i = 0; src[i] != ДО'; i++)
 ? dest[i] = src[i];
 ?
 }
```

4. Еще один пример копирования строк — на этот раз копируется n символов из s в t:

```
    ? void strncpy(char *t, char *s, int n)
? {
 ? while (n > 0 && *s != '\0') {
? *t = *s;
? t++;
 ? s++;
? n--;
 ? }
? }
```

5. Сравнение чисел:

```
    ? If (1 > j)
? printf("%d больше %d.\n", i, j);
 ? else
? printf("%d меньше %d.\n", i, j);
```

6. Проверка класса символа:

```
     ? if (c >= 'A' && c <= :Z;) {
 ? if (c <= 'L')
 ? cout « " первая половина алфавита ";
? else
 ? cout « " вторая половина^лфавита ";
? }
  }
```

Упражнение 6-2

Мы пишем эту книгу в конце 1998 года, поэтому призрак проблемы 2000 года неотступно стоит перед нами как самая глобальная ошибка граничных условий.

1. Какие даты вы используете для поверки программы на работоспособность в 2000 году? Предположим, что выполнять тесты очень дорого, в каком порядке вы будете их осуществлять после ввода даты 1 января 2000 года?

2. Как вы будете тестировать стандартную функцию ctime, которая возвращает строковое представление даты в такой форме:

```
Fri Dec 31 23:58:27 EST 1999\n\0
```

Предположим, что вы в своей программе вызываете ctime. Как вы будете предохранять свой код от некорректной реализации этой функции?

3. Опишите, как вы будете тестировать программу-календарь, которая генерирует вывод в таком виде:

```
January 2000 S M Tu W Th F S 1
2345678
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

4. Какие еще граничные условия в отношении времени и дат существуют в используемой вами системе? Как бы вы оттестировали их?

Систематическое тестирование

Очень важно тестировать программу систематически — на каждом этапе надо четко представлять, что вы тестируете в данный момент и каких результатов ожидаете.

Тестирование должно производиться последовательно, чтобы ничего не упустить: текущие результаты тестирования надо обязательно записывать, чтобы представлять, что уже сделано и что предстоит сделать.

Тестируйте по возрастающей. Тестирование должно идти рука об руку с созданием кода. Тестирование методом "большого скачка", когда сначала пишется вся программа, а потом тестируется целиком, гораздо сложнее и отнимает гораздо больше времени, чем постепенное. Напишите часть программы, оттестируйте ее, напишите очередной кусок кода, оттестируйте его и т. д. Если у вас есть два блока, которые писались и тестировались раздельно, оттестируйте их взаимодействие.

Например, когда мы тестировали программу CSV из главы 4, на первом шаге было достаточно написать только код, читающий ввод, и отладить его. На следующем шаге мы разделяли вводимые строки запятыми. Добившись работоспособности этих кусков, мы перешли к полям с кавычками и так мало-помалу подошли к тестированию всего вместе.

Тестируйте сначала простые блоки. Общий подход к тестированию по возрастающей применим и к отдельным деталям программы. В первую очередь тестированию подлежат самые простые и чаще всего исполняющиеся блоки; только после того, как вы удостоверитесь в их корректности, можно двигаться дальше. Таким образом, на каждом этапе вы увеличиваете объем тестируемого кода, будучи при этом уверенными в работоспособности основных его частей. Простые тесты обнаруживают простые ошибки. Каждый тест выполняет свой минимум по поиску новой потенциальной проблемы. И несмотря на то что каждую новую ошибку выловить труднее, чем предыдущую, вовсе не факт, что ее будет труднее и исправить.

В этом параграфе мы поговорим о путях выбора эффективных тестов и о порядке их применения, а в двух следующих параграфах обсудим способы механизации процесса для наиболее эффективного тестирования.

Первый шаг, по крайней мере для маленьких программ и отдельных функций, — расширение тестов на граничные условия, описанных в предыдущем разделе: систематическое тестирование отдельных случаев.

Предположим, что у нас есть функция, осуществляющая двоичный поиск в массиве целых чисел. Начнем со следующих тестов (как нетрудно заметить, расположены они в порядке увеличения сложности):

- поиск в пустом массиве;
- поиск в массиве с одним элементом — пробное значение:
  - меньше чем элемент массиша;
  - равно элементу массива;
  - больше чем элемент массива;
- поиск в массиве с двумя элементами — пробные значения:
  - тестируем все пять возможных вариантов;
- проверяем поведение при дублировании элемента — пробные значения:
  - меньше значения в массиве;
  - равно значению в массиве;
  - больше значения в массиве;
- поиск в массиве с тремя элементами (так же, как и с двумя);
- поиск в массиве с четырьмя элементами (так же, как с двумя и тремя).

Если функция пройдет эти тесты без ошибок, она, по всей видимости, находится в неплохой форме, однако ее можно тестировать и дальше.

Приведенный набор тестов достаточно мал, чтобы выполнять их все вручную, но лучше создать оснастку (test scaffold — подмости тестирования) для механизации процесса. С этой целью мы напишем простейшую программу (по сути, драйвер). Она будет считывать строки, содержащие ключ, по которому будет производиться поиск, и размер массива; после этого будет создан массив указанного размера, содержащий значения 1, 3, 5 и т. п.; результат поиска будет выводиться на экран.

```
    /* bintest main: утилита для тестирования binsearch */
int main(v.oid)
{
int i, key, nelem, arr[1000];
while (scanf("%d %d", &key, &nelem) !=.EOF)
 { for (i = 0; i < nelem; i++)
 arr[i] = 2*1 + 1;
printf("%d\n", binsearch(key, arr, nelem));
}
return 0;
 }
```

Простейшая программка, но утилиты для тестирования и не должны быть сложными (естественно, при желании можно расширить возможности этой утилиты), их единственная задача — избавить вас от монотонных ручных проверок.

Четко определите, чего вы ожидаете на выходе теста. При проведении всех тестов вы должны четко знать правильный результат; если вы его не знаете, то напрасно теряете время. На первый взгляд мысль кажется самоочевидной, поскольку для многих программ очень просто определить, работают они или нет. Например, создается ли копия файла или нет, отсортированы ли данные на выходе или нет и т. п.

Однако для большинства программ работоспособность определить труднее, например: для компиляторов (полностью ли правильно преобразованы входные данные?), численных алгоритмов (не превышена ли допустимая погрешность вычислений?), графики (все ли пиксели находятся на своих местах?) и т. п. Для таких программ необходимо сравнивать результаты тестов с заранее известными значениями.

- Для теста компилятора скомпилируйте и запустите тестовые файлы. Результаты работы этих программ надо сравнить с заранее определенными значениями.
- Для теста вычислительной программы выберите случаи, которые позволят проверить алгоритм со всех сторон, — как простые случаи, так и сложные. Где возможно, вставляйте код, удостоверяющий корректность параметров вывода. Например, вывод программы, численно считающей интегралы, может быть проверен на непрерывность и на соответствие результату, полученному по формуле.
- Для тестирования графической программы недостаточно удостовериться, что она в состоянии нарисовать ящик; вместо этого прочтите этот ящик обратно с экрана и проверьте, что его стороны находятся там, где требуется.

Если в программе выполняются какие-то обратимые действия, убедитесь, что вы можете обратить данные в исходное состояние. Шифровка и дешифровка во многих случаях обратимы; если вы что-то зашифровали и не смогли расшифровать, значит, что-то тут не так. То же самое относится и к программам сжатия данных. Иногда существует несколько способов обратного преобразования — тогда необходимо проверить все комбинации.

Проверяйте свойства сохранности данных. Многие программы сохраняют некоторые свойства вводимых данных. Инструменты вроде we (подсчитывает строки, слова и символы) и sum (вычисляет контрольную сумму) помогут удостовериться в том, что вывод имеет тот же размер, то же количество слов или те же байты в некотором порядке и т. п. Другие программы проверяют файлы на идентичность (cmp) или перечисляют их различия (cliff). Эти программы (или сходные с ними) доступны в большинстве сред программирования, и пренебрегать ими не стоит.

Программа определения частоты появления байтов может быть использована для проверки сохранности данных; кроме того, она может выявить аномалии вроде наличия нетекстовых символов в текстовых файлах. Вот версия такой программы, которую мы назвали f req:

```
    # include <stdio.h>
  # include <ctype.h>
 # include <limits.h>
 unsigned long count[UCHAR_MAX+1];
 /* freq main: выводит частоты появления байтов */
 int main(void)
 {
 int c;
 while ((c = getchar()) != EOF) count[c]++;
 for (c = 0; c <= UCHAR_MAX; c++)
 if (countfc] != 0)
printf("%.2x %c %lu\n",
 c, isprint(c) ? c : '-', count[c]); return 0;
 }
```

Сохранность данных можно также проверять и внутри самой программы. Функция, подсчитывающая количество элементов в структуре, осуществляет простейший тест целостности данных. Для хэш-таблицы должно выполняться ее основное свойство: каждый записанный в нее элемент может быть считан. Проверить это свойство нетрудно - достаточно написать функцию, которая бы выводила содержимое таблицы в файл или массив. В любой момент для любой структуры данных разность числа включений и числа исключений должна равняться числу хранимых элементов. Проверить это условие совсем не сложно.

Сравните независимые реализации. Независимые реализации библиотек или программ должны выдавать одни и те же результаты. Например, два компилятора должны из одного и того же текста создавать программы, которые на одной и той же машине будут вести себя одинаково, — по крайней мере, в большинстве случаев.

Иногда искомый ответ можно получить двумя различными способами, а иногда — написать тривиальную версию программы, не заботясь о быстродействии и используемых ресурсах, для сравнения результатов. Если две программы независимо друг от друга выдают одни и те же ответы, с большой долей

уверенности можно считать, что обе они работают корректно; если же ответы их разнятся, значит, по крайней мере, в одной из них есть ошибки.

Один из нас однажды работал над компилятором для новой машины в паре с другим программистом. Мы разделили работу по отладке кода, генерируемого компилятором, таким образом: один писал программу, кодирующую инструкции для машины, а другой — дизассемблер для отладчика. При таком разделении ошибки интерпретации или реализации набора инструкций, для того чтобы остаться незамеченными, должны были возникнуть синхронно в обеих программах, иначе же, как только компилятор неправильно кодировал инструкцию, это сразу замечал отладчик. На ранних стадиях весь вывод компилятора прогонялся через дизассемблер и сравнивался с распечатками собственно отладчика ком- | пилятора. Такая стратегия разделения давала хорошие результаты, и благодаря ей было найдено немало ошибок в обеих частях. Единственный сложный случай, затянувший работу, возник, когда оба программиста одинаково неверно истолковали громоздкую маловразумительную фразу из описания архитектуры машины.

Оценивайте охват тестов. Одна из главных целей тестирования — убедиться, что каждое выражение в программе было выполнено хотя бы единожды при проведении последовательности тестов; тестирование нельзя считать завершенным, пока этого не произошло. Однако надо признать, что полного охвата добиться достаточно трудно. Не принимая даже во внимание выражений, обрабатывающих ситуации "не может быть", с помощью нормального ввода вынудить программу обойти все возможные ветки не так-то просто.

Для оценки охвата существуют специальные коммерческие утилиты.' Профайлеры (программы-протоколисты), часто включаемые в комплект поставки компиляторов, предоставляют возможность осуществить подсчет частоты выполнения каждого выражения программы, — отсюда можно узнать и охват каждого теста.

Мы использовали комбинацию описанных выше методов для тестирования программы markov из главы 3, эти тесты будут подробно описаны в последнем разделе главы.

Упражнение 6-3

Опишите, как вы будете тестировать f req.

Упражнение 6-4

Спроектируйте и реализуйте версию f req, которая подсчитывала бы частоты для других типов данных — таких, как 32-битовые целые или числа с плавающей точкой. Сможете ли вы добиться того, чтобы программа элегантно обрабатывала данные различных типов?

Автоматизация тестирования

Осуществлять тестирование вручную — скучно и ненадежно: при нормальном подходе вам потребуется прогнать множество тестов, перебрать множество вариантов ввода и сравнить множество результатов. Так что тестированием должны заниматься программы: они не устают и не отвлекаются от работы. Стоит потратить время на написание кода простейшей программы или просто скрипта, который включает все тесты, и все тестирование сможет выполниться от (буквально или фигурально) простого нажатия кнопки. Чем проще будет запуск тестирования, тем

реже вы станете пренебрегать им из-за спешки. Мы написали набор тестовых утилит, которыми протестировали все программы, созданные для этой книги; этот же набор мы запускали после внесения любых изменений; некоторые утилиты запускались у нас автоматически после каждой успешной компиляции.

Автоматизируйте возвратное тестирование. Одной из основных форм автоматизации является возвратное тестирование (regression testing), при котором выполняется последовательность тестов, сравнивающих очередную новую версию программы с предыдущей. При исправлении ошибок зачастую проверяются только собственно исправленные недочеты, при этом не принимается в расчет возможность внесения новых ошибок. Основное назначение возвратного тестирования — убедиться в том, что поведение программы изменилось только в предусмотренных рамках.

В некоторых системах имеется большой арсенал средств, облегчающих подобную автоматизацию; языки скриптов позволяют писать короткие сценарии для запуска последовательностей тестов. В Unix утилиты сравнения файлов вроде cliff и стр дают возможность отслеживать изменения вывода, sort группирует схожие элементы, дгер фильтрует вывод тестов, we, sum и f req подсчитывают статистику вывода. С помощью этих утилит можно без труда создать подходящие к случаю тестовые оснастки — слабоватые, возможно, для больших проектов, но вполне пригодные для программ, создающихся одним или несколькими программистами.

Ниже приведен скрипт для возвратного тестирования программы ka. (от killer application). Этот скрипт выполняет старую (old_ka) и новую (new_ka) версии программы на большом наборе тестовых файлов данных и выдает сообщения обо всех случаях, когда результаты оказались не идентичными. Скрипт написан для оболочки Unix, но его можно просто модифицировать под Perl или другой язык скриптов:

```
    for i in ka_data.*
# цикл по всем тестовым файлам данных
 do
old_ka $i >out1
# выполнить старую версию new_ka $i
>out2
# выполнить новую версию if ! cmp -s out"! out2
 # compare output files then
  echo $i: BAD
 # различаются: сообщение об ошибке
 fi done
```

Тестовый скрипт в идеале должен работать молча, выводя какие-то сообщения только при возникновении непредвиденных ситуаций, — как это делает приведенный фрагмент. Можно было бы, конечно, печатать имя каждого тестируемого файла, при необходимости дополняя его сообщением об ошибке. Подобный индикатор процесса полезен для выявления проблем, связанных с бесконечными циклами или пропуском нужных тестовых файлов, однако при нормальном течении тестов лишние подсказки на экране несколько раздражают.

Аргумент -s заставляет стр возвращать результат прохождения теста и ничего не выводить. Если сравниваемые файлы оказываются эквивалентными, стр возвращает результат "истина", тогда ! стр оказывается "ложью", и, стало быть, ничего не печатается. Однако, если файлы вывода старой и новой версий

различаются, стр возвращает значение "ложь", и выводится имя файла и предупреждение.

При регрессивном тестировании предполагается, что предыдущая версия работала правильно. Это должно быть изначально (для первой версии) тщательнейшим образом проверено, поскольку, если ошибочный ответ каким-то образом проникает в систему возвратного тестирования, его очень трудно вычислить и, кроме того, все последующие версии программы будут обречены на его повторение. Поэтому хорошей практикой следует признать периодическую проверку самого возвратного теста.

Создавайте замкнутые тесты. В дополнение к возвратным тестам полезно использовать и замкнутые тесты, которые содержат в себе и вводимые данные, и ожидаемые результаты. Здесь может оказаться поучительным наш опыт в тестировании программ на Awk. Многие конструкции языка тестируются посредством запуска крошечных программ на различных входных данных и проверкой выводимых результатов. Приведенный кусок взят из большого набора разнообразных тестов, он проверяет некоторое хитроумное инкрементное выражение. Тест запускает очередную версию программы (newawk), записывает ее выходные результаты в файл, правильные ответы записывает в другой файл с помощью команды echo, файлы сравнивает и в случае их различия сообщает об ошибке.

```
    it тест инкремента полей:
 $i++ означает($1)++, а не $(!++) echo 3 5 |
newawk '{i = 1; print $i++; print $1, i}' >out1
echo '3
4 1' >out2 # правильный ответif ! cmp -s outl out2
# результаты различаются then
echo 'BAD: тест инкремента полей не прошел'
 fi
```

Первый комментарий — важная часть входных данных теста, ибо в нем описывается, что же именно проверяет данный тест.

Иногда большой набор тестов можно создать без особых усилий. Для простых выражений мы создали небольшой специализированный язык описания тестов, вводимых данных и ожидаемых результатов. Вот небольшая последовательность, тестирующая некоторые способы представления числового значения 1 в Awk:

```
    try {if ($1 == 1) print "yes"; else print "no"}
 1 yes
1.0 yes
1EO yes
 0.1E1 yes
 10E-1 yes
 01 yes
+1 yes
 10E-2 no
 10 no
```

Первая строка — тестируемая программа (все после слова try). Каждая последующая строка является набором вводимых значений и ожидаемого результата, разделенных знаками табуляции. В первом тесте вводится значение 1 и

ожидается вывод слова yes. Первые семь тестов должны все напечатать yes, а два последних — no.

Программа на Awk (а на чем же еще?) преобразует каждый тест в полноценную же программу на Awk, далее пропускает через него каждый возможный вариант ввода и сравнивает полученные результаты с ожидаемыми; сообщается только о тех случаях, когда результат сравнения окажется отрицательным.

Схожие механизмы используются для тестирования соответствия регулярных выражений и команд замещения. Специальный малый язык для написания тестовых программ облегчит вам создание большого количества тестов; использование программы для написания программы для тестирования программы своеобразно увеличивает "плечо рычага", и работа облегчается в несколько раз. (В главе 9 мы еще вернемся к разговору о небольших языках и о программах, пишущих программы.)

Всего у нас есть около тысячи тестов для Awk; весь их набор можно запустить одной-единственной командой, и если все прошло хорошо, то никаких сообщений не появится. Каждый раз, когда в программу добавляется новая возможность или исправляется какая-нибудь ошибка, мы добавляем новые тесты для поверки новых свойств. Каждый раз, когда программа изменяется, пусть даже совсем немного, запускается весь набор тестов — его исполнение занимает лишь пару минут. Нередко при этом выявляются совершенно неожиданные ошибки; применение такого набора не раз спасало авторов Awk от конфуза.

Что же делать, если вы обнаружили ошибку? Если она не была найдена существующими тестами, создайте новый текст, нацеленный на эту конкретную проблему, и проверьте его на некорректной версии. Обнаруженная ошибка зачастую является стимулом не только для создания одного нового теста, но и вообще нового направления проверок. Кстати, не надо забывать и о том, что иногда программу можно просто снабдить механизмом защиты, который отлавливал бы ошибку.

Никогда не удаляйте созданный тест. Он может помочь вам в определении того, исправлены ли уже те или иные ошибки. Ведите учет всех ошибок, изменений, исправлений — это поможет вам опознать старые проблемы и справиться с новыми. В большинстве коммерческих программистских фирм ведение подобных записей является строго обязательным. Для вас же эти записи станут способомтюхранения времени.

Упражнение 6-5

Спроектируйте набор тестов для printf, используя при этом как можно больше автоматических способов.

Тестовые оснастки

Тестирование, о котором мы вели разговор до этого момента, относилось в основном к одной обособленной программе в завершенном виде. Это, однако, не единственный вид автоматизации тестов, так же как и не единственный способ тестирования частей большой программы в период ее написания, особенно при работе в команде. Это также и не самый эффективный способ тестирования отдельных компонентов, которые со временем должны быть объединены во что-то глобальное.

Для тестирования отдельного компонента большой программы, как правило, необходимо создать некие строительные леса (scaffold -подмости), или оснастку, которая предоставит в ваше распоряжение достаточную поддержку и достаточное взаимодействие с остальной частью системы. Мы уже приводили маленький пример подобного рода — для тестирования двоичного поиска.

Нетрудно разработать оснастку для тестирования математических, строковых функций, алгоритмов поиска и тому подобных вещей, поскольку ее создание в этих случаях сводится к установлению параметров ввода, вызову тестируемых функций и проверке результатов. Куда сложнее создать оснастку для тестирования незавершенной программы.

Чтобы проиллюстрировать повествования, мы создадим тест для memset, одной из функций семейства mem... стандартной библиотеки C/C++. Эти функции часто пишутся на языке ассемблера для,конкретных машин, поскольку их быстродействие очень важно. Однако чем более тонко они настраиваются на конкретные условия, тем больше вероятность возникновения в них ошибок и тем более тщательно должны они тестироваться.

Первый шаг — создать наиболее простые версии на C, заведомо работоспособные; они будут эталоном для сравнения быстродействия и, что более важно, проверки правильности. При переходе в новую среду разработки наши простейшие версии останутся эталоном до тех пор, пока не заработает специально созданная "родная" версия.

Функция memset (s, c, n) записывает байт c в n байтов памяти, начиная с адреса s, и возвращает s. Если нет ограничений на скорость работы, написать такую функцию,— не проблема:

```
    /* memset: устанавливает
  первые n байтов s равными c */
  void *memset(void *s, int c, size_t n)
  {
  size_t i;
 char *p;
  p = (char *) s;
  for (i = 0; i < n; i++)
  p[i] = c; return s; }
```

Но как только главным параметром становится быстродействие, приходится прибегать к различным трюкам, вроде записи слов в 32 или 64 бита за раз. Подобные изыски могут вызвать появление ошибок, поэтому глобальное тестирование становится строго обязательным.

Тестирование базируется на комбинации всесторонних проверок (в частности, естественно, проверок граничных условий в потенциально опасных точках). Для memset граничными, очевидно, являются такие значения n, как ноль, один и два, числа, являющиеся степенями двойки, а также соседние с ними значения — от самых маленьких до громадных, вроде 216, что соответствует естественной границе во многих машинах — 16-битовому слову. Степени двойки привлекают внимание из-за того, что один из способов ускорить работу memset — устанавливать одновременно несколько байтов; это может быть выполнено с помощью специальных инструкций или посредством установки сразу не байта, а слова. Также надо проверять начальные значения массивов при различных выравниваниях — на

случай, если ошибка возникает из-за стартового адреса или длины. Мы поместим используемый массив внутрь большего массива, создав тем самым некую буферную зону, или запасной отступ с каждой стороны — для того, чтобы можно было не особо ограничивать себя в выборе выравнивания.

Кроме перечисленного нам надо проверить еще множество значений для с — включая ноль, Ox7F (самое большое значение для числа со знаком при 8-битовых байтах)-, 0x80 и OxFF (проверяя на потенциальные ошибки, связанные со знаковыми и беззнаковыми символами) и значения, превышающие один байт (чтобы удостовериться, что используется только один байт). Нам надо также записать в память некий шаблон, отличающийся от любого из этих значений, — с тем чтобы иметь возможность проверить, не производила ли memset запись вне границ предназначенной области.

Мы можем использовать нашу простую реализацию как стандарт для сравнения в тесте, который размещает в памяти два массива, а затем сравнивает поведение разных реализаций при разных значениях п, с и отступа внутри массива:

```
    big = максимальная левая граница + maximum n
 + максимальная правая граница
sO = malloc(big)
s1 = malloc(big)
 для каждого значения параметров n,
с и отступа offset: установить sO и s1
в шаблонное значение выполнить медленный
 memset(sO + offset, с, п) выполнить быстрый
 memset(s1 + offset, с, п) проверить возвращаемые
 значения сравнить содержимое sO и s1 побайтово
```

Ошибка, вынуждающая memset писать вне границ своего массива, скорее всего, проявится в байтах рядом с началом и концом массива, так что, оставив буферную зону, проще увидеть поврежденные байты. Уменьшается и вероятность того, что будут перезаписаны какие-то части программы в памяти. Для проверки записи вне границ мы должны проверить все значения sO и s1, а не только те п байтов, которые должны были быть записаны.

Таким образом, наш набор тестов должен включить в себя проверку всех комбинаций значений:

offset = 10, 11, .., 20
c = 0, 1, Ox7F, 0x80, OxFF, 0x11223344
n = 0, 1, 2, 3, 4, 5, 7, 8, 9, 15, 16, 17,
31, 32, 33, ... 65535, 65536, 65537

Для n должны быть подставлены, по крайней мере, значения 2' - 1, 2' и 2' + 1 для всех г от 0 до 16.

Все перечисленные значения, естественно, не надо встраивать в основную часть оснастки — надо предусмотреть возможность записывать их в отдельные массивы — вручную или программно. Лучше генерировать их программно — тогда не составит труда задать больше степеней двойки, включить большее количество разных отступов или больше символов.

Наши тесты заставят memset поработать на совесть; написать же их совсем не долго, не говоря уже об исполнении — всего надо проверить менее 3500 комбинаций. Все тесты полностью переносимы, так что при необходимости их можно использовать в любой среде.

С тестированием memset связана одна история, которая может послужить вам хорошим уроком. Однажды мы дали копию тестов для memset одному программисту, разрабатывавшему операционную систему и библиотеки для нового процессора. Через несколько месяцев мы (авторы тестов) начали работать с этой новой машиной. В какой-то момент большое приложение не прошло своего набора тестов. Мы стали искать причины и после кропотливого труда докопались до истоков — проблема состояла в трудноуловимой неточности, связанной со знаковым расширением" в реализации memset на ассемблере. По непонятным причинам создатель библиотеки изменил тесты для memset, исключив из них проверку значений, больших Ox7F. Естественно, ошибка была найдена при запуске изначальной версии теста сразу после того, как подозрение пало на memset.

Функции типа memset хорошо поддаются проверке замкнутыми тестами, потому что они достаточно просты для того, чтобы можно было подобрать тестовые данные, перебрав все возможные варианты и охватив тем самым весь код. Так, для функции memmove можно перебрать все возможные комбинации различных значений перекрытия, направления и выравнивания. Этого, конечно, недостаточно для проверки всех операций копирования, но достаточно для тестирования всех возможных значений вводимых параметров.

Как в любом тестовом методе, тестовой оснастке для проверки результатов операций нужно знать правильные ответы. Важнейшим является способ, использованный нами для тестирования memset, — создание простейшей версии тестируемой функции и сравнение ее результатов с результатами основных тестов. Это можно осуществлять в несколько этапов, как будет показано в следующем примере.

Один из авторов некогда создавал библиотеку для работы с растровой графикой; среди прочих в этой библиотеке существовал оператор для копирования блоков пикселей из одного изображения в другое. В зависимости от параметров эта операция осуществлялась как простое копирование памяти, или требовала преобразования пикселей из одного цветового пространства в другое, или выполняла мозаичное размещение введенного образца в прямоугольной области, или использовала комбинацию этих и некоторых других способов. Спецификация оператора выглядела просто, реализация же его требовала написания большого количества специфического кода для обработки всех возможных случаев. Для того чтобы убедиться в правильности всего кода, требовалась хорошая стратегия тестирования.

Сначала вручную был написан простейший кож, осуществляющий корректные операции для одного пикселя, который использовался для тестирования работы функций библиотеки с одним пикселем. Завершив этот этап, можно было быть уверенным в том, что для случая одного пикселя оператор библиотеки работает корректно.

Далее был написан код, использующий эту отлаженную однопиксельную обработку, — получился прообраз (медленный и неудобный, но это неважно) оператора, работающего с одной горизонтальной строкой пикселей; с этим прообразом и

производилось сравнение библиотечной обработки строк. По окончании данного этапа библиотека была проверена на обработку строк пикселей.

Далее так же последовательно строки использовались для обработки прямоугольных областей, те в свою очередь — для создания мозаик и т. д. По ходу дела было обнаружено немало ошибок, в том числе и в тестовой программе, но это-то как раз и является одним из преимуществ методики: тестируются две независимые версии, при этом обе — корректируются. Если какой-то тест не проходит, в первую очередь распечатываются результаты работы тестирующей программы, и на их основе выясняется место возникновения ошибки и проверяется корректность самой тестирующей версии.

Библиотека изменялась и переписывалась под разные платформы много лет, и тестирующая версия не раз оказывалась незаменимым инструментом для поиска ошибок.

При использованном поэтапном подходе тестирующая программа должна была запускаться каждый раз заново для проверки уверенности в работе библиотеки. Кстати, в данном случае тесты были не замкнутыми, а скорее вероятностными: тестовые задания генерировались случайным образом, и при достаточно большом количестве запусков можно было с хорошей вероятностью считать, что все возможные варианты (и, стало быть, все ветви кода) оказались проверенными. При большом количестве вариантов тестовых случаев такая стратегия более удачна, чем создание наборов тестов вручную, и гораздо более эффективна, чем замкнутое тестирование.

Упражнение 6-6

Создайте, основываясь на описанных нами приемах, тестовую оснастку для memset.

Упражнение 6-7

Создайте тесты для остальных функций семейства mem. . . .

Упражнение 6-8

Определите режим тестирования для числовых методов типа sq rt, s i n и им подобных из библиотеки math. h. Какие вводимые значения имеют смысл? Какие независимые проверки могут быть осуществлены?

Упражнение 6-9

Определите механизмы для тестирования функций С семейства st r. . . (например, st rcmp). Некоторые из этих функций, особенно те, что служат для разбиения на лексемы — типа st rtok или st rcspn, значительно сложнее, чем функции семейства mem. .., и, следовательно, для их проверки потребуются более изощренные тесты.

Стрессовое тестирование

Еще один эффективный прием тестирования — проверка программ большими объемами вводимых данных, сгенерированных компьютером. Входные данные, сгенерированные машиной, оказывают на программы несколько иное влияние, чем созданные вручную. Большие объемы сами по себе могут стать причиной сбоев, вызывая переполнение буферов ввода, массивов, счетчиков; они весьма

эффективны для поиска неоправданно ограниченных в размерах структур данных. Кроме того, люди часто неосознанно избегают "невозможных" значений (вроде пустой строки ввода или ввода недопустимых значений) и нечасто вводят очень длинные имена или очень большие значения. Компьютеры же генерируют данные точно в соответствии с запрограммированными зада-; ниями; у них нет никаких личных предпочтений или антипатий.

Вот для иллюстрации одна строка вывода, произведенного компиля-. тором Microsoft Visual C++ версии 5.0 при компиляции нашей программы markov (версия C++ с использованием STL):

```
    xtree(114) : warning C4786: ' std :
 :_Tree<std : :deque<std :
 : basic_string<char,std:
 :char J:raits<char>«< std: '.allocator <char», std:
 :allocator<std : :basic_string<char, std: :
  . . . опущено 1420 символов
 allocator<char»»»:
 :iterator' : identifier was truncated to '255' characters
 in the debug information
```

Компилятор предупреждает нас, что им было сгенерировано имя переменной с замечательной длиной в 1594 символа, но только 255 из них были сохранены в отладочной информации. Не все программы защищают себя от таких необычно длинных строк.

Выбор вводимых значений (не обязательно корректных) случайным образом — еще один достойный способ испытания программы на прочность. Это как бы дальнейшее развитие подхода "человек бы так не сделал". Некоторые коммерческие компиляторы C, например, тестируются посредством сгенерированных случайным образом (но синтаксически корректных) программ. Смысл состоит в том, чтобы использовать спецификацию проблемы — в данном случае стандарт C — для создания программы, генерирующей допустимые, но неестественные тестовые данные.

Подобные тесты полагаются на встроенные в программу механизмы защиты; поскольку удостовериться в правильности результатов удается не всегда, главной целью может быть провоцирование сбоя или непредусмотренной ситуации. При этом также проверяется и код, обрабатывающий ошибки. Если вводить только осмысленные, реалистичные данные, большинство ошибок ввода никогда не случится, и, следовательно, обрабатывающий их код не будет исполнен, а в нем могут таиться ошибки. Надо сказать, что иногда тестирование случайными данными может зайти слишком далеко и обнаружить ошибки, которые настолько маловероятны в реальности, что их можно и не исправлять.

Некоторые виды тестов основаны на введении преднамеренно некорректных данных. При попытках взлома часто используют объемистый или некорректный ввод, который перезаписывает ценные данные: имеет смысл самому проверить свою программу на восприимчивость к такому вводу. Некоторые функции стандартных библиотек оказываются уязвимы для подобных атак. Например, в функции gets из стандартной библиотеки не предусмотрено никакого способа ограничения размера вводимой строки, поэтому ее нельзя использовать никогда; вместо нее нужно применять функцию fgets(buf, sizeof(buf), stdin). В обычном своем простейшем формате функция scanf ("%s", buf) также не ограничивает размер вводимой строки,

поэтому ее можно использовать, только указывая размер строки в явном виде: scanf ("%20s", buf). В разделе 3.3 мы показали, как решить эту проблему для буфера произвольного размера.

Любой блок, который может получать данные извне программы (прямо или косвенно), должен проверять полученные значения перед тем, как их использовать. Следующая программа, взятая из учебника, по идее должна читать целое число, введенное пользователем, и, если это число слишком велико, выдавать предупреждение. Цель создания этой программы — продемонстрировать, как можно справиться с проблемой gets, однако предложенное решение работает не всегда:

```
    ? tfdefine MAXNUM 10 ?
? int main(void)
 ? {
? char num[MAXNUM];
 ?
? memset(nurn, 0, sizeof(num));
? printf("Введите число: ");
? gets(num);
? if (num[MAXNUM-1] != 0)
? printf("Число слишком велико!\n");
 ? /*...*/
 ?
```

Если вводимое число состоит из 10 символов, оно перепишет последний нуль массива num ненулевым значением, и по теории это может быть замечено после возврата gets. К сожалению, подобное решение нельзя признать удовлетворительным. Злонамеренный взломщик введет еще более длинную строку, которая перепишет какие-нибудь критические значения, — может быть, адрес возврата вызова, — и тогда программа никогда не вернется к выполнению условия if, а выполнит вместо этого инструкции взломщика. Вообще стоит запомнить, что любой неконтролируемый ввод есть, кроме всего прочего, еще и потенциальная лазейка для взлома системы.

Чтобы вы не думали, что описанные проблемы возможны только в программах из плохих учебников, вспомните о том, как в июле 1998 года ошибка подобного рода обнаружилась в нескольких основных программах электронной почты. Как писала New York Times:

«Лазейка в системе безопасности была вызвана тем, что принято называть "ошибкой переполнения буфера". Программисты должны включать в свои программы код, который проверяет, что вводимые данные имеют нужный тип и размер. Если элемент данных слишком велик, or может выйти за границу "буфера" — кусок памяти, специально выделенный для его хранения. В этом случае программа электронной'почты даст сбой, и злоумышленник может заставить ваш компьютер выполнять его программу». Среди атак во время знаменитого инцидента "Internet Worm" ("Сетевой червь") 1988 года была и такая.

Программы, производящие разбор форм HTML, также могут быть чувствительны к атакам, основанным на хранении очень длинных строк ввода в маленьких массивах:

```
    ? static char query[1024]; ?
? char *read_form(void)
? {
```

```
? int qsize;
?
? qsize = atoi(getenv("CONTENT_LENGTH"));
 ? fread(query, qsize, 1, stdin);
 ? return query;
 ? }
```

В этом коде предполагается, что ввод никогда не будет длиннее 1024 байтов, поэтому он, как и gets, открыт для атак переполнения буфера.

Более привычные виды переполнения также могут вызвать проблемы. Если переполнение целого числа происходит без предупреждения, результат может быть губительным для программы. Рассмотрим такое выделение памяти:

```
    ? char *p;
? p = (char *) malloc(x * y * z);
```

Если результат перемножения x, y и z вызывает переполнение, вызов malloc приведет к созданию массива приемлемого размера, но p[x] может ссылаться на раздел памяти вне выделенной области. Предположим, что целые числа являются 16-битовыми, а каждая из переменных x, y и z равна 41. Тогда x*y*z равно 68 921, то есть 3385 по модулю 216. Так что вызов malloc выделит только 3385 байтов, а любая ссылка по индексу вне этого значения будет выходить за заданные границы.

Еще одной причиной переполнения может стать преобразование типов, и обработки таких ошибок не всегда возможны корректным об-, разом. Ракета "Ariane 5" взорвалась во время своего первого запуска в июне 1996 года только из-за того, что навигационный пакет был унаследован от "Ariane 4" и не прошел тщательного тестирования. Новая ракета имела большую скорость, и, соответственно, навигационным программам приходилось иметь дело с большими числами. Вскоре после запуска попытка преобразовать 64-битовое число с плавающей точкой в 16-битовое целое со знаком вызвала переполнение. Ошибка была отловлена, но коду, обработавшему ее, пришлось прервать работу подсистемы. Ракета ушла с курса и взорвалась. Самое обидное, что код, в котором произошел сбой, генерировал данные, необходимые только до момента запуска; если бы при запуске эта часть программы была отключена, трагедии бы не произошло.

На более приземленном уровне двоичный ввод иногда выводит из строя программы, ожидающие текстового ввода, особенно если предполагается, что этот ввод должен относиться к 7-битовому набору символов ASCII. Полезно, а для многих ситуаций и просто необходимо ввести двоичные значения в тестируемую программу, ожидающую ввода текста, и посмотреть на ее реакцию.

Хорошие тесты и тестовые случаи нередко могут быть использованы для большого количества программ. Так, каждая программа, читающая файлы, должна быть проверена вводом пустого файла. Каждая программа, читающая текст, должна быть оттестирована двоичным вводом. Каждая программа, читающая строки текста, должна быть проверена вводом очень длинных строк, пустых строк и файлом без символов перевода строки вообще. Таким образом, полезно сохранять подобные общие тесты и всегда иметь их под рукой — тогда можно будет быстрее и с меньшими затратами тестировать любую программу. Полезно также раз и навсегда написать хорошую программу, которая бы создавала тестовые файлы в соответствии с запросами.

Когда Стив Борн (Steve Bourne) писал свою оболочку для Unix (которая теперь известна как Bourne shell), он создал каталог, содержащий 254 файла с именами из одного символа: по одному на каждое возможное значение байта, за исключением '\0' и косой черты — двух символов, которые не могут встретиться в именах файлов Unix. Этот каталог он всячески использовал для тестирования поисков по шаблону и программ разбиения входного потока. (Надо ли говорить, что каталог этот был создан программно.) Через много лет этот каталог стал настоящим проклятием программ обхода дерева файлов — множество тестов с его участием приводили к плачевным для этих программ результатам.

Упражнение 6-10

Постарайтесь создать файл, который бы вызвал сбой в вашем любимом текстовом редакторе, компиляторе или другой программе.

Полезные советы

Опытные тестеры имеют в активе большое количество способов тестирования, а также разнообразных уловок и ухищрений, которые делают их работу более продуктивной. В этом разделе мы поделимся с вами своими любимыми приемами.

Программы должны проверять границы массивов (если за них этого не делает собственно язык программирования), однако код таких проверок не обязательно тестировать в том случае, когда размеры массивов достаточно велики по сравнению с типичным вводом. Для выполнения таких проверок временно уменьшите размеры массивов — тогда вам не придется создавать очень больших тестовых случаев. Схожий прием .мы использовали в коде для приращения массива в главе 2 и в библиотеке CSV из главы 4. На самом деле мы даже оставили эти небольшие начальные значения, поскольку дополнительные затраты при запуске в данном случае несущественны.

Пусть при тестировании ваша хэш-функция возвращает константу — тогда каждый элемент будет записываться на одно и то же место. При этом будет работать механизм цепных списков; кроме того, это даст вам представление о быстродействии для самого худшего случая.

Напишите версию выделения памяти, которая специально даст сбой в скором времени — и вы протестируете код, восстанавливающий систему после ошибок нехватки памяти. Вот, например, версия, которая после 10 вызовов возвращает NULL:

```
    /* testmalloc: через 10 вызовов возвращает
NULL */ void *testmalloc(size_t n)
 {
 static int count = 0;
 if (++count > 10)
 return NULL; else
 return malloc(n); }
```

Перед тем как распространять свой код, уберите из него все тестовые ограничения — они могут повлиять на его производительность. Однажды мы долго бились над проблемой производительности одного коммерческого компилятора и в конце концов нашли причину: некая хэш-функция всегда возвращала 0 из-за того, что из нее не был удален код для тестирования.

Инициализируйте массивы и переменные некоторым запоминающимся характерным значением, а не 0, как это принято, — тогда, если произойдет выход за заданные границы или обнаружится неинициализированная переменная, вам будет проще заметить это. Будьте изобрета-" тельны, — например, константу OxDEADBEEF легко опознать в отладчике.

Варьируйте тестовые случаи, особенно при ручном тестировании, — иначе нетрудно попасть в узкую колею и все время тестировать одно и то же; при этом можно пропустить ошибку в каком-то другом месте.

Никогда не продолжайте писать новые блоки кода или даже тестировать старые, если существуют уже найденные ошибки — они могут повлиять на результаты тестов.

Вывод тестов должен включать в себя полный набор параметров ввода, с тем чтобы тест можно было в точности воспроизвести. Если ваша программа использует случайные числа, найдите способ устанавливать и выводить начальные условия вне зависимости от того, случайно ли выбираются сами тесты. Убедитесь в том, что тестовые ввод и вывод идентифицируются должным образом, чтобы их без труда можно было сопоставить, осмыслить и воспроизвести.

Полезно предусмотреть способы управления количеством отладочной выдачи и ее видом — дополнительная выходная информация может облегчить тестирование.

Тестируйте на разных машинах, компиляторах и операционных системах. Каждая новая комбинация может вскрыть ошибки, незаметные (или несуществующие) в других случаях, такие как порядок байтов, размер целых чисел, обработка пустых указателей, обработка символов возврата каретки и перевода строки, а также некоторые специфические свойства библиотек и заголовочных файлов. Тестирование на разных машинах может также выявить проблемы с окончательной сборкой компонентов и, как мы обсудим в главе 8, указать на неумышленную зависимость от среды разработки.

Тесты быстродействия мы обсудим в главе 7.

Кто осуществляет тестирование?

Тестирование, проводимое создателем кода или кем-то другим, имеющим тем не менее доступ к исходному коду, называется тестированием белого ящика. Термин придуман по аналогии с тестированием черного ящика, — это когда тестер не знает, как реализован компонент. Лучше передает суть происходящего, на наш взгляд, термин "прозрачный ящик". Свой код тестировать, конечно, необходимо — не ждите, что некая мифическая тестирующая организация или пользователь сделают это вместо вас. Однако мы зачастую склонны обманывать сами себя и верить в лучшее, поэтому при тестировании вам надо отрешиться от кода и стараться придумать как можно более каверзные ходы. Вот как Дон Кнут (Don Knuth) описывает процесс создания тестов для системы форматирования TEX: "Я роюсь в самых мерзких и отвратительных уголках своего мозга, потом сажусь и пишу самый мерзкий [тестирующий] код, который только могу придумать, после чего стремительно меняюсь и встраиваю его в еще более мерзкие конструкции — такие мерзкие, что и говорить противно". Цель тестирования — найти ошибки, а не объявить, что программа работает. Стало быть, тесты должны быть жесткими, и если с их

помощью вы обнаруживаете проблемы, то это является доказательством действенности ваших методов, а не сигналом тревоги.

Тестирование черного ящика означает, что тестер не имеет никакого представления ни о доступе к коду, ни о его внутреннем устройстве. При таком тестировании выявляются несколько иные сшибки, поскольку тестер имеет иные отправные точки для поиска. Хорошо начинать с проверки граничных условий; после этого стоит перейти к большим объемам и некорректному вводу. Естественно, не надо забывать и о "золотой середине": проверке работы программы в нормальных условиях.

Следующий этап — реальные пользователи. Новые пользователи находят новые ошибки, потому что они опробуют программу неожиданными способами. Важно провести этот этап тестирования до того, как вы выпустите свое детище в большой мир, однако, к сожалению, многие программы распространяются без прохождения достаточного тестирования какого бы то ни было вида. Бета-версии программ — попытка привлечь большое количество конечных пользователей к тестированию, однако такие бета-версии нельзя рассматривать как замену нормальному тестированию. Однако чем больше и сложнее становятся системы, чем короче сроки на их разработку, тем выше вероятность появления плохо оттестированных продуктов.

Трудно тестировать интерактивные программы, особенно если в них обрабатывается ввод с помощью мыши. Некоторые тесты можно выполнить с помощью сценариев (их конкретные возможности зависят от языка, среды и т. п.). Интерактивные программы можно контролировать из скриптов, имитирующих поведение пользователя. Здесь есть два способа: первый — перехватить действия реального пользователя и воспроизводить их; второй — создать язык скриптов, позволяющий описать последовательность и протяженность событий.

Наконец, стоит задуматься и о том, как тестировать сами тесты. В главе 5 мы упомянули о проблеме, вызванной некорректностью программы, которая тестирует пакет функций для работы со списками. Набор возвратных тестов, содержащий ошибку, вызовет проблемы во всех версиях, и вообще, результат выполнения набора тестов вряд ли будет что-то значить, если сами тесты окажутся некорректными.

Тестирование программы markov

Программа ma rkov из главы 3 достаточно сложна, поэтому ее надо особенно тщательно оттестировать. Производит она белиберду, которую трудно проверить на корректность, и, кроме того, мы написали несколько версий на разных языках. И последнее затруднение — вывод программы случаен по определению, и по идее при каждом запуске должен изменяться. Как же применить уроки данной главы к тестированию такой программы?

Первый набор тестов состоит из нескольких крошечных файлов — для проверки граничных условий. Цель этого этапа — убедиться в том, что программа работает нормально при вводе размером всего в несколько слов. Для префиксов длиной два мы использовали пять файлов, содержащих, соответственно (по одному слову-символу на строку!):

```
   (пустой файл)
a
a b
```

```
a b c
abcd
```

Для .каждого из приведенных файлов вывод должен быть тождественен вводу. При этой проверке были обнаружены несколько ошибок на единицу при инициализации таблицы, а также при запуске и остановке генератора.

Второй тест проверял сохранность данных. Для префиксов из двух слов каждое слово, каждая пара слов и каждая тройка слов в выходном тексте должны содержаться также и во введенном тексте. Мы написали программу на Awk, которая считывает входной текст в гигантский массив, строит массивы пар и троек слов, потом считывает вывод программы в другой массив и сравнивает массивы:

```
    # markov test: проверяет, что все слова,
пары и тройки слов
# в выводе ARGV[2]
 есть в исходном тексте ARGV[1]
 BEGIN {
 while (getline <ARGV[1J > 0) for (i = 1; i <= NF; i++)
{
 wd[++nw] = $1 # слова во вводе single[$i]++ }
 for (i = 1; i < nw; i++)
 pair[wd[i],wd[i+1]]++ for (i = 1; i < nw-1; i++)
tnple[wd[i],wd[i+1],wd[i+2]]++
 while (getline <ARGV[2] > 0) {
 outwd[++ow] = $0 # слова в выводе if (!
($0 in single))
 print "постороннее слово", $0
 }
 for (i = 1; i < ow; i++)
 I if (!((outwd[i],outwd[i+1])
in pair))% print "посторонняя пара", outwd[i], outwd[i+1] for

(1=1; i < ow-1; i++)
if (!((outwd[i],outwd[i+1],outwd[i+2]) in triple))
print "посторонняя тройка",
 outwd[i], outwd[i+1], outwd[i+2]
 }
```

Мы не пытались сделать этот тест особо эффективным, наоборот, хотели лишь написать как можно более простую программу. Сравнение 10 000 слов вывода с 42 685 словами ввода занимает у нее шесть или семь секунд — не дольше, чем компилируются некоторые версии самой программы markov. Проверка сохранности данных обнаружила важную ошибку в версии, написанной на Java: программа иногда переписывала значения таблицы, поскольку использовала ссылки вместо того, чтобы создавать копии префиксов.

Приведенный тест иллюстрирует принцип, согласно которому проще бывает проверить свойства результата, чем получить этот результат. Например, проще удостовериться в том, что файл отсортирован, чем выполнить саму сортировку.

Третий тест — статистический по своей природе. Ввод состоит из последовательностей

abcabc ... abd ...

в которых на одно вхождение abd приходится десять вхождений abc. Теперь, если генератор случайных чисел работает правильно, в выводе должно быть примерно в десять раз больше c, чем d. Проверяли мы это, естественно, с помощью f req.

Статистический тест показал, что ранняя Java-версия программы, в которой с каждым суффиксом ассоциировался счетчик, выводит около, двадцати с на каждое d, то есть в два раза больше, чем предполагалось. Немного поломав голову, мы осознали, что генератор случайных чисел в Java возвращает как положительные, так и отрицательные целые значения; множитель два появился, таким образом, из-за того, что диапазон значений для генератора был в два раза больше ожидаемого и поэтому первый элемент в списке выпадал чаще (а это была именно буква c). Исправить ошибку оказалось гораздо проще, чем найти, — достаточно взять значения по модулю. Без этого теста мы никогда не нашли бы ошибки, на глаз вывод выглядел совершенно нормально.

Наконец, мы скормили программе нормальный английский текст для того, чтобы убедиться, что на выходе будет очаровательная нелепица. Естественно, этот тест мы производили и на ранних стадиях написания программы. Однако теперь, даже убедившись, что программа нормально обрабатывает те данные, для которых, собственно, и создавалась, мы не прекратили тестирования. Всегда приятно оттестировать простые случаи и убедиться, что все в порядке, однако трудные случаи также должны быть проверены. Автоматизированное, систематическое тестирование — единственный способ обойти все ловушки.

Весь процесс тестирования программы markov был автоматизирован. Специальный скрипт генерировал необходимые входные данные, запускал тесты, отмечал время их работы и распечатывал аномальные результаты вывода. Скрипт мы написали настраиваемый, так что одни и те же тесты можно было применить к версии на любом языке: каждый раз при внесении изменений в одну из программ мы без дополнительных усилий прогоняли на ней все тесты.

Заключение

Чем лучше вы пишете код изначально, тем меньше ошибок он будет содержать. Тестирование граничных условий непосредственно при написании кода — эффективный способ удалить множество мелких глупых ошибок. Систематическое тестирование проверяет потенциально уязвимые места в строгом порядке; опять же, сбои чаще всего происходят на каких-то границах, которые можно проверить вручную или программно. Как можно шире используйте автоматизированное тестирование, поскольку машины не делают ошибок, не устают и не занимаются самообманом. Возвратные тесты проверяют, что результаты работы программы изменились при внесении изменений только так, как надо. Вообще, тестирование после внесения каждого, даже небольшого, изменения — верный способ локализации источника проблем, поскольку новые ошибки появляются, как правило, именно в новом коде.

Главное же правило тестирования — делать его.

Дополнительная литература

Один из способов узнать побольше о тестировании — изучить примеры на основе лучших образцов доступных программ. В статье Дона Кнута "Ошибки в TEX",

опубликованной в Software — Practice and Experience (Don Knuth. The Errors of TEX. Software — Practice and Experience, 19, 7, p. 607-685, 1989), описываются все ошибки, найденные к тому времени в системе TEX, и обсуждаются использованные Кнутом методы тестирования. Тест TRIP для TEX представляет соб^й отличный пример основательного комплекса тестирования. Perl также предлагает расширенный набор тестирования, предназначенный для проверки его правильности после компиляции и установки на новой системе и включающий такие модули, как MakeMaker и TestHarness, которые помогают в создании тестов для расширений Perl.

Ион Бентли (Jon Bentley) написал серию статей в Communications of the ACM, которые были собраны в сборниках "Programming Pearls" (1986) и "More Programming Pearls" (1988), изданных Addison-Wesley. В этих статьях затрагиваются вопросы тестирования, главным образом структуры для организации и автоматизации расширенных тестов.

Производительность

- Узкое место
- Замеры времени и профилирование
- Стратегии ускорения
- Настройка кода
- Эффективное использование памяти
- Предварительная оценка
- Заключение
- Дополнительная литература

Он был силен и много обещал, — Свершений нет,
а силы растерял.

Шекспир. Король Генрих VIII

Когда-то программисты затрачивали огромные усилия на то, чтобы сделать свои программы эффективными, так как компьютеры были очень медленными и очень дорогими. В наши дни компьютеры стали гораздо быстрее и сильно подешевели, так что необходимость в идеальной эффективности заметно снизилась. Стоит ли по-прежнему волноваться из-за производительности?

Да, но только в том случае, если проблема действительно важна, если программа действительно работает слишком медленно, а главное, есть надежды на то, что ее удастся ускорить, не потеряв при этом в корректности, строгости и ясности. Быстрая программа, выдающая неправильные результаты, никому времени не сэкономит.

Таким образом, первым принципом оптимизации можно считать принцип не делать лишнего. Достаточно ли хороша программа в своем теперешнем состоянии? Мы знаем, как и где она будет использоваться; будут ли заметны выгоды от ее ускорения? Программы, которые пишутся в качестве заданий в колледже, никогда больше не используются, их скорость редко имеет значение. Скорость также редко имеет значение для программ, написанных для собственного использования, временных утилит, испытательных стендов, экспериментов и программ-прототипов. Однако же скорость исполнения коммерческого продукта или центрального компонента системы, например графической библиотеки, может иметь первостепенную важность, так что нам надо знать, как подходить к вопросам производительности.

Когда надо пытаться ускорить программу? Как это можно сделать? На какие результаты мы можем рассчитывать? В этой главе мы обсудим, как добиться того, чтобы программа выполнялась быстрее или использовала меньше памяти. Как правило, больше всего нас интересует скорость программы, так что в основном речь пойдет о ней. Проблемы с памятью, оперативной или внешней, возникают реже, но иногда они могут быть критичными, так что мы затронем и этот аспект.

Как мы уже выяснили в главе 2, для выполнения задания сначала лучше выбрать самые простые и понятные алгоритмы и структуры данных. После этого надо измерить производительность и решить, нужны ли изменения. Далее следует настроить опции компилятора на создание наибо- : лее быстрого кода; потом оценить, какие изменения в самой программе могут дать наибольший эффект; потом

начинать вносить изменения — по одному за раз! — и после каждого повторять предыдущие шаги. При этом J всегда надо сохранять простые версии, чтобы продолжать сравнения.

Измерение необходимо при повышении производительности, поскольку умозаключения и интуиция — не вполне надежные советники, и без дополнительных инструментов, таких как команды измерения времени и профилировщики, не обойтись. Увеличение производительности име- 5 ет много общего с тестированием, в этом процессе в той же мере важны такие вещи, как автоматизация, ведение тщательных записей об изменениях и использование возвратных тестов, чтобы видеть, что состояние программы улучшается и не придется откатываться к предыдущим версиям. Если вы выбрали алгоритмы достаточно разумно и пишете аккуратно с самого начала, то очень может быть, что никаких мер для убыстрения ваших программ не потребуется. Нередко для разрешения проблем с производительностью в грамотно спроектированном коде хватает совсем небольших изменений, а вот в плохо спроектированном коде придется переписывать очень многое.

Узкое место

Нам хотелось бы начать с описания того, как мы избавились от узкого места в важной программе нашей вычислительной системы.

Наша входящая почта поступала к нам через одну машину, называемую шлюзом (gateway), которая объединяла нашу внутреннюю сеть с внешним Интернетом. Электронная почта, приходящая извне, — в нашу организацию, насчитывающую несколько тысяч человек, приходят десятки тысяч писем в день — поступает на шлюз и затем передается во внутреннюю сеть; такое разделение изолирует нашу локальную сеть от доступа из Интернета и позволяет указывать адрес только одной машины (этого самого шлюза) для всех членов организации.

Одной из услуг, предоставляемых шлюзом, является защита от "спама" (spam — мясные консервы, содержащие в основном сало), незатребованной почты, рекламирующей услуги сомнительных достоинств. После первых успешных испытаний спам-фильтр был установлен на шлюз и включен для всех пользователей нашей внутренней сети — и немедленно возникла проблема. Машина, исполняющая роль шлюза, уже несколько устаревшая и без того достаточно загруженная, была буквально парализована: поскольку фильтрующая программа работала слишком медленно, она отнимала гораздо больше времени, чем вся остальная обработка сообщений, и в результате доставка почты задерживалась на часы.

Это пример настоящей проблемы производительности: программа была не в состоянии уложиться во время, отводимое ей на работу, и пользователи серьезно от этого страдали. Программа должна была работать гораздо быстрее.

Несколько упрощая, можно сказать, что спам-фильтр работает примерно так: каждое входящее сообщение рассматривается как единая строка, которая обрабатывается программой поиска образцов с целью обнаружить, не содержит ли она фраз из заведомого спама — таких, как "Make millions in your spare time" (сделайте миллион в свободное время) или "XXX-rated" (крутые порно). Подобные сообщения имеют тенденцию появляться многократно, так что подобный подход достаточно эффективен, тем более что если какой-то спам проходил через фильтр, то характерные фразы из него добавлялись в список.

Ни одна из существующих утилит сравнения строк — вроде д rep — не устраивала нас по соотношению производительности и возможностей, поэтому для спам-фильтра была написана специальная программа. Первоначальный код ее был весьма прост, он просматривал каждое сообщение и проверял наличие в нем заданных фраз (образцов):

```
    /* isspam: проверяет mesg на
  вхождение в него образцов pat */
 int isspam(char *mesg)
  {
  int i;
 for (i = 0; i < npat; i++)
  if (strstr(mesg, pat[i]) != NULL) {
  printf("spam: совпадает с -%s'\ri", pat[i]); return 1;
 }
 return 0;

    }
```

Как можно сделать этот код более быстрым? Нам нужно искать в строке, а лучшим способом для этого является функция st rst r из библиотеки языка C: она стандартна и эффективна.

Благодаря профилированию — технологии, о которой мы поговорим в следующем параграфе, — мы выяснили, что реализация st rst r такова, что использование ее в спам-фильтре неприемлемо. Изменив способ работы st rst r, можно было сделать ее более эффективной для данной конкретной проблемы.

Существующая реализация st rst r выглядела примерно так:

```
    /* простая strstr: просматривает первый символ */
 /* с помощью strchr */
  char *strstr(const char *s1, const char *s2) {
 int n;
  n = strlen(s2); for (;;) {
 si = strchr(s1, s2[0]); if (s1 == NULL)
 return NULL; if (strncmp(s1, s2, n) == 0)
  return (char *) s1; s1++; }
  }
```

Функция была написана с расчетом на эффективную работу, и действительно, для типичных случаев использования этот код оказывался достаточно быстрым, поскольку в нем используются хорошо оптимизированные библиотечные функции. Сначала вызывается st rch r для поиска следующего вхождения первого символа образца, а затем strncmp — для проверки, совпадают ли остальные символы строки с остальными символами образца. Таким образом, изрядная часть сообщения пропускалась — до первого символа образца, и проверка начиналась только с него. Почему же возникли проблемы с производительностью?

На это есть несколько причин. Во-первых, strncmp получает в качестве аргумента длину образца, которая должна быть вычислена с помощью st rlen. Однако длина образца у нас фиксирована, так что нет необходимости вычислять ее заново для каждого сообщения.

Во-вторых, strncmp содержит достаточно сложный внутренний цикл. В нем не только осуществляется сравнение байтов двух строк, но и производится поиск символа окончания строки \0 в обеих строках, да еще при этом отсчитывается длина строки, переданной в качестве параметра. Поскольку длина всех строк известна заранее (хотя и не в функции st rncmp), в этих сложностях нет необходимости, мы знаем, что подсчеты верны, поэтому проверка на \0 — пустая трата времени.

В-третьих, strchr также сложна, поскольку она должна просматривать символы и при этом отслеживать \0, завершающий строку. При каждом конкретном вызове isspam сообщение фиксировано, поэтому время, использованное на поиск \0, опять же тратится зря, так как мы знаем, где окончится сообщение.

И наконец, даже если решить, что strncrnp, strchr и strlen достаточно эффективны сами по себе, затраты на вызов этих функций сравнимы с затратами на вычисления, которые они осуществляют. Более эффективно выполнять все действия в отдельной аккуратно написанной версии st rst r, а не вызывать другие функции.

Трудности подобного рода — обычный источник проблем с производительностью: функция или интерфейс хорошо работают в обычных ситуациях, но недостаточно производительны для нестандартного случая, а именно этот случай и является основным для решаемой задачи. Существующая версия strstr работала вполне удовлетворительно, когда и образец, и строка были достаточно коротки и менялись при каждом новом вызове, но когда строка оказалась длинной и при этом фиксированной, издержки оказались чрезмерно высоки.

Исходя из вышеизложенных соображений, st rst r была переписана так, чтобы и обход строк образца и сообщения, и поиск совпадений осуществлялись прямо в ней, причем без вызова дополнительных функций. Поведение получившейся реализации было предсказуемо: в некоторых случаях она работала несколько медленнее, но зато в спам-фильтре — гораздо быстрее, и что самое главное, не встретилось случаев, когда бы она работала очень медленно. Для проверки корректности и производительности этой новой реализации был создан набор тестов производительности. В этот набор вошли не только обычные примеры вроде поиска слова в предложении, но и патологические случаи, такие как поиск образца из одной буквы x в строке из тысячи e и образца из тысячи x в строке с одним-един-ственным символом e, а надо сказать, что оба эти случая могли бы стать настоящей проблемой для непродуманной реализации. Подобные экстремальные случаи — ключевая часть оценки производительности.

Наша новая st rst r была добавлена в библиотеку, и в результате спам-фильтр стал работать примерно на 30 % быстрее, чем раньше, — хороший результат для изменения одной функции.

К сожалению, и это было слишком медленно.

Чтобы решить задачу, важно задавать себе правильные вопросы. До сего момента мы искали самый быстрый способ поиска текстового образца в строке. Но на самом деле наша проблема заключается в поиске большого фиксированного набора текстовых образцов в большой строке переменной длины. Если исходить из этого, то наша st rst r не представляется, очевидно, лучшим решением.

Наиболее эффективный способ ускорить программу — применить алгоритм получше. Теперь, когда у нас есть четкое определение проблемы, можно подумать и об алгоритме.

Основной цикл

```
    for (i = 0; i < npat; i++)
 if (strstr(mesg, pat[i]) != NULL)
 return 1;
```

сканирует сообщение в точности npat раз; таким образом, в случае, если совпадений нет, он просматривает каждый байт сообщения npat раз, выполняя strlen(mesg)*npat сравнений.

Разумнее было бы поменять циклы местами, обходя сообщение единожды — во внешнем цикле, а сравнения со всеми образцами осуществлять в параллельном или вложенном цикле:

```
    for (j = 0; mesg[j] != '\0'; j++)
if (совпадение с каким-либо
 образцом начиная с mesg[j])
return 1;
```

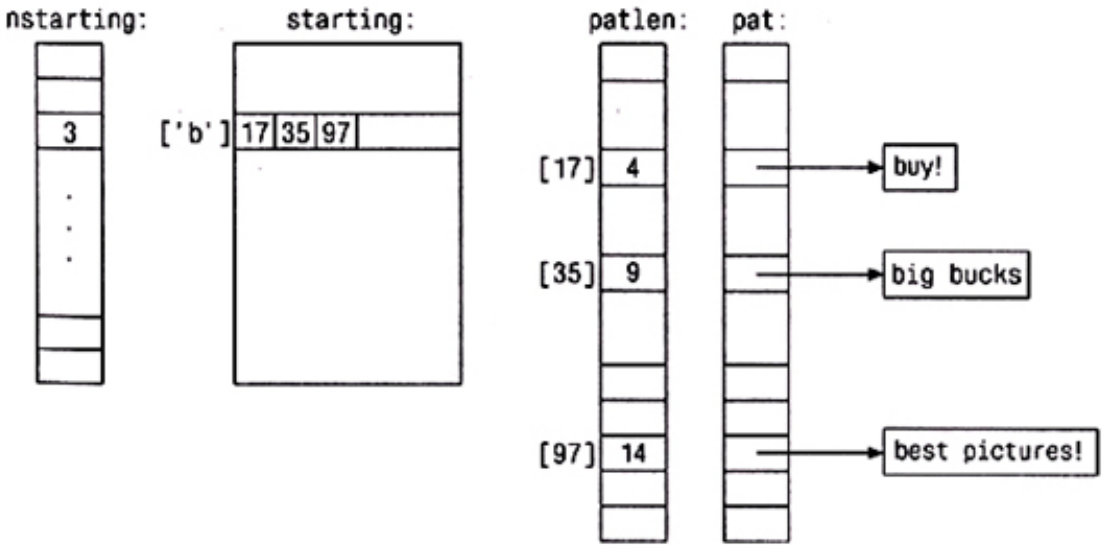
Повышение производительности достигнуто на основании простейшего наблюдения. Для того чтобы выяснить, не совпадает ли какой-нибудь образец с текстом сообщения, начиная с позиции j, нам не надо просматривать все образцы — интересовать нас будут только те, что начинаются с того же символа, что и mesg[j]. В первом приближении, имея 52 буквы верхнего и нижнего регистров, мы можем ожидать выполнения только strlen(mesg)*npat/52 сравнений. Поскольку буквы распределены не одинаково — слова гораздо чаще начинаются с s, чем с x, — мы, конечно, не добьемся увеличения производительности в 52 раза, но все же кое-что у нас получится. Так что фактически мы создали хэш-таблицу, в которой в качестве ключей используются первые буквы образцов.

Благодаря выполнению предварительных действий по созданию таблицы, определяющей, какой образец с какой буквы начинается, код isspam по-прежнему остался достаточно лаконичным:

```
    int patlen[NPAT]; /* длина образца */
int starting[UCHAR_MAX+1][NSTART];
 /* образец с данным началом */
 int nstarting[UCHAR_MAX+1 ];
 /* количество образцов для поиска*/
 /* isspam: проверяет mesg на вхождение любого pat */
 int isspam(char *mesg)
 {
 int i, j, k;
 unsigned char c;
 for (j =0; (c = mesgfj]) != ДО'; j++)
 {
for (i = 0; i < nstarting[c]; i++)
{ $ k = starting[c][i];
 if (memcmp(mesg+j, pat[k], patlen[k]) == 0)
{ printf("spam: совпадает с '%s'\n", pat[k]);
 return 1; } } }
 return 0;
  }
```

Двумерный массив starting[c][ ] хранит для каждого символа с индексы образцов, которые начинаются с этого символа, а его напарник nstarting[c] фиксирует, сколько образцов начинается с этого с. Без этих таблиц внутренний цикл выполнялся бы от 0 до npat, то есть около тысячи раз; в нашем варианте он выполняется от 0 до примерно 20. Наконец, элемент массива patlen[k] содержит вычисленный заранее результат strlen(pat[k]), то есть длину k-го образца.

На приводимом ниже рисунке показаны эти структуры данных для трех образцов, начинающихся с буквы b.



Код для построения этих таблиц весьма прост:

```
    int i;
 unsigned char c;
for (i = 0; i < npat; i++) { c = pat[i][0];
 if (nstarting[c] >= NSTART)
eprintf("слишком много образцов!
(>=%d)"
"начинается на -%c'", NSTART, c);
starting[c][nstarting[c]++] = i;
patlen[i] = strlen(pat[i]);
 }
```

В теперешнем варианте — в зависимости от ввода — спам-фильтр стал работать от пяти до десяти раз быстрее, чем в версии с улучшенной st rst r, и от семи до пятнадцати раз быстрее, чем в исходной реализации. Мы не достигли показателя в 52 раза — отчасти из-за неравномерного распределения букв, отчасти из-за того, что цикл в новом варианте стал более сложным, и отчасти из-за неизбежного выполнения бессмысленных сравнений строк, но все же спам-фильтр перестал быть слабым звеном в обеспечении доставки почты. Проблема производительности решена.

Оставшаяся часть главы будет посвящена технологиям, используемым для выявления проблем производительности, вычленения медленного кода и его ускорения. Однако перед тем, как двигаться дальше, обратимся еще раз к спам-фильтру и посмотрим, чему же он нас научил. Самое главное — убедиться, что производительность имеет критическое значение. Во всех наших действиях не было

бы никакого смысла, если бы спам-фильтр не являлся узким местом почтовой системы. Поскольку мы знали, что проблема заключается именно в нем, мы применили профилирование и другие технологии для изучения его поведения и выяснения главных недостатков. Далее мы убедились, что проблема сформулирована правильно и решать надо именно ее — глобально, а не концентрироваться на улучшении st rst r, на которую падало небезосновательное, однако же, неверное подозрение. Наконец, мы решили эту проблему, применив более удачный алгоритм, и, проверив, выяснили, что скорость действительно возросла. Поскольку она возросла в достаточной степени, мы остановились — зачем заниматься ненужными усовершенствованиями?

Упражнение 7-1

Таблица, которая соотносит отдельный символ с набором образцов, начинающихся с него, стала основой существенного повышения производительности. Напишите версию isspam, которая использует в качестве индекса два символа. Насколько это будет лучше? Это — простейший особый случай структуры данных, которая называется "бор" (trie)'. Большинство подобных структур данных основаны на затратах места ради экономии времени.

Замеры времени и профилирование

Автоматизируйте замеры времени. В большинстве систем существуют команды, позволяющие выяснить, сколько времени работала программа. В Unix такая команда называется time:

```
    % time slowprogram
real 7.0 user 6.2 sys 0.1
 %
slowprogram
7.0 6.2 0.1
```

Эта команда запускает программу и возвращает три числа, означающих время в секундах: время real — физическое время, израсходованное до завершения работы программы; время user — время процессора, потраченное на исполнение самой программы; время sys — время, потраченное на программу операционной системой. Если в вашей системе есть аналогичная команда, используйте ее — числа будут более информативными и надежными, и делать отсчеты будет проще, чем при использовании секундомера. Ведите подробные записи. При работе над программой — внесении модификаций и/или проведении измерений -т у вас накопится огромное количество данных, в которых вам будет трудно разобраться по памяти уже через день-два. (Какая именно версия работает на 20 % быстрее?) Многие технологии, которые мы обсуждали в главе о тестировании, могут быть адаптированы и к измерениям времени и улучшению производительности. Используйте компьютер для запуска набора тестов и измерения времени их работы, а самое главное — используйте регрессивное тестирование, чтобы удостовериться в том, что "улучшения" производительности не нарушили корректности программы.

Если в вашей системе нет команды time или вы хотите замерить время работы отдельно взятой функции, не так трудно создать себе оснастку для таких замеров — по аналогии с тестовой оснасткой. В C и C++ существует стандартная функция clock, которая сообщает, сколько процессорного времени программа использовала до текущего момента. Ее можно вызвать перед интересующей вас функцией и после нее и таким образом вычислить время ее исполнения процессором:

```
    ((include <time. h>
((include <stdio. h>
clock_t before; double elapsed;
before = clockQ; long_running_function();
 elapsed = clock() - before;
printf("Функция использовала %.3f секунд\n",
elapsed/CLOCKS_PER_SEC);
```

Масштабирующий коэффициент CLOCKS_PEfi_SEC характеризует разрешение таймера, возвращаемое clock. Если функция выполняется за доли секунды, запустите ее в цикле, но (если измерения должны быть особо точными) не забудьте компенсировать затраты времени на сам цикл:

```
    before = clockQ;
for (i = 0; i < 1000; i++)
short_running_function();
elapsed = (clock()-before)/(double)i;
```

В Java функции класса Date выдают текущее системное время, которое приблизительно равно времени, использованному процессором:

```
    Date before = new Date();
long_running_function();
Date after = new Date();
 long elapsed =
after.getTime() - before.getTirne();
```

Значение, возвращаемое getTime, измеряется в миллисекундах.

Используйте профилировщик. Не считая такого надежного и достоверного способа, как замер времени исполнения программы, самым важным инструментом для анализа производительности является система для генерации профилей. Профиль (profile) -- это измерение того,,где именно программа "тратит время". Некоторые профили предоставляют список всех функций, количество вызовов каждой из них и долю потребляемого каждой функцией времени от общего времени исполнения программы. Другие показывают, сколько раз было выполнено каждое выражение. Выражения, которые выполняются часто, сильнее влияют на время исполнения, а те, которые не выполняются никогда, скорее всего, являются бесполезным или неадекватно оттестированным кодом.

Профилирование — эффективный способ поиска горячих точек (hot spots), или критических мест программы, то есть функций ши^фрагмен-тов кода, которые расходуют львиную долю всего времени исполнения программы. Однако же интерпретировать результаты профилирования надо с достаточной степенью осторожности. Из-за изощренности компиляторов, сложности организации кэширования и работы с памятью, а также из-за того, что сам процесс профилирования влияет на производительность, статистические показатели в профилях могут быть только приблизительными.

В 1971 году в работе, где впервые был представлен термин "профилирование", Дон Кнут написал, что, "как правило,-менее 4 % программы поглощают более половины от всего времени исполнения". Принимая во внимание это замечание, можно сделать вывод о способах применения профилирования: с его помощью

определяются участки программы, поглощающие большую часть времени работы, в них вносятся улучшения, и процесс повторяется еще раз для поиска новых критических мест. Выясняется, что после одной-двух подобных итераций характерных "горячих точек" не остается.

Профилирование обычно включается специальным флагом или опцией компилятора. Программа выполняется, и после этого показываются результаты работы профилировщика. В Unix таким флагом является, как правило, -p, а профилировщик называется prof:

```
    % ee -p spamtest.c -o spamtest
  % spamtest
  % prof spamtest
```

В приводимой далее таблице показан профиль, сгенерированный рабочей версией спам-фильтра, которую мы создали специально для того, чтобы разобраться в его поведении. В ней используется фиксированное сообщение и фиксированный набор из 217 фраз, который сравнивается с сообщением 10 000 раз. Запуск производился на машине MIPS R10000 250 MHz; использовалась изначальная версия st rst r — с вызовами других стандартных функций. Результаты были переформатированы для более удачного размещения на странице. Обратите внимание на то, что размер ввода (217 фраз) и количество запусков (10 000) можно использовать для проверки правильности работы — эти числа появляются в колонке "calls" (вызовы), в которой показано количество вызовов каждой функции.

```
     12234768552: Total number of instructions executed
  13961810001: Total computed cycles
  55.847: Total computed execution time (sees.)
  1.141: Average cycles / instruction
```

secs	%	cum%	cycles	instructions	calls	function
45.260	81.0	81.0	11314990000	9440110000	48350000	strchr
6.081	10.9	91.9	1520280000	1566460000	46180000	strncmp
2.592	4.6	96.6	648080000	854500000	2170000	strstr
1.825	3.3	99.8	456225559	344882213	2170435	strlen
0.088	0.2	100.0	21950000	28510000	10000	isspam
0.000	0.0	100.0	100025	100028	1	main
0.000	0.0	100.0	53677	70268	219	memccpy
0.000	0.0	100.0	48888	46403	217	memcmp
0.000	0.0	100.0	17989	19894	219	isspam
0.000	0.0	100.0	16798	17547	230	strlen
0.000	0.0	100.0	10305	10900	204	realfree
0.000	0.0	100.0	6293	7161	217	estrdup
0.000	0.0	100.0	6032	8575	231	cleanfree
0.000	0.0	100.0	5932	5729	1	readpat
0.000	0.0	100.0	5899	6339	219	getline
0.000	0.0	100.0	5500	5720	220	malloc

Очевидно, что в затратах времени доминируют две функции: strchr и strncmp, причем обе вызываются из strstr. Итак, Кнут ориентирует нас правильно: небольшая часть программы поглощает большую часть времени ее исполнения. При первом профилировании программы очень часто выясняется, что лидирующие в профиле две-три функции поглощают половину времени и более, как в приведенном случае, и, соответственно, становится сразу же ясно, на чем сконцентрировать внимание.

Концентрируйтесь на критических местах. Переписав st rst r, мы еще раз выполнили профилирование для spamtest и выяснили, что теперь 99.8 % времени тратит собственно strstr, хотя в целом программа и стала работать быстрее. Когда одна функция является таким явным узким местом программы, существует только два способа исправить положение: улучшить саму функцию, применив более эффективный алгоритм, или/ке избавиться от этой функции вообще, переписав всю программу.

Мы выбрали второй способ — переписали программу. Ниже приведены первые несколько строк профиля работы программы sparest, использующей последнюю, наиболее быструю реализацию isspam. При этом общее время работы программы существенно уменьшилось, критической точкой стала функция memcmp, а isspam стала заш'мать более существенную долю от общего времени. Эта версия гораздо сложнее предыдущей, однако эта сложность окупается с лихвой благодаря отказу от использования strlen и strchr в isspam и замене sees % cum% cycles instructions calls function strncmp на memcmp, которая тратит меньше усилий и» обработку каждого байта.

secs	%	cum%	cycles	instructions	calls	function
3.524	56.9	56.9	880890000	1027590000	46180000	memcmp
2.662	43.0	100.0	665550000	902920000	10000	isspam
0.001	0.0	100.0	140304	106043	652	strlen
0.000	0.0	100.0	100025	100028	1	main

Весьма поучительно будет потратить некоторое время на сравнение счетчиков циклов и количества вызовов различных функций в двух профилях. Отметьте, что количество" вызовов strlen упало от нескольких миллионов до 652, а strncmp и memcmp вызываются одинаковое количество раз. Обратите внимание и на то, что isspam, которая сейчас взяла на себя и функцию st rch r, умудряется обойтись гораздо меньшим количеством циклов благодаря тому, что она проверяет на каждом шаге только нужные образцы. Изучение чисел может раскрыть и многие другие детали хода выполнения программы.

"Горячая точка" нередко может быть уничтожена или, по крайней мере, существенно "охлаждена" гораздо более простыми способами, чем мы применили для спам-фильтра. Когда-то давным-давно профилирование Awk показало, что одна функция вызывается примерно миллион раз — в таком цикле:

```
? for (j ='i; j < MAXFLD; j++)

 ? clear(j);
```

Выполнение этого цикла, в котором поля очищались перед считыванИем каждой новой строки ввода, занимало почти 50 % от всего времени исполнения. Константа MAXFLD — максимальное количество полей в строке ввода — была равна 200.

Однако в большинстве случаев использования Awk реальное количество полей не превышало двух-трех. Таким образом, огромное количество времени тратилось на очистку полей, которые никогда и не устанавливались. Замена константы на предыдущее значение максимального количества полей дало выигрыш в скорости 25 %. Все исправления свелись к изменению верхней границы цикла:

```
for (j = i; j < rnaxfld; j++)

clear(j);

maxfld = i;
```

Постройте график. Особенно удачным получается представление замеров производительности в виде графиков. Графики помогут более наглядно представить информацию о влиянии изменения параметров, сравнить алгоритмы и структуры данных, а иногда и указать на неожиданное поведение программы. Графики длин цепочек для нескольких хэш-мультипликаторов в главе 5 явно продемонстрировали преимущества одних мультипликаторов перед другими.

На представленном ниже графике показано влияние размера массива хэш-таблицы на время исполнения С-версии программы ma rkov; в качестве вводимого текста использовались Псалмы (42 685 слов, 22 482 префикса)., Мы поставили два эксперимента. В первой серии запусков программа использовала размеры массивов, которые являлись степенями двойки — от 2 до 16 384; в другой версии использовались размеры, которые являлись максимальным простым числом, меньшим каждой степени двойки. Мы хотели посмотреть, приведет ли использование простых чисел в качестве размеров массивов к ощутимой разнице в производительности.

На графике ясно видно, что время исполнения для такого ввода не зависит от размера таблицы при размерах, больших 10 000 элементов; также нет и существенной разницы между использованием в качестве размеров таблицы степеней двойки или простых чисел.

Упражнение 7-2

Вне зависимости от того, есть на вашей машине команда time или нет, используйте clock или getTime для создания собственного измерителя времени. Сравните его результаты с системным временем. Как влияет на результаты другая активность на машине?

Упражнение 7-3

В первом профиле st rch г вызывалась 48 350 000 раз, а st rncmp — только 46 180 000 раз. Объясните это различие.

Стратегии ускорения

Перед тем как начинать изменять программу, чтобы сделать ее более быстрой, убедитесь, что она действительно работает слишком медленно, а затем используйте инструменты для замера времени и профилировщики для выяснения, на что именно уходит время. После того как вы получили представление о том, что происходит в действительности, можно следовать различным стратегиям ускорения. Мы перечислили некоторые из них, упорядочив их по уменьшению эффективности.

Улучшайте алгоритм и структуру данных. Наиболее важным фактором в уменьшении времени работы программы является выбор алгоритма или структуры данных — между эффективным алгоритмом и неэффективным разница может быть просто огромной. Для нашего спам-фильт-ра мы изменили структуру данных и получили выигрыш в десять раз; еще более внушительным улучшением могло стать применение нового алгоритма, который бы сократил вычисления на порядок, скажем с 0(п2) до 0(п log n). Мы уже обсуждали эту тему в главе 2, так что теперь не будем к ней возвращаться.

Убедитесь в том, что сложность такая, как надо; если она завышена, то именно в ней может крыться источник недостаточной производительности. Этот, линейный на первый взгляд, алгоритм для сканирования строки

```
    ? for (i = 0; i < strlen(s); i++)
? if (s[i] == c)
 ?
```

на самом деле вовсе не линейный, а квадратичный: если s содержит п символов, то цикл выполняется п раз и при каждом исполнении вызов st rlen обходит п символов строки.

Используйте оптимизацию компилятора. Одно улучшение, часто приводящее к достаточно неплохим результатам, вы можете сделать совершенно "бесплатно" — включить всю возможную оптимизацию компилятора. Современные компиляторы справляются с оптимизацией достаточно хорошо, так что необходимости во внесении мелких улучшений вручную, как правило, не возникает.

По умолчанию большинство компиляторов C и C++ не используют все свои возможности по оптимизации, но у них есть опции, позволяющие включить оптимизатор (optimizer). Он не применяется по умолчанию из-за того, что оптимизация обычно конфликтует с отладчиками исходного кода, поэтому включать оптимизатор явным образом можно только после того, как вы будете абсолютно уверены в том, что программа как следует отлажена.

Оптимизация компилятора обычно увеличивает производительность программы в диапазоне от нескольких процентов до двух раз. Иногда, однако, она может даже замедлить программу, так что не забудьте произвести новые замеры времени. Мы сравнили результаты неоптимизированной и оптимизированной компиляции пары версий спам-фильтра. В тестовых примерах для окончательной версии алгоритма сравнения изначальное время исполнения равнялось 8.1 секунды и упало до 5.9 секунды после включения оптимизации, так что улучшение составило более 25 %. В то же время версия, которая использовала исправленную strstr, после включения оптимизации не показала никаких видимых улучшений, поскольку библиотечная функция strstr уже была оптимизирована при инсталляции; оптимизатор обращается только к исходному коду, компилируемому непосредственно в данный момент, но не к системным библиотекам. Правда, некоторые компиляторы имеют глобальный оптимизатор, который в поисках возможных улучшений анализирует всю программу целиком. Если такой компилятор есть в вашей системе, попробуйте использовать его, — может, он ужмет еще несколько циклов.

Надо предупредить, что чем агрессивнее оптимизация компилятора, тем : больше вероятность внесения им ошибок в скомпилированную программу. Поэтому после применения оптимизатора, как, собственно, и после внесения любого изменения, не забудьте запустить набор возвратных тестов.

Тонкая настройка кода. Если объемы данных достаточно существенны, серьезное значение имеет правильный выбор алгоритма. Более того, улучшенный алгоритм будет работать на любых машинах, компиляторах и языках. Но если и при должном алгоритме вопрос быстродействия по-прежнему стоит на повестке дня, то можно попробовать тонко настроить (tune) код — отполировать детали циклов и выражений, чтобы заставить их работать еще быстрее.

Версия isspam, которую мы рассмотрели в конце параграфа 7.1, небыла тонко настроена. Теперь мы покажем, как можно ее улучшить, изменив цикл. Итак, последний вариант выглядел следующим образом:

```
    for (j =0; (c = mesg[j]) != '\O'; j++)
{ for (1=0; i < nstarting[c]; i++)
{ k = starting[c][i];
if (memcmpCmesg+j, pat[k], patlen[k]) == 0)
 { printf("spam: совпадает с '%s'\n", pat[k]);
return 1;
 } } }
```

На этой версии после компиляции с оптимизатором наш набор тестов исполнялся за 6.6 секунды. Внутренний цикл в своем условии содержал индекс массива (nstarting[c]), а его значение фиксировано для каждой итерации внешнего цикла. Мы можем избежать его многократного вычисления, единожды сохранив значение в локальной переменной:

```
    for (j =0;
(c = mesg[j]) != ДО'; j++)
{ n = nstarting[c];
  for (i = 0; i < n; i++)
 { k = starting[c][i];
```

После этого изменения время уменьшилось до 5.9 секунды, то есть примерно на 10 % — типичное значение для ускорения, которого можно достичь с помощью тонкой

настройки. Есть и еще одна переменная, которую мы можем вытащить из тела цикла, — startingfc] также фиксировано. Вроде бы, если мы уберем ее вычисление из цикла, то это улучшит производительность, однако наши тесты не показали никакого изменения. Надо сказать, что это тоже типично при тонкой настройке, — некоторые вещи помогают, а некоторые нет, и это можно определить только замерами времени. Кроме того, результаты могут варьироваться для разных машин и компиляторов.

Есть и еще одно изменение, которое можно внести в спам-фильтр. Внутренний цикл сравнивает образец со строкой, однако алгоритм построен на том, что их первые символы совпадают. Соответственно, мы можем настроить код так, чтобы memcmp начинала сравнение со второго байта. Мы попробовали так сделать, и результатом стал 3-процентньп прирост производительности. Это, конечно, немного, но от нас требовалось изменить всего три строчки кода, что не так уж сложно.

Не оптимизируйте то, что не имеет значения. Иногда настройка не приводит ни к каким результататам только потому, что оказывается направленной на вещи, которые не играют никакой роли. Не забывайте убедиться в том, что оптимизируемый вами код — это именно то место, где теряется драгоценное время. За подлинность следующей истории ручаться не можем, но вам будет полезно ее услышать. Некая старинна машина закрывшейся уже к настоящему времени компании была проанализирована с помощью монитора производительности компонентов Выяснилось, что исполнение одной и той же последовательности из нескольких команд занимало 50 % машинного времени. Инженеры создали специальную команду, выполняющую функции этой последовательности, собрали систему заново и обнаружили, что это не изменило ровным счетом ничего — они оптимизировали цикл ожидания операционной системы.

Сколько времени стоит тратить на увеличение скорости работы программы? Главный критерий — будут ли изменения достаточно результативны, чтобы окупиться. В качестве простейшего принципа можно принять следующее требование: время, потраченное на увеличение производительности программы, не должно быть больше, чем тот выигрыш во времени, который накопится благодаря внесенным изменениям за жизненный цикл программы. Исходя из этого правила, можно сказать, что изменения алгоритма в isspam были оправданы, — они потребовали дня работы, а сэкономили (и продолжают экономить) по нескольку часов каждый день. Удаление индекса массива из внутреннего цикла не сыграло столь глобальной роли, однако все равно имело смысл, поскольку программа эксплуатируется большим количеством пользователей. Оптимизация программ, которые используются коллективно, — вроде спам-фильтра или библиотеки — выгодна практически всегда, а оптимизация тестовых программ не выгодна почти никогда. Программу, которая будет считать что-нибудь в течение года, стоит доводить до максимального совершенства; более того, если через месяц ее работы вы придумаете способ 10-процентного ее ускорения, то эту программу стоит запустить заново.

Программы, продающиеся на рынках с сильной конкуренцией, — игры, компиляторы, текстовые процессоры, электронные таблицы, системы управления базами данных — также относятся к категории программ с окупающимися затратами на улучшения, поскольку коммерческий успех нередко сопутствует тем из них, что быстрее прочих, по крайней мере судя по публикуемым в прессе результатам тестирования.

Важно производить замер времени после внесения каждого изменения, чтобы быть уверенными в том, что ситуация действительно улучшается. Иногда два изменения, каждое из которых порознь улучшает программу, аннулируют эти улучшения при совместном использовании. Кроме того, надо помнить о том, что механизмы,

лежащие в основе измерений времени, могут быть настолько непостоянны в работе, что вынести однозначное решение о пользе изменений весьма проблематично. Даже в однопользовательских системах врем^исполнения может изменяться непредсказуемым образом. Если разброс внутреннего таймера (или, по крайней мере, тех результатов, которые он вам возвращает) составляет 10 %, то изменения, которые увеличивают производительность менее чем на эти 10 %, будет очень трудно отличить от нормальной погрешности результатов.

Настройка кода

Существует большое количество различных способов уменьшить время исполнения, — естественно, после того как критическое место в программе найдено. Ниже мы приводим ряд рекомендаций в этой области, однако надо всегда помнить, что применять что бы то ни было надо с осторожностью и после каждого изменения непременно выполнять возвратные тесты, чтобы быть уверенными в корректности работы кода. Помните, что хорошие компиляторы сами выполняют некоторые из описываемых усовершенствований, и вы можете только запутать программу. Что бы вы ни применяли, после каждого изменения выполняйте замеры времени.

Объединяйте общие выражения. Если некоторое сложное вычисление встречается в вашем коде несколько раз, выполните его единожды и запомните результат. Например, в главе 1 мы показали вам макрос, который вычислял расстояние, дважды подряд вызывая sq rt с одинаковыми значениями; в результате вычисление выглядело так:

```
? sqrt(dx*dx + dy*dy) + ((sqrt(dx*dx + dy*dy) > 0) ? ... )
```

Вычислите корень один раз, а дальше используйте полученное значение. Если вычисление выполняется в цикле, но при этом не зависит ни от каких параметров, изменяющихся в этом цикле, вынесите его из цикла и подставляйте далее его значение:

```
for (i = 0; i < nstarting[cj;
```

можно заменить на

```
n = nstarting[c] ;

for (i = 0; i < n; i++) {
```

Заменяйте дорогостоящие операции на более дешевые. Термин понижение мощности (reduction of strength) относится к такому виду оптимизации, как замена дорогостоящей операции на более дешевую. Когда-то давно этот термин обозначал в основном замену умножения сложениями или сдвигами, правда, сейчас непосредственно на этом вы вряд ли что-то выгадаете. Однако деление и взятие остатка гораздо медленнее умножения, так что код можно несколько улучшить, заменив деление умножением на обратное значение, а взятие остатка при делителе, являющемся степенью двойки, можно заменить использованием битовой маски. Замена индексации массивов указателями в C или C++ может ускорить код, правда, большинство компиляторов выполняет это автоматически. Может быть вполне выгодной и замена вызова функции простым непосредственным вычислением. Расстояние на плоскости определяется по формуле sqrt(dx*dx+dy*dy), так что для выяснения того, какая из двух точек находится дальше, в принципе надо вычислить

два квадратных корня. Но это решение можно принять и сравнивая квадраты расстояний, то есть

```
if (dx1*dx1+dy1*dy1 < dx2*dx2+dy2*dy2)
  ....
```

даст тот же результат, но без вычисления корней.

Другой пример дают сравнения строк с текстовыми образцами, вро-| де нашего спам-фильтра или д rep. Если образец начинается с буквы, то по всему введенному тексту осуществляется быстрый поиск этой буквы, и если вхождений не обнаружено, то более сложные механизмы поиска вообще не вступают в дело.

Избавьтесь от циклов или упростите их. Организация и исполнение цикла требуют некоторых временных затрат, так что, если тело цикла не слишком длинно и выполняется не слишком много раз, может оказаться более эффективным избавиться от цикла вообще и выписать все дей-| ствия последовательно. Таким образом, например,

```
for (i =0;

i < 3; i++) [i] + cfi];
```

станет выглядеть так:

```
  a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + C[2];
```

При этом мы избавляемся от затрат на организацию цикла, особенно от ветвления, которое может существенно замедлять современные процессоры, прерывая поток исполнения.

Если цикл более длинный, тот же тип преобразования можно применить для уменьшения количества операций:

```
for (1=0; i < 3*n; i++) a[i] = b[i] + c[i];
```

станет

```
   for (i = 0; i < 3*n; i += 3)
{ a[i+0] = b[i+0] + c[i+0];
a[i+1] = b[i+1] + c[i+1];
a[i+2] = b[i+2] + c[i+2];
 }
```

Обратите внимание на то, что это можно осуществить только в том случае, если длина в кратное количество раз отличается от размера шага, в противном случае надо обрабатывать дополнительный код на границах, а в таких местах очень часто возникают ошибки, заодно теряется и часть отыгранной эффективности.

Кэшируйте часто используемые значения. Кэшированные значения не надо вычислять заново. Кэширование использует преимущества локолъ-ности (locality) — тенденции программ (да и людей) чаще использовать те элементы, обращение к которым происходило недавно. В вычислительном оборудовании кэши используются очень широко; оснащение компьютера кэшем серьезно ускорило его работу. То же самое относится и к программам. Например, web-браузеры кэшируют страницы и рисунки для того, чтобы как можно меньше прибегать к повторному получению данных из Интернета. Много лет назад мы писали программу просмотра текста для вывода на печать; в этой программе специальные символы, вроде %, приходилось выбирать из специальной таблицы. Измерения показали, что по большей части специальные символы использовались для рисования линий (строк), состоящих из многократного
повторения одного и того же символа. Кэширование только одного последнего использованного символа серьезно улучшило быстродействие программы при обработке типичного ввода.

Лучше всего, если операция кэширования будет невидима снаружи, то есть не будет влиять на программу никаким образом, кроме как ускорять ее исполнение. В случае с нашим предпечатным просмотром интерфейс функции отрисовки не изменился, вызов всегда выглядел как

```
drawchar(c);
```

В исходной версии drawchar осуществлялся вызов show(lookup(c)). В реализации с кэшированием для запоминания последнего вызванного символа и его кода использовались внутренние статические переменные:

```
    f (c != lastc) { /* обновляем кэш */
lasic = c;
 lastcode = lookup(c);
} show(lastcode);
i
```

Напишите специальную функцию захвата памяти (аллокатор). Довольно часто единственной горячей точкой программы с точки зрения быстродействия является захват памяти, который проявляется в огромном количестве обращений к malloc или new. Когда по большей части за-прашиваются блоки одного размера, то можно существенно ускорить программу, заменив вызовы общего аллокатора вызовами специализированной подпрограммы. Такой специальный аллокатор один раз вызывает malloc, получая большой массив элементов, а потом "выдает" их по одному; в результате получается более дешевая операция. Освобождаемые элементы помещаются обратно в список свободных элементов '(free list), благодаря чему их можно сразу же использовать снова.

Если запрашиваемые блоки примерно равны, то можно пожертвовать J местом ради выигрыша во времени и выделять сразу столько памяти, чтобы хватило на самый большой запрос. Например, для работы с короткими строками может оказаться эффективным использование одного и того же размера для всех строк, не превышающих фиксированной длины.

Некоторые алгоритмы могут использовать выделение памяти, осуще- i ствляемое по принципу стека: когда выполняется сразу целая последовательность выделений памяти, а затем она освобождается вся целиком. При таком подходе аллокатор получает сразу большой блок памяти, который использует как стек, по

необходимости заталкивая в него новые элементы и в конце высвобождая их все одной операцией. Некоторые библиотеки С содержат специальную функцию alloca для такого способа выделения памяти, хотя она и не входит в стандарт. В качестве источника памяти она использует локальный стек вызовов; высвобождаются элементы в тот момент, когда заканчивает работу функция, вызвавшая alloca.

Буферизуйте ввод и вывод. При буферизации трансакции объединяются в блоки с тем, чтобы часто исполняемые операции выполнялись с минимально возможными затратами, а очень дорогостоящие операции выполнялись только в случае крайней необходимости. Стоимость операции, таким образом, распределяется между несколькими значениями данных. Например, когда программа на С вызывае-Aprintf, символы сохраняются в буфере, но не передаются операционной системе до тех пор, пока буфер не будет заполнен или не поступит явный запрос на его очистку. В свою очередь, операционная система может задержать запись данных на диск. Помеха в том, что данные становятся видимыми после очистки буферов вывода; и в.худшем случае — информация, содержащаяся в буфере, при аварийной остановке программы пропадает.

Специальные случаи обрабатывайте отдельно. Обрабатывая объекты одинакового размера в отдельном коде, специализированные аллокато-ры уменьшают затраты места и времени и при этом уменьшают степень фрагментации памяти. В графической библиотеке для системы Inferno основная функция d raw была написана насколько возможно просто и непосредственно. После того как она заработала в таком виде, началась оптимизация различных частных случаев, выбранных в результате профилирования (по одному изменению за раз). Большим плюсом являлось то, что оптимизированную версию всегда можно было сравнить с исходной (мы уже говорили о прототипах, — это тот самый случай). В конце концов оптимизации подверглось лишь незначительное число случаев — из-за того, что динамическое распределение вызовов функции рисования очень сильно зависело собственно от вывода символов на экран; для многих случаев просто не имело смысла писать какой-то особо умный код.

Используйте предварительное вычисление результатов. Иногда ускорить работу программы можно предварительно вычислив некоторые значения. Мы проделали это в спам-фильтре, вычислив заранее strlen(pat[i]) и сохранив его в массиве patlen[i]. Если графической системе приходится постоянно вычислять математическую функцию вроде синуса, но только Для какого-то дискретного набора значений, например только для целых значений градусов, то быстрее заранее вычислить таблицу из 360 элементов (или вообще включить ее в программу в виде данных) и обращаться к этой таблице по индексу. Это типичный пример экономии времени за счет места. Существует множество способов замены кода данными или выполнения вычислений при компиляции для сохранения времени, а иногда и места. Например, функции из библиотеки ctype, вроде isdigit, почти всегда реализованы с помощью индексации в таблице битовых флагов, а не посредством вычисления последовательности тестов.

Используйте приближенные значения. Если идеальная аккуратность; вычислений не нужна, лучше использовать типы данных с низкой точностью. На старых или маломощных машинах, а также на машинах, симулирующих плавающую точку программно, вычисления с плавающей точкой, выполняемые с обычной точностью, происходят быстрее, чем с двойной точностью, так что для экономии времени стоит использовать float вместо double. Подобный прием используется в некоторых современных графических редакторах. Стандарт IEEE для плавающей точки требует "чистого выхода за пределы точности" (graceful underflow) no мере приближения

вычислений к нижней границе точности значений, однако такие вычисления достаточно накладны. Для изображений подобные изыски не обязательны, гораздо быстрее отсекать малозначимые части значений; главное, при этом абсолютно ничего не теряется. Это не только экономит время при уменьшении значимости чисел, но и позволяет использовать для вычислений более простые аппаратные средства. Использование целочисленных функций sin и cos — еще один пример использования приближенных вычислений.

Перепишите код на языке более низкого уровня. Как правило, чем | ниже уровень языка, тем более быстрым оказывается код, хотя за это и приходится платить большими затратами на его написание. Так, переписав некоторый критичный фрагмент программы с C++ или Java на C или заменив интерпретируемый скрипт программой на компилируемом языке, вы, скорее всего, добьетесь сокращения времени исполнения.

Порой использование машинно-ориентированного (и, стало быть, машинно-зависимого) кода может дать существенный выигрыш в быстродействии. Это скорее последняя надежда, чем часто применяемый ход, поскольку это прямо-таки уничтожает переносимость и вызывает большие трудности при дальнейшей поддержке и модернизации программы. Почти всегда операции, которые надо выражать на языке ассемблера, — это небольшие функции, которые следует объединить в библиотеку: типичными примерами являются memset и memmove или графические операции. Общий подход должен быть таким: сначала надо написать максимально понятный код на языке высокого уровня и проверить его на корректность, старательно оттестировав; мы проделывали такое для memset в главе 6. Получается переносимая версия, которая будет работать всегда и везде, хотя и довольно медленно. Теперь при переходе в новую среду вы всегда будете иметь версию, которая с гарантией работает, и работает правильно, и, написав версию на ассемблере, сравните ее с работающим прототипом. При возникновении ошибок проверять надо в первую очередь, естественно, новую версию. Вообще, полезно и удобно иметь под рукой реализацию для сравнения.

Упражнение 7-4

Один из способов ускорить функцию типа memset — переписать ее так, чтобы она записывала данные порциями в слово, а не в байт; такая версия почти наверняка будет ближе к аппаратнымт1редставлениям, и затраты на цикл снизятся в четыре или восемь раз. Отрицательной стороной является то, что при этом возникает большое количество вариантов окончаний блоков, если объект приложения функции не выровнен по границе слова (или не кратен слову).

Напишите версию memset, выполняющую подобную оптимизацию. Сравните ее производительность с существующей библиотечной версией и с простейшим вариантом побайтового цикла.

Упражнение 7-5

Напишите функцию выделения памяти smalloc для строк C, которая бы использовала специальный аллокатор для коротких строк, но вызывала непосредственно malloc для больших. Для представления тех и других строк вам придется определить структуру struct. Как вы определите, где переходить от вызова smalloc к malloc?

Эффективное использование памяти

Память — один из самых дорогостоящих компьютерных ресурсов, которого вечно не хватает; огромное количество плохих программ возникло из попыток выжать все, что можно, из того немногого, что имелось в наличии. Небезызвестная "проблема 2000 года" зачастую причисляется к разряду именно таких проблем: когда память была действительно буквально на вес золота, потратить два байта на число 19 считалось непозволительной роскошью. Действительно ли память является главной причиной проблемы или нет, — возможно, все беды возникли из-за перенесения наших бытовых привычек (мы нечасто указываем в документах, к какому веку они относятся, — это и так очевидно) на программы, — но приведенный пример отлично демонстрирует всю опасность недальновидной оптимизации.

В любом случае, времена меняются, теперь и оперативная память, и внешние носители информации стали просто удивительно дешевыми. Таким образом, главный принцип оптимизации использования памяти — тот же, что и в увеличении быстродействия: не беспокойтесь понапрасну.

Однако и сейчас могут возникнуть ситуации, когда вопрос используемого пространства играет существенную роль. Если программе не хватает имеющейся оперативной памяти, то некоторые ее части (страницы) выгружаются на диск, а это отрицательно сказывается на производительности. Часто приходится видеть, насколько расточительно относятся к памяти новые версии программ. Как ни печально, но такова сегодняшняя реальность: обновление программного обеспечения часто влечет за собой покупку дополнительной памяти.

Используйте минимально возможный размер типа данных. Первый шаг к более эффективному использованию места — попытаться с минимальными изменениями лучше распределить существующую память, используя, например, минимальные из возможных типов данных. Эта экономия может означать замену int на short, если данные это позволяют, в частности такое представление используется для координат в системах двумерной графики, поскольку 16 битов, по-видимому, достаточно для любых диапазонов экранных координат. Можно экономить память и заменой double на float, но при этом надо опасаться потери точности, ведь float содержит, как правило, только 6 или 7 десятичных знаков.

В описанных и аналогичных им случаях в программу почти наверняка придется внести и другие соответствующие изменения, самым очевидным из которых будет замена спецификаторов формата в printf и, что особенно важно, в scanf.

Логическим расширением этого подхода является кодирование информации в байте или даже в нескольких битах. Не используйте битовые поля C/C++; они ухудшают переносимость и нередко ведут к неоднозначному и неэффективному коду. Вместо этого поместите нужные операции в функции, которые будут считывать и устанавливать отдельные биты внутри слова или массива слов с помощью сдвигов или битовых масок. Такие функции возвращают группу последовательных битов из середины слова:

```
     /* getbits: получает n
  битов начиная с позиции p */
 /* биты нумеруются с 0
 (наименее значимого) вверх */
 unsigned int getbits(unsigned int x, int p, int n)
 {
```

```
 return (x » (p+1-n)) & "("0 « n);
}
```

Если эти функции окажутся слишком медленными, их можно улучшить с помощью технологий, описанных ранее. В C++ переопределение операторов позволяет сделать доступ к битам похожим на простое индексирование.

Не храните то, что можете без труда вычислить. Надо сказать, что подобные изменения — не самые эффективные; они похожи на тонкую настройку кода. Глобальные улучшения, как правило, достигаются только подбором более удачной структуры данных и соответствующим изменением алгоритма. Приведем пример. Много лет назад к одному из нас обратился за помощью наш коллега: он пытался производить некоторые вычисления над матрицей, которая была так велика, что для того, чтобы она уместилась в памяти, приходилось перезагружать компьютер, сильно урезая операционную систему. Очень хотелось найти какую-нибудь альтернативу, потому что работать в подобном режиме было ужасно неудобно. Мы сразу попросили рассказать, что же представляет собой эта матрица, и выяснили, что она содержала целые числа, большая часть из которых была нулями' . И лишь менее 5 % элементов матрицы были отличны от нуля. Подобная структура тут же навела нас на мысль о представлении, в котором хранились бы только ненулевые элементы матрицы, а доступ к каждому элементу m[i][j] заменялся бы вызовом функции m(i, j). Существует несколько способов хранить данные. Наверное, самый простой из них — это массив указателей, по одному на столбец, каждый из которых указывает на компактный массив номеров строк и соответствующих значений. При такой организации на каждый ненулевой элемент тратится больше места, однако суммарное место все же экономится; доступ к каждому элементу получается более медленным, но это все равно быстрее, чем перезагружать машину. Наш коллега принял наше предложение и был вполне доволен результатом.

Аналогичный подход мы использовали и при решении современного варианта этой проблемы. Некая система проектирования должна была представлять данные о виде поверхности и силе радиосигнала, относящиеся к достаточно большим площадям (от 100 до 200 километров в поперечнике), с разрешением в 100 метров. Сохранение этих данных в одном большом прямоугольном массиве неизбежно привело бы к переполнению памяти на машинах, для которых эта система разрабатывалась, и, соответственно, к активному дисковому обмену. Однако было ясно, что для достаточно больших территорий данные о поверхности Земли и о силе сигнала будут одинаковы, так что для решения проблемы можно было применить иерархическое представление данных, при котором регионы с одинаковыми параметрами хранились в одном элементе.

Вариации на эту тему встречаются на практике достаточно часто, и их проявления различны, но для решения можно применять один и тот же подход: храните одинаковые значения в неявном виде или просто в компактной форме, а время и пространство тратьте на оставшиеся значения. Если одинаковые данные действительно встречаются в вашей задаче достаточно часто, вы сможете добиться впечатляющих результатов.

Программа должна быть организована таким образом, чтобы специфическое представление данных сложного типа было спрятано в классе или в наборе функций, оперирующих внутренними типами данных. Подобная предосторожность даст гарантию, что возможные изменения в представлении данных не затронут остальную часть программы.

Проблемы эффективного использования места иногда проявляются и по отношению к внешнему представлению информации — и к преобразованию, и к хранению. В общем и целом, всегда, когда это возможно, информацию лучше хранить в текстовом виде, а не в каком-то двоичном представлении. Текст хорошо переносится, удобно читается; для его обработки создано великое множество инструментов; двоичное же представление лишено этих преимуществ. Аргументом в защиту двоичных данных служит обычно "скорость", однако к нему стоит относиться с изрядной долей скептицизма, поскольку различия между текстовой и двоичной формой, как правило, не столь глобальны.

Более эффективное распределение пространства зачастую покупается ценой увеличения времени исполнения программы. Некое приложение должно было передавать большой рисунок из одной программы в другую. Рисунки в достаточно простом формате PPM занимали, как правило, около мегабайта, так что мы подумали, что было бы гораздо быстрее кодировать их для передачи в сжатом формате GIF, — тогда файлы получались бы размером примерно 50 килобайтов. Однако кодирование и декодирование GIF занимало столько времени, что это сводило на нет всю экономию от передачи файла меньшего размера, то есть мы ничего не выгадывали. Код для обработки формата GIF состоял примерно из 500 строк, а для PPM — из 10. Таким образом, для облегчения поддержки кода мы отказались от преобразования в GIF, и программа до сих пор работает с форматом PPM. Естественно, если бы речь шла о передаче файлов по сети, то предпочтение было бы отдано формату GIF.

Предварительная оценка

Трудно заранее оценить, насколько быстрой получится программа, и вдвойне трудно оценить скорость специфических конструкций языка или машинных инструкций. Однако совсем не трудно создать модель затрат (cost model) языка или системы, которая даст, по крайней мере, общее представление о том, сколько времени занимает выполнение основных операций.

Одним из способов, часто применяемых к стандартным языкам программирования, является использование программы, которая замеряет скорость исполнения задаваемых последовательностей кода. При этом существует ряд сложностей вроде получения воспроизводимых результатов и отсечения случайных затрат времени, обусловленных конкретной средой, однако главное — наличие возможности получить некоторую информацию, хотя бы с точностью до порядка, и при этом без особых усилий. Например, у нас есть программа для создания модели затрат для C и C++ — она приблизительно оценивает затраты на исполнение отдельных выражений, помещая их в цикл, повторяющийся несколько миллионов раз, и вычисляя затем среднее время. На машине MIPS R10000 250 MHz мы получили следующие данные (время измеряется в наносекундах на операцию):

```
Целочисленные (int) операции
  i1++                                  8
  i1 = i2 + i3                         12
  i1 = i2 - i3                         12
  i1 = i2 * i3                         12
  i1 = i2 / i3                        114
  i1 = i2 % i3                        114

Операции с float
  f1 = f2                               8
  f1 = f2 + f3                         12
  f1 = f2 - f3                         12
  f1 = f2 * f3                         11
  f1 = f2 / f3                         28

Операции с double
  d1 = d2                               8
  d1 = d2 + d3                         12
  d1 = d2 - d3                         12
  d1 = d2 * d3                         11
  d1 = d2 / d3                         58

Преобразования чисел
  i1 = f1                               8
  f1 = i1                               8
```

Целочисленные операции достаточно быстры, за исключением деления и взятия остатка. Операции для чисел с плавающей точкой так же быстры или даже быстрее — сюрприз для тех, кто вырос во времена, когда операции с плавающей точкой были на порядок медленнее, чем целочисленные.

Другие базовые операции также достаточно быстры, включая и вызовь функций (они представлены в последних трех строках данного фрагмента)

```
Целочисленные (int) векторные операции
  v[i] = i                                  49
  v[v[i]] = i                               81
  v[v[v[i]]] = i                           100

Управляющие структуры
  if (i == 5) i1++                           4
  if (i != 5) i1++                          12
  while (i < 0) i1++                         3
  i1 = sum1(i2)                             57
  i1 = sum2(i2, i3)                         58
  i1 = sum3(i2, i3, i4)                     54
```

А вот операции ввода-вывода стоят гораздо дороже шинство других библиотечных функций:

```
  Ввод/вывод
    fputs(s, fp)                           270
    fgets(s, 9, fp)                        222
    fprintf(fp, "%d\n", i)                1820
    fscanf(fp, "%d", &i1)                 2070

  Malloc
    free(malloc(8))                        342
Строковые функции
  strcpy(s, "0123456789")                  157
  i1 = strcmp(s, s)                        176
  i1 = strcmp(s, "a123456789")              64

Преобразования строка/число
  i1 = atoi("12345")                       402
  sscanf("12345", "%d", &i1)              2376
  sprintf(s, "%d", i)                     1492
  f1 = atof("123.45")                     4098
  sscanf("123.45", "%f", &f1)             6438
  sprintf(s, "%6.2f", 123.45)             3902
```

Время, показанное дош f гее и malloc, вряд ли точно соответствует их реальной производительности, поскольку освобождение памяти сразу после выделения — не самое распространенное действие.

И наконец, математические функции:

Математические функции	
i1 = rand()	135
f1 = log(f2)	418
f1 = exp(f2)	462
f1 = sin(f2)	514
f1 = sqrt(f2)	112

Естественно, эти цифры будут разными на разных машинах, однако общие тенденции сохранятся, так что мы можем их использовать для прикидочной оценки тех или иных конструкций, или для сравнения операций ввода-вывода с базовыми операциями, или для принятия решения о том, стоит ли переписывать выражение или использовать встраиваемую (inline) функцию.

Различие при замерах на разных машинах обусловлено разными причинами. Одной из них является уровень оптимизации компилятора. Современные компиляторы способны создать оптимизацию, до которой большинству программистов не додуматься. Более того, современные процессоры настолько сложны, что лишь хороший компилятор способен воспользоваться их возможностями одновременной выборки нескольких команд, конвейерному их исполнению, заблаговременной выборке данных и команд и т.

Еще одна важная причина того, что показатели производительности трудно предсказать заранее, кроется в самой архитектуре компьютеров. Наличие кэш-памяти значительно изменяет скорость исполнения программ, и большая часть аппаратных ухищрений направлена на то, чтобы нивелировать разницу в быстродействии кэш-памяти и обычной памяти. Показатели частоты процессора вроде "400 MHz" дают некоторое представление о быстродействии машины, но не более того: один из наших старых компьютеров Pentium-200 работает гораздо медленнее, чем еще более старый Pentium-100, потому только, что последний имеет гораздо больший кэш второго уровня. А разные поколения процессоров — даже при одинаковом наборе команд — тратят для исполнения одной и той же операции разное количество циклов процессора.

Упражнение 7-6

Создайте набор тестов для оценки времени исполнения основных операций на используемых вами компьютерах и компиляторах. Изучите похожие и различающиеся аспекты производительности.

Упражнение 7-7

Создайте модель затрат для высокоуровневых операций в C++ — таких как создание, копирование и удаление объектов классов, вызовы функций-членов класса, виртуальных функций, встраиваемых (inline) функций, библиотеки lost ream,

STL. Это упражнение не имеет фиксированного завершения, так что сосредоточьтесь на небольшом репрезентативном наборе операций.

Упражнение 7-8

Повторите предыдущее упражнение для Java.

Заключение

Если вы выбрали верный алгоритм, то, как правило, об оптимизации его производительности можно уже особо и не задумываться. Однако, если необходимость в ней все же возникла, основной цикл процесса должен выглядеть так: выполните замеры; займитесь местами, изменения в которых дадут максимальный эффект; проверьте корректность изменений и выполните замеры снова. Останавливайтесь сразу же, как только достигнете приемлемых показателей; не забывайте сохранить первоначальную версию как эталон для проверки новых версий на корректность и быстродействие.

Улучшая скорость программы и ее потребность в памяти, создайте для себя набор эталонных тестов, чтобы было проще оценить и впоследствии отслеживать быстродействие программы. При наличии стандарных тестов используйте и их тоже. Если программа получается достаточно изолированной, самодостаточной, то нередко создают набор "типичных" вариантов ввода — такой подход является основой тестов быстродействия коммерческих и академических систем вроде компиляторов, вычислительных программ и т. п. Так, для Awk создан набор примерно из 20 маленьких программ, которые в совокупности перекрывают большую часть широко используемых конструкций языка. Эти программы применяются для того, чтобы удостовериться, что результаты работы верны и нет сбоев в производительности. Кроме того, у нас есть набор больших стандартных файлов ввода, которые мы также используем для тестирования времени исполнения программ. Полезно придать таким файлам какие-то легко проверяемые свойства, например размер их может быть степенью двух или десяти.

Замеры и шаблонные сравнения можно осуществлять с помощью оснасток того же типа, что $h>i использовали в главе 6 для тестирования: замеры времени запускаются автоматически; в выводимых результатах достаточно информации для того, чтобы они были понятны и воспроизводимы; записи ведутся аккуратно, чтобы можно было отследить основные тенденции и наиболее значительные изменения.

На самом деле создать хорошие тесты-замеры весьма непросто, и об этом прекрасно осведомлены компании, которые специально настраивают свои продукты так, чтобы те показывали как можно более высокие результаты именно на подобных тестах. Так что к их результатам надо относиться с известной долей скептицизма.

Дополнительная литература

Наше обсуждение спам-фильтра основывалось на работе Боба Флан-дрены (Bob Flandrena) и Кена Томпсона (Ken Thompson). Их фильтр включает в себя регулярные выражения для выполнения более осмысленных проверок и автоматической классификации сообщений ("точно спам", "возможный спам", "не спам") в соответствии со строками, которые были обнаружены.

Профилировочная статья Кнута "An Empirical Study of FORTRAN Programs" появилась в журнале Software — Practice and Experience (1, 2, p. 105-133, 1971). Ее

основой стал статистический анализ ряда программ, раскопанных в мусорных корзинах и общедоступных каталогах машин компьютерного центра.

Ион Бентли в своих книгах "Программирование на Pearls" и "Еще программирование на Pearls" (Jon Bentley. Programming Pearls. Addison-Wesley, 1986;Jon Bentley.More Programming Pearls. Addison-Wesley, 1988) приводит ряд хороших примеров улучшений программ, основанных на изменении алгоритма или настройке кода; очень неплохо описано создание оснасток для улучшения производительности и использование профилирования.

Книга Рика Буфа "Внутренние циклы" (Rick Booth. Inner Loops. Addison-Wesley, 1997) — хорошее руководство по настройке программ для PC. Правда, процессоры эволюционируют так быстро, что специфические детали стремительно устаревают.

Серия книг Джона Хеннеси и Дэвида Паттерсона по архитектуре компьютеров (например: John Hennessy, David Patterson. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 1997) содер жит вдумчивые рассуждения о вопросах производительности современных компьютеров.

Переносимость

- Язык
- Заголовочные файлы и библиотеки
- Организация программы
- Изоляция
- Обмен данными
- Порядок байтов
- Переносимость и внесение усовершенствований
- Интернационализация
- Заключение
- Дополнительная литература

> Наконец, стандартизация, так же как и соглашения, может служить еще одной демонстрацией строгого порядка. Но, в отличиеЪт соглашений, она принимается в современной архитектуре как продукт, хоть и украшающий нашу технологию, но опасный из-за ее потенциального доминирования и грубости.
>
> Роберт Ветури Сложности и противоречия в архитектуре

Написать корректную и эффективную программу трудно. Поэтому если программа уже работает в одной среде, то вам, скорее всего, не захочется повторять пройденный путь ее создания при переходе на другой компилятор, процессор или операционную систему. В идеале программа не должна требовать внесения никаких изменений.

Этот идеал называется переносимостью (portability). На практике термин "переносимость" нередко относят к более слабому свойству: программу проще видоизменить при переносе в другую среду исполнения, чем написать заново. Чем меньше изменений надо внести, тем выше переносимость программы.

Вы можете спросить, почему вообще зашел разговор о переносимости: если программа будет исполняться в какой-то конкретной среде, зачем тратить время на то, чтобы обеспечивать ей более широкую область применения? Ну, во-первых, каждая хорошая программа почти по определению начинает с какого-то момента применяться в непредсказуемых местах. Создание продукта с функциональностью более общей, чем изначальная спецификация, приведет к тому, что меньше сил придется тратить на поддержку продукта на протяжении его жизненного цикла. Во-вторых, среда исполнения меняется. При замене компилятора, операционной системы, железа меняются многие параметры окружения программы. Чем меньше программа зависит от специальных возможностей, тем меньше вероятность возникновения сбоев при изменении условий и тем проще программа адаптируется к этим условиям. И наконец, последнее и самое важное обстоятельство: переносимая программа всегда лучше написана. Усилия, затраченные на обеспечение переносимоести программы, сказываются на всех ее аспектах; она оказывается лучше спроектированной, аккуратнее написанной и тщательнее протестированной. Технологии программирования переносимых программ непосредственно привязаны к технологиям хорошего программирования вообще.

Естественно, степень переносимости должна определяться исходя из реальных условий. Нет такой вещи, как абсолютно переносимая программа, — разве только программа, опробованная не во всех средах. Однако мы можем сделать переносимость одной из своих главных целей, стараясь создать программу, которая бы выполнялась без изменений практически везде. Даже если эта цель не будет достигнута в полном объеме, время, потраченное на ее достижение, с лихвой окупится, если программу придется переделывать под другие условия.

Мы хотим посоветовать следующее: старайтесь писать программы, которые бы работали при всех комбинациях различных стандартов, интерфейсов и сред, в принципе подходящих для ее исполнения. Не старайтесь исправить каждую ошибку переносимости, добавляя дополнительный код, наоборот, адаптируйте программу для работы при новых ограничениях. Используйте абстракцию и инкапсуляцию, чтобы ограничить и контролировать те непереносимые фрагменты кода, без которых не обойтись. Если ваш код сможет работать при всех ограничениях, а системные различия в нем будут локализованы, он станет еще и более понятным.

Язык

Придерживайтесь стандарта. Первое, что необходимо для создания переносимого кода, — это, конечно, использование языка высокого уровня, причем его стандарта, если таковой определен. Двоичные коды переносятся плохо, исходный код — несколько лучше. Однако и для него трудно, даже для стандартных языков, описать точно, каким именно образом компилятор преобразует его в машинные инструкции. Очень немногие из распространенных языков представлены единственной реализацией: обычно имеется множество производителей различных версий для различных операционных систем, которые к тому же со временем изменяются. Каждая версия будет обрабатывать ваш код по-своему.

Почему стандарт не является строгим описанием? Иногда стандарт неполон и не описывает отдельных специфических случаев. Иногда он намеренно неконкретен: например, тип cha r в C и C++ может иметь знак, а может и не иметь; он даже не обязательно должен быть 8-битовым. Подобные детали оставлены на усмотрение создателя компилятора; в этом есть свои плюсы: стимулируется появление новых эффективных реализаций; снимаются ограничения, накладываемые на железо, но жизнь программиста, конечно, несколько усложняется. Вообще, степень проработанности стандарта зависит от многих глобальных причин. И наконец, нельзя забывать, что языки достаточно запутанны, а компиляторы весьма сложны; в них могут быть неправильности интерпретации и ошибки реализации.

Иногда же языки вообще не стандартизованы. Официальный стандарт ANSI/ISO C был принят в 1988 году, а стандарт ISO C++ ратифицирован только в 1998-м. На момент написания этой книги не все из распространенных компиляторов поддерживают это официальное описание. Язык Java сравнительно молод, и его стандарт можно ждать только через несколько лет. Вообще стандарт языка разрабатывается только после того, как создаются несколько конфликтующих* версий, которые надо унифицировать, а сам язык получает достаточно широкое распространение, оправдывающее затраты на стандартизацию. А между тем по-прежнему надо писать программы и поддерживать в этих программах различные среды исполнения.

Итак, несмотря на то что пщ знакомстве со справочными руководствами и стандартами складывается впечатление жесткой спецификации языка, они никогда не описывают язык полностью, и различные версии компиляторов могут создавать

работоспособные, но несовместимые друг с другом реализации. Иногда возникают даже ошибки. Вот довольно характерный пример: подобные внешние описания недопустимы в C и C++

```
? *x[] = {"abc"};
```

Проверив с десяток компиляторов, мы выяснили, что лишь несколько из них корректно определяют пропущенный определитель типа — слово char для x. Значительная часть выдает предупреждение о несовместимости типов (очевидно, те, что используют старое описание языка: они неверно полагают x массивом указателей на int), а еще пара компилировала этот недопустимый код, не сделав ни вздоха.

Следуйте основному руслу. Неспособность некоторых компиляторов выявить ошибку в приведенном примере весьма прискорбна, но зато мы теперь сможем осветить важный аспект переносимости. У каждого языка есть свои непроработанные моменты, точки зрения на которые разнятся (например, битовые поля в C и C++), и благоразумие подсказывает избегать их использования. Стоит использовать только те возможности, описание которых недвусмысленно и хорошо понятно. Очевидно, что именно такие возможности, скорее всего, будут широко доступны и реализованы везде одинаковым образом. Мы называем их основным руслом (mainstream) языка.

Трудно однозначно определить, какие именно конструкции входят в это основное русло, но зато те, что в него не входят, определить просто. Совершенно новые возможности, такие как complex или комментарии // в C, или возможности, специфичные для конкретной архитектуры, вроде ключевых слов near и far, обязательно создадут вам проблемы. Если что-то в языке настолько необычно или непонятно, что для того, чтобы разобраться, вам приходится консультироваться с "языковым правоведом" — экспертом по чтению его описаний, не используйте такую возможность вовсе.

В своем обсуждении мы сконцентрируем основное внимание на C и C++. широко распространенных и универсальных языках. Стандарт C существует уже более десятка лет, и язык очень стабилен, правда, разрабатывается новый стандарт, так что впереди очередные подвижки. А вот стандарт C++ появился совсем недавно, и потому еще не все реализации этого языка приведены с ним в соответствие.

Что можно считать основным руслом в C? Этот термин обычно относят к установившемуся стилю использования языка, но иногда лучше принимать во внимание грядущие изменения. Например, первоначальная версия C не требовала создания прототипов функций. Описание функции sq rt выглядело при этом так:

```
? double sqrt();
```

что определяло тип возвращаемого значения, но не параметров. В ANSI C были добавлены прототипы функций, в которых определялось уже все:

```
do.uble sqrt(double);
```

Компиляторы ANSI C должны воспринимать и прежний синтаксис, но мы настоятельно рекомендуем вам забыть об этом и всегда писать прототипы для всех функций. Это сделает код более безопасным, вызовы функций будут полностью проверены на совместимость типов, и при изменении интерфейса компилятор это отловит. Если в коде употребляется вызов

```
func(7, PI);
```

а func не имеет прототипа, компилятор не сможет проверить корректность такого вызова. Если библиотека впоследствии изменится так, что у func станет три аргумента, компилятор не сможет предупредить о необходимости внесения изменений в программу, потому что старый синтаксис не предусматривает проверки типов аргументов функций.

C++ — более мощный и обширный язык, и стандарт его появился лишь недавно, поэтому говорить об основном русле применительно к нему несколько сложнее. Например, нам представляется очевидным, что STL войдет в это основное русло, что происходит, однако, не мгновенно, и поэтому некоторые существующие реализации языка поддерживают STL не в полной мере.

Избегайте неоднозначных конструкций языка. Как мы уже говорили, некоторые вещи в стандартах умышленно оставлены неопределенными для того, чтобы предоставить создателям компиляторов большую свободу для маневра. Список таких неопределенностей обескураживающе велик.

Размеры типов данных. Размеры основных типов данных в C и C++ не определены. Не существует никаких гарантированных свойств, кроме общих правил, гласящих, что

```
sizeof(char) <= sizeof (short) <= sizeof(int)

<= sizeof(long) sizeof(float)

<= sizeof(double) %
```

а также. что char должен иметь как минимум 8 битов, short и int — как минимум 16, а long — по крайней мере 32. Не требуется даже, чтобы значение указателя умещалось в inj.

Проверить, какие значения использует конкретный компилятор, достаточно просто:

```
/* sizeof: выводит размеры базовых типов
*/ int main(void)
{
printf("char %d, short %d, int %d, long %d,",.
sizeof(char), sizeof(short),
sizeof(int), sizeof(long));
printf(" float %d, double %d, void* %d\n",
sizeof(float), sizeof(double), sizeof(void *));
return 0;
}
```

Результат будет одинаковым для большинства распространенных машин:

```
char 1, short 2, int 4, long 4, float 4, double 8, void* 4
```

однако возможны и другие значения. Некоторые 64-битовые машины покажут такие значения:

char 1, short 2, int 4, long 8, float 4, double 8, void* 8

а ранние компиляторы под PC показали бы такое:

char 1, short 2, int 2, long 4, float 4, double 8, void* 2

Во времена появления PC аппаратура поддерживала несколько видов указателей. Для того чтобы справиться с такой неразберихой, были придуманы модификаторы указателей far и near, ни один из которых не входит в стандарт, но до сих пор эти ключевые слова-призраки появляются во многих компиляторах. Если ваш компилятор может менять размеры базовых типов или если вы имеете доступ к машинам, поддерживающим другие размеры, постарайтесь скомпилировать и оттестировать вашу программу при таких новых для нее условиях.

Стандартный заголовочный файл stddef. h определяет ряд типов, которые могут помочь с переносимостью. Наиболее часто используемый из них — size_t, который представляет собой тип беззнакового целого, возвращаемого оператором sizeof. Значения этого типа возвращаются функциями типа strlen и во многих функциях, включая mall oc, используются в качестве аргументов.

Наученная чужим горьким опытом, Java четко определяет размеры всех своих базовых типов: byte — 8 битов, char и short — 16, int — 32 и long — 64 бита.

Мы не будем рассматривать набор проблем, связанных с вычислениями с плавающей точкой, потому что разговор об этом достоин отдельной книги. Скажем только, что большинство современных машин поддерживают стандарт IEEE на аппаратуру для плавающей точки, и, следовательно, для них свойства арифметики с плавающей точкой определены достаточно четко.

Порядок вычислений. В C и C++ порядок вычислений операндов выражений, побочных эффектов и аргументов функций не определен. Например, в присваивании

? n = (getcharQ « 8) getchar();

второй getcha r может быть вызван первым: порядок, в котором выражение записано, не обязательно соответствует порядку, в котором оно исполняется. В выражении

? ptr[count] = name[++count];

значение count может быть увеличено как до, так и после использования его в качестве индекса pt r, а в выражении

? . printf("%c %c\n", getchar(), getchar());

первый введенный символ может быть распечатан на втором месте, а не на первом. В выражении

? printf("%f %s\n", logC-1.23), strerror(errno));

значение errno может оказаться вычисленным до вызова log.

Для некоторых выражений существуют четкие правила вычисления. По определению все побочные эффекты и вызовы функций должны быть завершены до

ближайшей точки с запятой или при вызове функции. Операторы && и | | выполняются слева направо и только до тех пор, пока это требуется для определения их значения (включая побочные эффекты). В операторе ?: сначала вычисляется условие, включая возможные побочные эффекты, и только после этого вычисляется одно из двух завершаюшжс оператор выражений.

В Java порядок вычислений описан более жестко. В нем предусмотрено, что выражения, включая побочные выражения, вычисляются слева направо; правда, в одном авторитетном руководстве дан совет не писать кода, который бы "критично" зависел от этого порядка. Прислушаться к этому совету просто необходимо при создании программ, у которых есть хотя бы призрачный шанс конвертироваться в С или С++: как мы уже сказали, там нет никаких гарантий соблюдения такого же порядка. Конвертирование из одного языка в другой — экстремальный, но иногда весьма полезный тест на переносимость программы.

Наличие знака у char. В С и С++ не определено, является ли char знаковым или беззнаковым типом данных. Это может привести к проблемам при использовании комбинаций char и int, как, например, в приводимом коде, где вызывается функция getchar(), возвращающая значение типа int: Если вы напишете

```
?char c; /* должно было быть int */

? c = getchar();
```

то значение с будет в диапазоне от 0 до 255, если char — беззнаковый тип, и в диапазоне от -128 до 127, если char — знаковый тип; речь идет о практически стандартной конфигурации с 8-битовыми символами на машинах с дополнительным до двух кодом. Это имеет особый смысл, если символ должен использоваться как индекс массива или для проверки на EOF, который в stdio обычно представляется значением -1. Например, представим, что мы разработали этот код из параграфа 6.1 после исправления некоторых граничных условий в начальной версии. Сравнение s[i] == EOF никогда не будет истиной, если char — беззнаковый тип:

```
? int i;
? char s[MAX];
?
? for (1=0; i < MAX-1; i++)
? if ((s[i] = getcharO) == An' ||
s[i] == EOF)
? break;
? s[i] = '\O':
```

Когда getchar возвратит EOF, в s[i] будет сохранено значение 255 (OxFF, результат преобразования -1 в unsigned char). Если s[i] беззнаковое, то при сравнении с EOF его значение останется 255, и, следовательно, проверка не пройдет.

Однако, даже если char является знаковым типом, код все равно некорректен. В этом случае сравнение с EOF будет проходить нормально, но при вводе вполне допустимого значения OxFF оно будет воспринято как EOF и цикл будет прерван. Так что вне зависимости от того, знаковый или беззнаковый у вас char, хранить значение, возвращаемое getcha r, вы должны в int, и тогда проверка на конец файла будет осуществляться нормально. Вот как должен выглядеть наш цикл в переносимом виде:

```
    int c, i; char s[MAX];
for (1=0; i < MAX-1; i++) {
 if ((c = getcharO) =='\n'
|| c == EOF) break;
s[i] = c; }

    s[i] = '\O';
```

В языке Java вообще нет спецификатора unsigned; порядковые типы данных являются знаковыми, а тип char (16-битовый) — беззнаковым.

Арифметический или логический сдвиг. Сдвиг вправо знаковых величин с помощью оператора » может быть арифметическим (при сдвиге распространяется копия знакового бита) или логическим (при сдвиге освободившиеся биты заполняются нулями). И здесь Java, наученная горьким опытом С и С++, резервирует » для арифметического сдвига вправо и предоставляет отдельный оператор >» для логического сдвига вправо.

Порядок байтов. Порядок байтов внутри short, int и long не определен; байт с младшим адресом может быть как наиболее значимым, так и наименее значимым. Этот вопрос зависит от аппаратуры, и мы подробно обсудим его несколько ниже в этой главе.

Выравнивание членов структуры или класса. Расположение элементов внутри структур, классов и объединений (union) не определено, утверждается лишь, что члены располагаются в порядке объявления. Например, в структуре

```
struct X {
char c;
int i;
};
```

адрес iможет находиться на расстоянии 2, 4 или 8 байтов от начала структуры. Некоторые (немногие) машины позволяют целым храниться на нечетных границах, но большинство требует, чтобы п-байтовые элементарные типы данных хранились на и-байтовых границах, чтобы, например, double, которые имеют, как правило, длину в 8 байтов, хранились по адресам, кратным 8. В дополнение к этому создатели компиляторов могут вносить и свои ограничения — такие как принудительное выравнивание для повышения производительности.

Никогда не рассчитывайте на то, что элементы структуры занимают смежные области памяти. Ограничения на выравнивание вызывают появление "дыр" в структурах — так, st ruct X всегда будет содержать по крайней мере один байт неиспользуемого пространства. Из-за этих дыр размер структуры может быть больше, чем сумма размеров ее членов, причем этот размер может быть разным на разных машинах. Если вы хотите зарезервировать память под структуру, всегда запрашивайте sizeof (struct X) байтов, ноне sizeof (char) + sizeof(int).

Битовые поля. Битовые поля настолько зависят от конкретных машин, что никому не следует их использовать.

Все перечисленные опасные места можно миновать, следуя нескольким правилам. Не используйте побочные эффекты нигде, кроме как в идиоматических конструкциях типа

```
    a[i++] = 0;
c = *p++;
*s++ = *t++;
```

Не сравнивайте char с EOF. Всегда используйте sizeof для вычисления размера типов и объектов. Никогда не сдвигайте вправо знаковые значения. Убедитесь, что тип данных достаточно велик для диапазона значений, которые вы собираетесь в нем хранить.

Попробуйте несколько компиляторов. Иногда вам может показаться, что вы решили все проблемы с переносимостью, однако компиляторы в состоянии увидеть проблемы, который не вы заметили, и вообще, раз-, ные компиляторы воспринимают вашу программу по-разному, и этим; можно воспользоваться. Включите все предупреждения компилятора. Попробуйте использовать разные компиляторы на одной машине и на разных машинах. Попытайтесь компилировать вашу С-программу на компиляторе С++.

Поскольку язык, воспринимаемый различными компиляторами, может несколько отличаться от стандарта, тот факт, что ваша программа компилируется одним компилятором, не дает гарантии даже того, чтс она корректна синтаксически. А вот если несколько компиляторов принимают ваш код, значит, все не так плохо. Мы компилировали каждув программу, приведенную в книге, на трех компиляторах С для трех различных операционных систем (Unix, Plan 9, Windows) и на паре компиляторов С++. При таком подходе были найдены десятки ошибок переносимости — никакое самое пристальное изучение программ человеком не смогло бы найти их все. Исправлялись же все ошибки тривиально.

Естественно, сами компиляторы тоже вызывают проблемы с переносимостью из-за разного толкования не описанных в стандарте случаев. Однако наш подход — писать программы, избегая применения деталей, которые могут варьироваться, позволяет надеяться на создание программ, работающих вне зависимости от внешних условий.

Заголовочные файлы и библиотеки

Заголовочные файлы и библиотеки предоставляют возможности, расширяющие базовый язык. Например, ввод и вывод осуществляются с помощью библиотек stdio в С, lost ream в С + + и Java, io в Java. Строго говоря, эти элементы не являются частью языка, но они определен! вместе с языком и представляют собой составную часть любой среды, поддерживающей этот язык. Однако, поскольку библиотеки покрывают широкий спектр возможностей и нередко имеют дело со специфическими вопросами устройства операционных систем, в их использовании может крыться причина плохой переносимости программы.

Используйте стандартные библиотеки. Здесь, как и при обсуждении самого языка, применимо то же самое общее правило: придерживайтесь стандарта, отдавая предпочтение старым, устоявшимся компонентам. В С определена стандартная библиотека функций ввода-вывода, операций со строками, проверок символов, выделения памяти под данные и ряда других задач. Если вы ограничите взаимодействие своей программы с операционной системой использованием этих функций, с хорошей долей уверенности можно считать, что программа будет вести себя одинаково при переходе от одной операционной системы к другой. Однако и здесь надо соблюдать осторожность: существуют различные реализации библиотек, и некоторые из них имеют отклонения от стандарта.

В ANSI C не определена функция копирования строк st rdup, хотя она имеется в большинстве сред программирования — даже в тех, что декларируют строгую приверженность стандарту. Может показаться заманчивым использовать данную функцию, но делать этого не следует: компилятор не предупредит вас, что функция не стандартная, а в дальнейшем программу будет не перенести в среду, этой функции не имеющую. Проблемы подобного рода — основной источник головной боли при использовании библиотек, и единственное решение — придерживаться стандарта и тестировать свою программу на возможно большем количестве конфигураций.

Заголовочные файлы и описания пакетов описывают интерфейс со стандартными функциями. Один из недостатков многих заголовочных файлов состоит в том, что в них приводятся описания сразу для нескольких языков. Нередко можно встретить один файл вроде stdio. h с описаниями одновременно для старого (до стандарта ANSI) C, ANSI C и даже C++ компиляторов. Такой файл получается очень громоздким — в нем много директив условной компиляции вроде #if и ffif def. Язык препроцессора не слишком гибок, поэтому такие файлы получаются довольно сложными для восприятия; иногда в них даже содержатся ошибки.

Ниже приведен фрагмент заголовочного файла одной из систем, причем он еще гораздо лучше многих, по крайней мере нормально отформатирован:

```
    ? # ifdef _OLD_C
? extern int fread();
? extern int fwrite();
?  # else
?       # if defined(__STDC__) || defined(__cplusplus)
?     extern size_t fread(void*, size_t, size_t, FILE*);
?     extern size_t fwrite(const void*, size_t, size__t, FILE*);
?       # else /* not __STDC__ || __cplusplus */
?     extern size_t fread();
?     extern size_t fwriteO;
?       # endif /* else not „_STDC__ | _cplusplus */
?       #endif
```

Даже из этого сравнительно простого примера видно, что заголовочные файлы и программы, структурированные подобным образом, получаются довольно запутанными и сопровождать их достаточно сложно. Возможно, проще использовать отдельный заголовочный файл для каждого компилятора или среды. При этом придется сопровождать множество отдельных файлов, но каждый из них будет предназначен только для использования в конкретной конфигурации, что уменьшит вероятность появления ошибок вроде включения функции st rdup в среду, строго поддерживающую стандарт ANSI C.

Заголовочные файлы также иногда "засоряют" пространство имен, определяя функции с именами, уже использующимися в программе. Например, наша функция оповещения об ошибках wep rintf изначально называлась wp rintf, однако мы выяснили, что в некоторых средах функция с таким именем определена в stdio. h (можно сказать, что сделано это в преддверии нового стандарта C). Для того чтобы скомпилировать программу в этих средах и защитить себя в будущем, нам пришлось изменить название своей функции. Если бы проблема состояла в некорректном компиляторе, а не в ожидаемом изменении спецификации, как в нашем случае, то ее можно было бы решить, переопределяя имя при подключении заголовочного файла:

```
    ? /* в stdio.h иногда входит wprintf, переопределим его: */
 ? tfdefine wprintf stdio_w'printf
 ? Jtinclude <stdio. h>
  ? #undef wprintf
 ? /* далее можно использовать нашу wprintf() ... */
```

Этот фрагмент изменяет все появления wprintf в заголовочном файле на stdio_wp rintf, так что теперь они не повлияют на нашу версию. Теперь мы можем использовать нашу wprintf, не изменив ее имени, правда, при этом неизбежно появится некая путаница, а также риск, что подключенная библиотека будет вызывать нашу wprintf, подразумевая обращение к своей версии. Для одной функции проблемы, может, и невелики, но уже для нескольких лучше придумать более радикальное решение. Всегда комментируйте назначение конструкции; без крайней необходимости не ухудшайте ее добавлением условной компиляции. Если в некоторых средах определена wprintf, то стоит считать, что она определена во всех; тогда единственный разумный выход — переименовать ее, избавившись при этом от выражения tfifdef. Нередко проще не превозмогать трудность, а подстраиваться под нее; да это и безопаснее, вот почему мы решили переименовать свою функцию в weprintf.

Даже если вы следуете всем правилам и неясностей со средой не возникает, все равно вполне возможно появление ошибок. Так, можно ошибиться, предположив, что какая-нибудь ваша излюбленная возможность одинакова во всех системах. К примеру, ANSI C определяет шесть сигналов, которые можно поймать с помощью signal, в стандарте POSIX их определено 19, а большинство"систем Unix поддерживает 32 и более. Если вы хотите использовать сигнал, отличный от описанного в ANSI C, вам придется выбирать между функциональностью и переносимостью, так что сами решайте *ITO для вас важнее.

Существует болыпсте количество других стандартов, не являющихся частью определения языка: среди них можно назвать интерфейсы операционных систем и сетей, графические интерфейсы и тому подобные вещи. Некоторые стандарты распространяются на несколько систем — например POSIX; другие определены исключительно для одной системы, например различные API Microsoft Windows. Здесь можно еще раз повторить наши главные советы: ваша программа станет более переносимой, если вы выберете самые распространенные и устоявшиеся стандарты и будете пользоваться самыми важными и общепринятыми их свойствами.

Организация программы

Существуют два основных подхода к переносимости, которые мы назовем объединением и пересечением. Объединение подразумевает использование лучших возможностей каждой конкретной системы; компиляция и установка при этом зависят от условий конкретной среды. Результирующий код обрабатывает объединегше всех сценариев, используя преимущества каждой системы. Недостатки этого подхода включают большой размер кода и сложность установки, а также сложность кода, напичканного условными компиляциями.

Используйте только то, что доступно везде. Мы рекомендуем придерживаться другого подхода, пересечения: использовать только конструкции, имеющиеся во всех системах, для которых делается программа. Этот подход также не лишен

недостатков. Первый его недостаток состоит в том, что требование универсальной применимости может ограничить либо круг систем, предназначенных для использования, либо перечень приемлемых языковых конструкций. Второй недостаток — в некоторых системах производительность программ может оказаться далекой от совершенства.

Для сравнения двух описанных подходов рассмотрим пару примеров, сделанных по принципу объединения, и обдумаем, как они будут выглядеть для пересечения. Как вы увидите, код, основанный на объединении, уже проектируется как непереносимый, хотя хорошая переносимость и является вроде бы основной его целью, а код пересечения получается не только переносимым, но еще и более простым.

Следующий небольшой фрагмент пытается справиться с системой, в которой по некоторым причинам нет стандартного заголовочного файла stdlib. h:

```
    ? #if defined (STDC_HEADERS) || defined (_LIBC)
? #include <stdlib.h>
? #else
? extern void *malloc(unsigned int);
 ? extern void *realloc(void *, unsigned int);
 ? #endif
```

Защитный стиль приемлем, если он применяется время от времени,; а не всегда. Возникает резонный вопрос: а для скольких еще функций из stdlib. h придется писать аналогичный код? В частности, если вы собираетесь использовать mall oc и real loc, то явно потребуется еще и free. А что, если тип unsigned int нетождественен size_t — правильному типу аргумента для malloc и realloc? Более того, откуда мы знаем, что STOC_HEADERS и _LIBC определены, и определены корректно? Можем ли мы быть уверенными в том, что не существует другого имени, которое потребует замены для другой среды? Любой условный код вроде этого неполон, а значит — непереносим, поскольку рано или поздно встретится система, не удовлетворяющая его условию, и тогда придется редактировать #ifdef. Если нам удастся решить задачу без помощи условной компиляции, мы избавимся и от проблем, связанных с дальнейшим поддержанием этого кода.

Итак, проблема, которая решается в рассмотренном примере, существует в реальности. Так как же нам решить ее раз и навсегда? На самом деле нам просто надо предположить, что стандартные заголовочные файлы присутствуют во всех системах всегда; если одного из них нет, то это уже не наши проблемы. Но мы можем решить и их; для данного случая достаточно вместе с программой поставить и заголовочный файл, который определяет malloc, realloc и free в точности так, как этого требует стандарт ANSI C. Такой файл всегда может быть включен полностью вместо "заплаток", и мы будем уверены, что нужный интерфейс обеспечен.

Избегайте условной компиляции. Условной компиляцией с помощью ttifdef и подобных ей директив препроцессора трудно управлять, поскольку информация оказывается рассеянной по всему коду:

```
? #ifdef NATIVE
? char *astring = "convert ASCII to native character set";
? #else
? #ifdef MAC
? char *astring = "convert to Mac textfile format";
```

```
 ? #else
 ? #ifdef DOS
 ? char *astring = "convert to DOS textfile format";
 ? #else *„
 ? char *astring = "convert^ to Unix textfile format";
 ? #endif /*:?DOS */
 ? #endif /* ?MAC */
 ? #endif /* ?NATIVE */
```

В этом фрагменте, вообще говоря, лучше было бы использовать tfelif после каждого определения — тогда бы не было такого ненужного скопления tfendif в конце. Однако главная проблема вовсе не в этом, а в том, что, несмотря на все старания, код плохо переносим, потому что он ведет себя по-разному в разных системах, а для работы в каждой новой системе должен быть дополнен новым Sifdef. Одна-единственная строка с унифицированным текстом (но на самом деле столь же информативным) была бы гораздо удобнее, проще и переносимее:

```
char *astring = "convert
 to local text format";
```

Для этой строки никаких условий не нужно, она будет выглядеть одинаково во всех системах.

Смешивание управляющей логики времени компиляции (определяемой выражениями «if def) и времени исполнения приводит к еще более трудно воспринимаемому коду:

```
    ? #ifndef DISKSYS
 ? for (i = 1; i <= msg->dbgmsg.msg_total; i++)
 ? #endif
  ? #ifdef DISKSYS
 ? i = dbgmsgno;
 ? if (i <= msg->dbgmsg.msg_total)
 ? #endif
 ? {
  ?
 ? if (msg->dbgmsg.msg_"total == i)
 ?#ifndef DISKSYS
 ? break; /* больше ожидаемых сообщений нет */
 ? еще около 30 строк с условной компиляцией
 ? #endif
  ? }
```

Даже будучи явно безопасной, условная компиляция может быть заменена более простым кодом. Например, tfifdef часто используют для управления отладочным кодом:

```
    ? ftifdef DEBUG

    ? printf(...);

    ? tfendif
```

однако обычное выражение if с константой в условии может делать то же самое:

```
    enum { DEBUG = 0 };
 if (DEBUG) {
 printf(...);
}
```

Если DEBUG есть ноль, то большинство компиляторов не сгенерируют для приведенного фрагмента никакого кода, но при этом они еще и проверят синтаксис. Секция с #if def, наоборот, может содержать синтаксические ошибки, которые сорвут компиляцию, как только соответствующее условие #if def окажется выполнено.

Иногда условия компиляции содержат большие блоки кода:

```
    tfifdef notdef /* неопределенный символ */ ttendif
```

или

```
    #if O
 tfendif
```

В таком случае этот код лучше вынести в отдельные файлы, которые будут подключаться при компиляции при соблюдении определенных условий. К этой теме мы еще вернемся в следующем разделе.

Начиная адаптировать программу к новой среде, не делайте копии всей программы, а перерабатывайте исходный код. Скорее всего, вам придется вносить изменения в основное тело программы, и при редактировании копии вы через какое-то время получили бы новую, отличающуюся от исходной версию. Изо всех сил стремитесь к тому, чтобы у вас существовала единственная версия программы; при необходимости подстроиться под конкретную систему старайтесь вносить изменения таким образом, чтобы они работали во всех системах. Измените внутренние интерфейсы, если надо, но не нарушайте целостности кода; не пытайтесь решить проблему с помощью #if def. При таком подходе каждое изменение сделает вашу программу все более переносимой, а не более специализированной. Сужайте пересечение, а не расширяйте объединение.

Мы уже привели много доводов против использования условной компиляции, но не упоминали еще о главной проблеме: ее практически невозможно оттестировать. Каждое выражение tfif def разделяет всю программу на две по отдельности компилируемые программы, и определить, все ли возможные варианты программ были скомпилированы и проверены, очень сложно. Если в один блок tfifdef было внесено изменение, то может статься, что изменить надо и другие блоки, но проверить эти изменения можно будет только в той среде, которая вызовет эти tfifdef к исполнению. Точно так же, если мы добавляем блок #if def, то трудно изолировать это изменение, то есть определить, какие еще условия должны быть учтены и в каких еще местах должен быть изменен код. Наконец, если некий блок кода должен быть опущен в соответствии с условием, то компилятор его просто не видит, и проверить этот блок можно только в соответствующей конфигурации. Вот небольшой пример подобной проблемы — программа компилируется, если _MAC определено, и отказывается это делать в противном случае:

```
flifdef _MAC
 printf("This is Macintosh\r");
```

```
 #else
This will give a syntax error on other systems
 #endif
```

Итак, мы предпочитаем использовать только те возможности, которые присутствуют во всех средах, где будет исполняться программа. Мы всегда можем скомпилировать и протестировать весь код. Если что-то вызывает проблемы с переносимостью, мы переписываем этот кусок, а не добавляем условно компилируемый код; таким образом, переносимость все время улучшается.

Некоторые большие системы распространяются с конфигурационными скриптами, которые помогают приспособить код к локальной среде. Во время компиляции скрипт проверяет возможности среды: расположение заголовочных файлов и библиотек, порядок байтов внутри слов, размер типов, уже известные неверные реализации функций (таких на удивление много) и т. п. — и генерирует параметры настройки или make-файлы (makefile), которые описывают нужные настройки для данной ситуации. Эти скрипты могут быть большими и сложными, они являются важной частью дистрибутивного пакета и требуют постоянной поддержки. Иногда такие сложные способы оказываются полезны, но все же, чем переносимее будет ваш код и чем меньше #if def будет в нем использова-' но, тем проще и безопаснее будет происходить его настройка и установка.

Упражнение 8-1

Выясните, как ваш компилятор обрабатывает код, содержащийся внутри условного блока типа

```
    const int DEBUG = 0;
 /* или enum { DEBUG = 0 }; */
 /* или final boolean DEBUG = false; */
if (DEBUG) {
}
```

При каких обстоятельствах компилятор проверяет синтаксис? Когда он генерирует код?

Если у вас есть доступ к разным компиляторам, поэкспериментируйте с ними и сравните результаты.

Изоляция

Нам хотелось бы иметь единый исходный код, который бы компилировался без изменений во всех системах, но, к сожалению, это может быть нереально. Однако было бы ошибкой позволить непереносимому коду расползтись по всей программе, как это и происходит при условной компиляции.

Выносите системные различия в отдельные файлы. Когда для разных систем требуется разный код, его лучше выносить в отдельные файлы — один файл на каждую систему. Например, текстовый редактор Sam работает под Unix, Windows и в ряде других операционных систем. Интерфейсы с системой меняются от среды к среде очень широко, но большая часть кода Sam везде идентична. Все различия, связанные с конкретной системой, вынесены в отдельные файлы: unix. с содержит код интерфейса с системами Unix, а windows, с — со средой Windows. В этих файлах реализуются переносимые интерфейсы с операционной системой, различия же

получаются скрытыми. Таким образом, можно сказать, что Sam написан для своей собственной виртуальной операционной системы, которая переносится на различные реальные системы с помощью пары сот строк кода на С, реализующих несколько небольших, но непереносимых операций, вызывающих функции конкретной системы.

Самое интересное, что графические оболочки различных операционных систем не играют почти никакого значения: Sam имеет собственную переносимую библиотеку для своей графики. Несмотря на то что написать такую библиотеку, конечно же, гораздо труднее, чем просто адаптировать код под данную систему (код интерфейса с системой X Window, например, по своим размерам приближается к половине всего остального ко,ш Sam), суммарные затраты для нескольких систем получаются все равно меньше. При этом не надо забывать, что графическая библиотека ценна сама по себе и использовалась для создания и других переносимых программ.

Sam — это довольно старая программа; в наши дни переносимые графические оболочки, такие как OpenGL, Tcl/Tk и Java, доступны для большого числа платформ. Создание кода на их основе вместо использования собственных графических библиотек обеспечит вашим программам большую область применения.

Прячьте системные различия за интерфейсами. Абстрагирование — мощный способ разграничения переносимой и непереносимой частей программы. Хороший тому пример — библиотеки ввода-вывода, имеющиеся в большинстве языков программирования: они оперируют абстрактным представлением внешней памяти в терминах файлов, которые можно открывать, закрывать, читать или записывать, не затрагивая вопросов их физического расположения или структуры. Программы, которые придерживаются этого интерфейса, будут исполняться на любой системе, где он реализован.

Реализация редактора Sam представляет собой пример абстракции другого рода. Интерфейс описан для файловой системы и графических операций, а программа использует только свойства этого интерфейса. Сам интерфейс использует те возможности, которые имеются в конкретной операционной системе, что приводит к появлению абсолютно разных реализаций для разных систем, но программа, использующая его, зависит только от этого интерфейса, а не от его конкретной реализации, и поэтому ее не требуется изменять при переносе между системами.

Подход к переносимости, реализованный в системе Java, — пример того, насколько далеко можно здесь продвинуться. Программа на Java транслируется вЪперации "виртуальной машины", модели компьютера, которую можно реализовать на любой настоящей машине. Библиотеки Java предоставляют унифицированный доступ к возможностям системы: графике, пользовательскому интерфейсу, сети и т. п.; библиотеки же отображают этот доступ в возможности локальной системы. Теоретически должно быть возможно исполнить Java-программу (даже после трансляции) где угодно без каких-либо изменений.

Обмен данными

Текстовые данные передаются между системами в неизменном виде, | так что это самый простой и универсальный способ для передачи произ вольной информации между системами.

Для обмена данными используйте текст. С текстом легко оперировать посредством множества программ; его можно обрабатывать разными, самыми неожиданными

способами. Например, если вывод одной программы нельзя использовать напрямую для ввода в другую, то скрипт на Awk или Perl позволяет легко его преобразовать в нужный вид. С помощью g rep можно производить отбор строк, а ваш любимый редактор поможет произвести более сложные преобразования. Кроме всего прочего, I текстовые файлы легко документировать, да и пояснений к ним требуется гораздо меньше, потому что их всегда можно прочитать. Комментарий в текстовом файле может указывать, какая версия программы необ- \ ходима для обработки данных: например, первая строка файла PostScript определяет версию языка (и, возможно, тип документа):

```
%!PS-Adobe-2.0
```

В противоположность текстовым двоичные файлы требуют для своей j обработки специализированные средства, и их редко удается использовать совместно даже на одной машине. Существует множество известных программ, преобразующих произвольные двоичные данные в текст. Среди них стоит назвать binhex для Macintosh, uuencode и uudecode для Unix и различные инструменты, использующие кодировку MIME для преобразования двоичных данных в почтовые сообщения. В главе 9 мы расскажем о ряде средств паковки и распаковки двоичных данных для их передачи с сохранением переносимости. Кстати, уже само обилие таких инструментов подчеркивает наличие серьезных проблем, связанных с двоичными форматами.

С передачей текста связана одна давняя проблема: PC-системы (операционные системы D'OS и Windows) используют для обозначения конца строки символ возврата каретки '\r' к символ перевода строки ' \п', а системы Unix — только символ перевода строки. Возврат каретки — это артефакт, дошедший до нас от древнего устройства, называемого телетайпом, который имел операцию возврата каретки (CR) для возврата печатающего механизма в начало строки и отдельный оператор протяж-'ки на строку (LF — от Line Feed) для перевода этого механизма на следующую строку.

Несмотря на то что в современных компьютерах уже нет кареток, которые бы надо было возвращать, программное обеспечение для PC, по большей своей части, продолжает ожидать этой комбинации (известной также как CRLF, произносится "curliff") в конце каждой строки. Если в файле отсутствуют возвраты каретки, то он может быть проинтерпретирован как одна гигантская строка, при этом счетчики строк и символов могут вести себя непредсказуемым образом. Некоторые программы умеют изящно справляться с этой проблемой, но таких - меньшинство. Надо сказать, что PC не единственный виновник подобного безобразия: благодаря последовательному внедрению требований совместимости некоторые современные сетевые стандарты, такие как HTTP, также используют CRLF для разделения строк.

Мы можем посоветовать использовать стандартные интерфейсы, которые воспринимают CRLF в зависимости от конкретной системы -либо (для PC) удаляя символ \r при вводе и добавляя его обратно на выходе, либо (для Unix), никогда даже не создавая его. Для файлов, которые должны будут передаваться туда и обратно, необходимо написать программу, преобразующую их форматы.

Упражнение 8-2

Напишите программу, которая бы удаляла из файла случайно образовавшиеся возвраты каретки. После этого напишите еще одну, которая бы добавляла их в

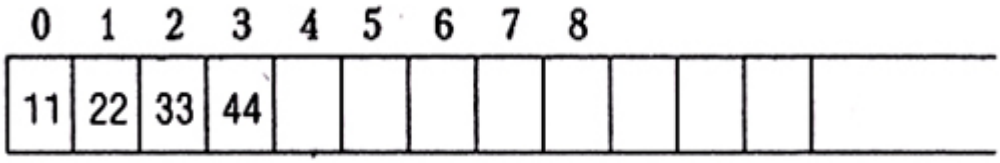
файл, заменяя символы перевода строки комбинацией возврата каретки и перевода строки. Как вы будете тестировать эти программы?

Порядок байтов

Несмотря на все описанные выше недостатки, иногда двоичные данные оказываются необходимы. Например, они несравненно более компактны и их гораздо быстрее декодировать, а эти факторы очень важны для компьютерных сетей. Но двоичные данные имеют серьезнейшие проблемы с переносимостью.

По крайней мере один вопрос решен: все современные машины имеют 8-битовые байты. Однако все объекты, большие байта, представляются на разных машинах по-разному, поэтому полагаться на какие-то определенные свойства было бы ошибкой. Короткие целые числа (обычно 16 битов, или 2 байта) могут иметь младший байт, расположенный как по меньшему адресу (little-endian, младшеконечное расположение), чем старший, так и по большему (big-endian, старшеконечное)i. Выбор варианта произволен, а некоторые машины вообще поддерживают обе! модели.

Итак, несмотря на то, что и старшеконечные и младшеконечные машины рассматривают память как последовательность слов, расположенных в одном и том же порядке, байты внутри слова они интерпретируют! различно. На приведенной диаграмме четыре байта, начинающиеся с поя зиции 0, представляют шестнадцатеричное целое 0x11223344 для старшеконечников и 0x44332211 —для младшеконечников.



Для того чтобы увидеть порядок байтов в действии, запустите следующую программу:

```
    /* byteorder: отображает байты длинного целого */
int main(void)
{
 unsigned long x;
unsigned char *p;
int i;
 /* 11 22 33 44 => big-endian */
/* 44 33 22 11 => little-endian */
/* x = Ox1122334455667788UL; для 64-битового long */
 x = Ox11223344UL;
 p = (unsigned char *) &x;
 for (i = 0; i < sizeof(long); i++)
pnntf("%x ", *p++); printf("\n"); return 0; }
```

На 32-битовом старшеконечнике на экран будет выведено

11 22 33 44

на младшеконечнике —

44 33 22 11

а на PDP-11 (16-битовая машина, все еще встречающаяся во встроенных системах) результатом будет

22 11 44 33

На машинах с 64-битовым типом long мы можем рассмотреть константу большей длины и увидеть те же результаты.

Наша программа выглядит довольно глупо, но если нам надо послать целое число через побайтовый интерфейс, например по сети, то надо решить, какой байт посылать первым, а какой вторым, и в этом выборе суть проблемы со старше- и младшеконечниками. Другими словами, наша программа неявнр выполняет то, что выражение

```
fwrite(&x, sizeof(x), 1, stdout);
```

делает явным образом. Небезопасно писать (отправлять) int (или short, или long) на одном компьютере и читать это число как int на другом.

Например, если компьютер-передатчик пишет с помощью

```
unsigned short x;

fwrite(&x, sizeof(x), 1, stdout);
```

а компьютер-приемник производит чтение так:

```
unsigned short x;

fread(&x, sizeof(x), 1, stdin);
```

то, если эти компьютеры имеют разный порядок байтов, значение x будет воспроизведено неправильно. Например, если отправлено было число 0x1000, то прочитано оно будет как 0x0010.

Эта проблема часто решается посредством условной компиляции и перестановки байтов, то есть примерно так:

```
? short x;
? fread(&x, sizeof(x), 1, stdin);
? flifdef BIG_ENDIAN
? /* осуществляем перестановку байтов */
? x = ((x&OxFF) « 8) | ((x»8) & OxFF);
? #endif
```

При пересылке большого количества двух- и четырехбайтовых целых такой подход получается слишком громоздким. На практике получает- ; ся, что байты при пересылке не один раз подвергаются перестановке.

Если ситуация выглядит невесело для short, то для более длинных типов она еще хуже — для них существует больше способов перепутать байты. А если к этому добавить еще всяческие преобразования между членами структур, ограничения на выравнивание и абсолютно таинственный порядок байтов в старых машинах, то проблема покажется 4 просто неразрешимой.

Используйте при обмене данными фиксированный порядок байтов.
Решение проблемы все же существует. Записывайте байты в каноническом порядке, используя переносимый код:

```
unsigned short x;
putchar(x » 8); /* пишем старший байт */
putchar(x & OxFF); /* пишем младший байт */
```

и считывайте их обратно побайтово, собирая первоначальные значения

```
    unsigned
    short x;
 x = getchar() « 8; /* читаем старший байт */
 x |= getchar() & OxFF; /* читаем младший байт */
```

Этот подход можно перенести и на структуры: записывайте значе-" ния членов структур в определенной последовательности, побайтово, без выравнивания. Неважно, какой порядок байтов вы изберете, — подойдет любой, лишь бы передатчик и приемник согласовали порядок байтов в передаче и количество байтов в каждом объекте. В следующей главе мы покажем несколько инструментов для упаковки и распаковки данных.

Побайтовая обработка может показаться довольно дорогим удовольствием, но в сравнении с вводом-выводом, при котором необходима упаковка и распаковка данных, потери времени окажутся незначительными. Представьте себе систему X Window, в которой клиент пишет данные с тем порядком байтов, что используется у него на машине, а сервер должен распаковывать все, что клиент посылает. На клиентском конце достигнута экономия в несколько инструкций, но сервер становится больше и сложнее, поскольку ему приходится обрабатывать одновременно данные с разными порядками байтов: ведь клиенты вполне могут иметь различные типы машин. В общем, сложностей куда больше. Кроме того, не забывайте, что это графическая оболочка, где расходы на распаковку байтов будут с лихвой перекрыты выполнением той графической операции, которую они содержат.

Система X Window воспринимает данные от клиента с любым порядком байтов: считается, что это забота сервера. А система Plan 9, наоборот, сама задает порядок байтов для отправки сообщений файл-серверу (или графическому серверу), а данные запаковываются и распаковываются неким универсальным кодом вроде приведенного выше. На практике определить временные затраты во время исполнения оказывается практически невозможно; можно только сказать, что на фоне ввода-вывода упаковка данных оказывается малозаметной.

Java — язык более высокого уровня, чем C и C++, в нем порядок байтов скрыт совсем. Библиотеки представляют интерфейс Serializable, который определяет, как элементы данных пакуются для передачи.

Однако, если вы пишете на C или C++, всю работу придется выполнять самостоятельно. Главное, что можно сказать про побайтовую обработку: она решает имеющуюся проблему для всех машин с 8-битовыми байтами, причем решает без участия flifdef. Мы еще вернемся к этой теме в следующей главе.

Итак, лучшим решением нередко оказывается преобразование информации в текстовый формат, который (не считая проблемы CRLF) является абсолютно переносимым: не существует никаких неопределенностей с его представлением. Однако и текстовый формат не является панацеей. Время и размер для некоторых программ могут быть критичны, кроме того, часть данных, особенно числа с плавающей точкой, могут потерять точность при округлениях в процессе передачи через printf и scanf. Если вам надо передавать числа с плавающей точкой, особо заботясь о точности, убедитесь, что у вас есть хорошая библиотека форматированного ввода-вывода; такие библиотеки существуют, просто ее может не оказаться конкретно в вашей системе. Особенно сложно представлять числа с плавающей точкой для переноса их в двоичном формате, но при должном внимании текст может эту проблему решить.

Существует один тонкий момент, связанный с использованием стан-| дартных функций для обработки двоичных файлов, — эти файлы необходимо открывать в двоичном режиме:

```
FILE *fin;
fin = fopen(binary_file, "rb");
c = getc(fin);
```

Если ' b' опущено, то в большинстве систем Unix это ни на что не повлиЯ яет, но в системах Windows первый встретившийся во вводе байт 1 Control-Z (восьмеричный 032, шестнадцатеричный 1A) прервет чтение! (такое происходило у нас с программой st rings из главы 5). В то же вре- j мя при использовании двоичного режима для чтения текстовых файлов; вам придется вставлять символы \ r во ввод и убирать их из вывода.

Переносимость и внесение усовершенствований

Одним из наиболее огорчительных источников проблем переносимо-1 сти является изменение системного программного обеспечения за время жизненного цикла. Изменения могут затронуть любой интерфейс системы, приводя к неоправданной несовместимости версий.

При изменении спецификации изменяйте и имя. Наш любимый (если) можно так выразиться) пример — изменение свойств команды Unix: echo, которая в изначальном виде предназначалась для простого вывода аргументов:

```
% echo hello, world

hello, world

 %
```

Однако со временем эта команда стала ключевой частью многих тов оболочки, и перед ней встала необходимость генерировать формате рованный вывод. Теперь echo стала некоторым образом интерпретирс вать аргументы, то есть стала неким аналогом printf:

```
% echo 'hello\nworld

hello

world

%
```

Новые возможности, конечно, полезны, но из-за них у всех скриптов, использующих echo в изначальном варианте, возникли проблемы с совместимостью. Поведение

```
% echo $PATH
```

стало зависеть от того, какая из версий echo используется. Если переменная случайно содержит обратную косую черту (что вполне может произойти в DOS или Windows), то echo попытается интерпретировать ее. Это похоже на разницу в выводе через printf(str) и printf ("%s", str) в случае, если переменная str содержит знак процента.

Мы привели только часть истории про echo, но уже то, что сказано, иллюстрирует основную проблему: изменения в системе приводят к появлению версий программ с преднамеренно различным поведением, что создает непреднамеренные проблемы с переносимостью. И исправить эти ошибки зачастую оказывается непросто. Проблем было бы гораздо меньше, если бы новая версия echo получила и новое имя.

Приведем еще один пример. В Unix существует команда sum, которая вводит размер файла и его контрольную сумму. Предназначена эта команда для проверки правильности передачи данных:

```
% sum file
52313 2 file
 %
% копируем file на другую машину
 %
 % telnet othermachine
 $
$ sum file
52313 2 file
 $
```

После передачи контрольная сумма не изменилась, так что с хорошей вероятностью можно считать, что передача прошла успешно.

Система разрасталась, появлялись новые версии, и в какой-то момент кто-то решил, что алгоритм вычисления контрольной суммы не идеален, и sum была изменена с использованием лучшего алгоритма. Кто-то еще пришел к тому же выводу и тоже изменил sum, реализовав другой, столь же хороший алгоритм, и т. д. В результате сейчас имеется несколько версий sum, каждая из которых выдает свой вариант ответа. Мы поставили ссперимент, скопировав некий файл на другие машины, чтобы выяснить, жие же результаты покажет sum в каждом конкретном случае:

```
    % sum file
52313 2 file
%
% копируем file на машину 2
% копируем file на машину 3
 % telnet machine2
$ Ф
$ sum file eaaOd468 713 file
 $ telnet machines >
> sum file 62992 1 file >
```

Непонятно, произошел сбой в передаче или просто мы столкнулися разными версиями sum. Может быть и то, и другое.

Таким образом, sum являет собой яркий пример препятствия на пути переносимости: программа, призванная помогать в копировании файлов с одной машины на другую, имеет несколько несовместимых версий, что делает ее абсолютно непригодной для использования.

Для выполнения изначально поставленной задачи первая версия sum] юлнелодходила: алгоритм, заложенный в ней, был не самым эффективным, но приемлемым. Ее "улучшение", может, и сделало собственно команду лучше, но зато использовать ее по назначению стало нельзя. И делят, надо сказать, не в том, что получилось несколько разных по существу! зманд, а в том, что все эти команды имеют одно и то же имя. Как видите! юблема несовместимости версий может оказаться весьма серьезной.

Поддерживайте совместимость с существующими программами и данными. Когда выпускается новая версия программы, например текстового редактора, то она, как правило, умеет читать файлы, созданный предыдущей версией. При этом мы ожидаем, что из-за добавления новых возможностей формат должен измениться. Но зачастую новые версии оказываются не в состоянии обеспечить способ записи в предыдущем формате. Пользователи новых версий, даже если они не обращаются к добавленным возможностям, не могут применять свои файлы совместно с пользователями более старой версии, таким образом, обновлять программы приходится сразу всем. Независимо от того, чем определяются такие решения — технической необходимостью или маркетинговой политикой, — о таких случаях можно только сожалеть.

Совместимостью сверху вниз называется возможность программы соответствовать спецификациям своих более ранних версий. Если вы собираетесь изменить свою программу, убедитесь, что при этом вы не создадите противоречий со старыми версиями и связанными с ними данными. Тщательно документируйте изменения и продумайте способ восстановить первоначальные возможности. И главное, задумайтесь над тем, перевесят ли достоинства предлагаемых вами усовершенствований потери от непереносимости, которые при этом возникнут.

Интернационализация

Если вы живете в Соединенных Штатах, то вы, может быть, забыли, что английский — не единственный язык на свете, ASCII — не единственный набор символов, $ — не единственный символ валюты, что даты могут записываться с указанием сначала

дня, а потом уже месяца, что время может записываться в формате с 24-мя часами и т. п. Так вот, еще один аспект переносимости в общем виде связан с созданием программ, переносимых между разными языками и культурными границами. Это на самом деле весьма обширная тема для разговора, и мы будем вынуждены ограничиться освещением лишь нескольких основных концепций.

Интернационализация — этот термин означает создание программ, исполняемых в любой культурной среде. Проблем с этим связано море — от набора символов до интерпретации иконок интерфейса.

Не рассчитывайте на ASCII. В большинстве стран мира наборы символов богаче, чем ASCII. Стандартная функция проверки символов из ctype. h, в общем, успешно справляется с этими различиями:

```
if (isalpha(c)) ...
```

Такое выражение не зависит от конкретной кодировки символов, а главное — если программу скомпилировать в локальной среде, то она будет работать корректно и в тех случаях, когда букв больше или меньше, чем от а до г. Правда, имя isalpha ("это буква?") говорит само за себя, а ведь существуют языки, в которых алфавита нет вообще.

В большинстве европейских стран кодировка ASCII, определяющая только значения до 0x7F (7 битов), расширяется дополнительными символами, которые представляют собой буквы национальных языков.

Кодировка Latin-1, широко распространенная в Западной Европе, является расширением ASCII, определяющим значения байтов от 80 до FF я небуквенных символов и акцентированных букв — так, значение E7 представляет букву д. Английское слово boy представляется в ASCII ли Latin-1) тремя байтами с шестнадцатеричными значениями 62 6F 79, эранцузское слово да гсоп представляется в Latin-1 байтами 67 61 72 E7 6E. В других языках определяются, соответственно, другие символы, но они не могут уложиться в 128 значений, не используемых в ASCII, к что существует множество конфликтующих стандартов для символов, привязанных к байтам от 80 до FF.

Некоторым языкам вообще не хватает 8 битов: в большинстве азиат-их языков существуют тысячи символов. В кодировках, используемых в Китае, Японии и Корее, на символ отводится 16 битов. В результате возникает глобальная проблема переносимости: как прочитать кумент на некотором языке на компьютере, настроенном на другой язык. Даже если все символы передадутся без ошибок, для прочтения на американском компьютере документа на китайском языке должны как шимум стоять специальные шрифты и соответствующее программное обеспечение. Если же мы захотим на одной машине использовать английский, китайский и русский языки, проблем у нас возникнет море.

Набор символов Unicode — попытка улучшить описанную ситуацию, предоставив единую кодировку для всех языков мира. Unicode совместима с 16-битовым подмножеством стандарта ISO 10646; в ней используется 16 битов на символ. Значения от OOFF и ниже относятся к Latin-1,то есть слово gargon будет представлено 16-битовыми значениями 0067 61 0072 OOE7 006F 006E. Кириллица занимает значения от 0401 до 04FF,а идеографическим языкам отведен большой блок, начинающийся ЮОО. Все известные и некоторые почти неизвестные языки мира представлены в Unicode, так что именно этой кодировкой и стоит пользовать

для передачи документов между странами или для хранения текста, .писанного на разных языках. Unicode стала весьма популярна в Интер-;те, и некоторые языки программирования даже поддерживают ее как стандартный формат: например, Java использует Unicode как родной набор символов для строк. Операционные системы Plan 9 и Inferno испольч ют Unicode более широко — даже для имен файлов и пользователей, Microsoft Windows поддерживает набор символов Unicode, но не считает о стандартом; большинство приложений Windows до сих пор лучше; .ботает с ASCII, хотя соотношение стремительно меняется в пользу: nicode.

Надо сказать, что и у Unicode есть недостатки: символы в ней уже не гещаются в один байт, поэтому текст в Unicode страдает от проблемы порядка байтов. Для преодоления этой напасти документы в Unicod перед передачей между программами или по сети обычно преобразуются в кодировку потока байтов, называемую UTF-8. В ней каждый 16-битовый символ кодируется для передачи как последовательность из 1, 2 или 3 байтов. Набор символов ASCII использует значения от 00 до 7F, все они умещаются в один байт при использовании UTF-8. Таким образом, получается, что UTF-8 односторонне совместима с ASCII. Значения между 80 и 7FF представляются двумя байтами, а значения от 800 и выше — тремя.LB UTF-8 слово gargon представляется байтами 67 61 72 C3 A7 6F 6E; значение Unicode E7 — символ g — представляется в UTF-8 двумя байтами — C3 A7.

Совместимость UTF-8 с ASCII весьма полезн-а, поскольку благодаря ей программы, рассматривающие текст как непрерывный поток байтов, могут работать с текстом Unicode на любом языке. Мы опробовали программу markov из третьей главы с текстом в UTF-8 на русском, греческом, японском и китайском языках, и она работала без каких-либо проблем. Для европейских языков, слова в которых разделяются ASCII-символами пробелов, табуляции или перевода строки, программа выдавала вполне сносный текст. При использовании других языков для того, чтобы получить что-то приемлемое на выходе, пришлось бы изменять правила разбиения текста на слова.

C и C++ поддерживают "широкие символы" (wide characters), которые представляются 16-битовыми или еще большими целыми. Суще-CTByiw и соответствующие функции, которые могут быть использованы для обработки символов в Unicode или в другом расширенном наборе символов. Строковые константы из широких символов записываются как L". . ,". Однако и здесь возникает большая проблема с переносимостью: программа с константами из широких символов может быть воспроизведена только на дисплее, использующем тот же набор символов. Поскольку символы должны быть конвертированы в поток байтов вроде UTF-8 для передачи между машинами, язык C предоставляет функции для преобразования широких символов в байты и обратно. Однако какое преобразование использовать? Интерпретация набора символов и описания кодировки потока байтов таятся в недрах библиотек, и вытащить их оттуда достаточно сложно; ситуация складывается не в нашу пользу. Может статься, в отдаленном светлом будущем все наконец придут к согласию об использовании единого набора символов, но пока что от проблемы порядка байтов никуда нам не деться.

Не ориентируйтесь только на английский язык. Создатели пользова-гельского интерфейса должны помнить, что в различных языках на выражение одного и того же понятия может потребоваться совершенно эазное количество символов, так что на экране и в массивах должно быть достаточно места.

Как же быть с сообщениями об ошибках? По крайней мере, в них не должно использоваться жаргона или сленга; лучше всего писать на самом простом языке. Полезно еще собрать тексты всех сообщений в каком-то одном месте программы — тогда можно будет быстро перевести их все.

Существует множество местных культурных особенностей, например формат дат mm/dd/yy используется только в Северной Америке. Если существует вероятность того, что ваша программа будет использоваться в другой стране, от таких особенностей надо по возможности избавиться. Иконки в графическом интерфейсе очень часто зависят от традиций; если понятия, на которых базируется зрительный образ, пользователю незнакомы, такая иконка его только дезориентирует.

Заключение

Переносимый код — это идеал, к которому надо стремиться, поскольку только он способен сэкономить вам время при переносе программы из системы в систему или при изменении текущего окружения. Однако переносимость достается не бесплатно — вам придется быть особенно внимательными при написании кода, а также представлять себе детали всех потенциально пригодных систем.

Мы рассмотрели два подхода к обеспечению переносимости — объединение и пересечение. Объединение предусматривает создание ; версий, которые работают в каждой конкретной среде; акцент при этом делается на механизмы вроде условной компиляции. Недостатков у этого подхода много: требуется писать больше кода, и нередко этот код получается весьма сложным; трудно изменять версии, очень трудно тестировать.

Пересечение предусматривает написание возможно большей части кода так, чтобы он работал без каких-либо изменений или условий в любой системе. Обработка системных различий, от которых все равно никуда не денешься, должна быть вынесена в отдельные файлы, которые будут играть роль интерфейса между программой и конкретной системой. И у этого подхода есть свои недостатки, главным из которых является потенциальное ухудшение производительности и даже возможностей, но в общем и целом все недостатки окупаются открывающимися преимуществами.

Дополнительная литература

Есть много описаний языков программирования, но немногие из них достаточно точны, чтобы служить полноценным справочным руководством по языку. Авторы данной книги имеют личные причины, чтобы предпочитать книгу "Язык программирования C" Брайана Кернигана и Денниса Ритчи (Brian Kernighan, Dennis Ritchie. The C Programming Language. Prentice Hall, 1988), но она не заменяет стандарт. В книге "C: Справочное руководство" Сэма Харбисона и Гая Стила (Sam Harbison, Guy Steele. C: A Reference Manual. Prentice Hall, 1994), которая дожила уже до четвертого издания, даны хорошие советы по переносимости. Официальные стандарты языков C и C++ доступны в ISO (The International Organization for Standardization). Книга, наиболее близкая к официальному стандарту языка Java, - "Спецификация языка Java" Джеймса Гослинга, Билла Джоя и Гая Стила (James Gosling, Bill Joy and Guy Steele. The Java Language. Specification. Addison-Wesley, 1996).

Книга Ричарда Стивенса "Программирование в системе Unix" (Richard Stevens. Advanced Programming in the Unix Environment. Addison-Wesley, 1992) является

отличным пособием для программистов под Unix; в частности, там дан подробный обзор вопросов переносимости между различными Unix-системами.

POSIX (the Portable Operating System Interface) — международный стандарт команд и библиотек, основанный на Unix-системах. Он описывает стандартную среду, переносимость исходного кода, а также унифицированный интерфейс для ввода-вывода, файловых систем и процессов. Этот стандарт описан в нескольких книгах, опубликованных IEEE.

Термин "big-endian" был введен Джонатаном Свифтом в 1726 г.1 Статья Денни Коэна "О святых войнах и мольбе о мире" (Danny Cohen. On holy wars and a plea for peace. IEEE Computer, October 1981) является замечательной басней о порядке байтов, в которой термин "endian" был впервые применен в компьютерной области.

В операционной системе Plan 9, разработанной в Bell Labs, переноносимость является главным приоритетом. Система компилируется из одного и того же исходного кода (без директив условной компиляции!) наЦИ ожестве разных процессоров и повсеместно использует символы icode. Последние версии редактора Sam, впервые описанного в "The Text Editor sam" (Software — Practice and Experience, 17, 11, p. 813-845, 1987), используют Unicode, но тем не менее работают на большом количестве систем. Проблемы работы с 16-битовыми наборами символов вроде Unicode описаны в статье Роба Пайка и Кена Томпсона "Hello, World ? (Документы зимней конференции USENIX'1993. Сан-Диего, 1993. С. 43-50). Впервые кодировка ГР-8 была представлена именно в этой статье. Данный документ, как тоследняя версия редактора Sam, также доступен на Web-сайте, посвященном системе Plan 9 в Bell Labs.

Система Inferno основывается на опыте Plan 9 и в чем-то похожа на j va, поскольку она определяет виртуальную машину, которая может ггь реализована на любой реальной машине, предоставляет язык (Limbo), который может быть скомпилирован в инструкции для этой фтуальной машины, и использует Unicode в качестве основного набор символов. Она также включает виртуальную операционную систему,которая предоставляет переносимый интерфейс ко множеству коммер-;ских систем. Она описана в статье "Операционная система Inferno" Шона Дорварда, Роба Пайка, Дэвида Л. Презотто, Денниса Ритчи, Говарда Трики и Филиппа Винтерботтома (Sean Dorward, Rob Pike, David eo Presotto, Dennis M. Ritchie, Howard W. Trickey и Philip Winter-ottom. The Inferno Operating System. Bell Labs Technical Journal, 2, 1, /inter, 1997).

Нотация

- Форматирование данных
- Регулярные выражения
- Программируемые инструменты
- Интерпретаторы, компиляторы и виртуальные машины
- Программы, которые пишут программы
- Использование макросов для генерации кода
- Компиляция "на лету"
- Дополнительная литература

Из всех творений человека
самым удивительным является язык.

Джайлс Литтон Страчи. Слова и поэзия

Правильный выбор языка может решающим образом влиять на простоту написания программы. Поэтому в арсенале практикующего программиста находятся не только языки общего назначения вроде C и его родственников, но и программируемые оболочки, языки скриптов, а также большое количество языков, ориентированных на конкретные приложения.

Преимущества хорошей нотации — способа записи — появляются при переходе от традиционного программирования к узкоспециальным проблемным областям. Регулярные выражения позволяют использовать компактные (из-за этого подчас превращающиеся в тайнопись) описания классов строк. Язык HTML позволяет определять внешний вид интерактивных документов, нередко используя встроенные программы на других языках, вроде JavaScript. PostScript рассматривает целый документ — например эту книгу — как стилизованную программу. Электронные таблицы и текстовые процессоры часто содержат в себе языки программирования типа Visual Basic, они используются для вычисления выражений, доступа к информации, управления размещением данных в документе.

Если вы ловите себя на том, что приходится писать слишком много кода' для выполнения рутинных операций, или если у вас возникают проблемы с тем, чтобы в удобной форме описать весь процесс, знайте — скорее всего, вы выбрали неправильный язык. Отсутствие правильного языка можно считать хорошим поводом написать его самостоятельно. Придумать свой язык вовсе не означает создать преемника Java: просто нередко самые запутанные проблемы проясняются при выборе должной нотации. В связи с этим вспомните форматные строки семейства printf, которые дают нам компактный и выразительный способ для управления выводом.

В этой главе мы говорим о том, как способ записи может помочь в решении наших проблем, а также демонстрируем ряд приемов, которые вы мо-Ц «ете использовать для создания собственных специализированных языков. Мы даже заставим программу писать другую программу, такова экстремальное использование способа записи распространено гораздо пире и осуществляется гораздо проще, чем думают многие программисты.

Форматирование данных

Между тем, что мы хотим сказать компьютеру ("реши мою проблему"), и тем, что нам приходится ему говорить для достижения нужного результата, всегда существует некоторый разрыв. Очевидно, что чем этот разрыв меньше, тем лучше. Хорошая нотация поможет нам сказать лменно то, что мы хотели, и препятствует ошибкам. Иногда хороший ;пособ записи освещает проблему в новом ракурсе, помогая в ее решении и подталкивая к новым открытиям.

Малые языки (little languages) — это нотации для узких областей применения. Эти языки не только предоставляют удобный интерфейс, но| л помогают организовать программу, в которой они реализуются. Хоро-лим примером является управляющая последовательность printf:

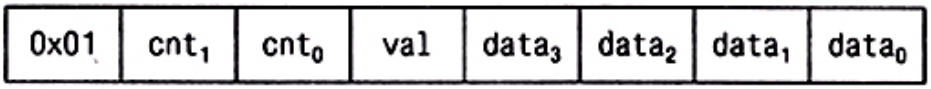
```
printf("%d %6.2f %-10.10s\n", f, s);
```

Здесь каждый знак процента обозначает место вставки значения сле-гующего аргумента printf ; за ним следуют необязательные флаги и раз- \ меры поля и, наконец, буква, которая указывает тип параметра. Такая нотация компактна, интуитивно понятна и легка в использовании; ее реализация достаточно проста и прямолинейна. Альтернативные возможности в C++ (lost ream) и Java (j ava . io) выглядят гораздо менее привле-сательно, поскольку они не предоставляют специальной нотации, хотя могут расшириться типами, определяемыми пользователем, и обеспечив 5ают проверку типов.

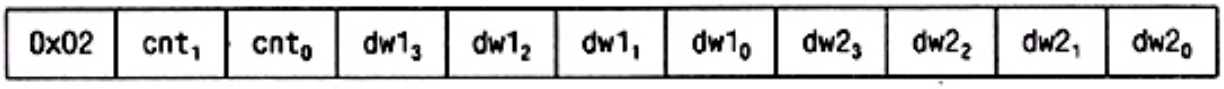
Некоторые нестандартные реализации printf позволяют добавлять] ;свои приведения типов к встроенным. Это удобно, когда вы работаете] : другими типами данных, нуждающимися в преобразованиях при вы- р| зоде. Например, компилятор может использовать знак %L_ для обозначения номера строки и имени файла; графическая система — использовать P для точки, а %R — для прямоугольника. Строка шифра из букв и номеров — сведения о биржевых котировках, которая рассматривалась нами главе 4, относится к тому же типу: это компактный способ записи таких котировок.

Схожие примеры можно придумать и для C и C++. Представим себе, что нам нужно пересылать пакеты, содержащие различные комбинации типов данных, из одной системы в другую. Как мы видели в главе 8, самым чистым решением была бы передача данных в текстовом виде. Однако для стандартного сетевого протокола лучше иметь двоичный формат по причинам эффективности и размера. Как же нам написать код для обработки пакетов, чтобы он был переносим, эффективен и прост в эксплуатации?

Для того чтобы дальнейшее обсуждение было конкретным, представим себе, что нам надо пересылать пакеты из 8-битовых, 16-битовых и 32-битовых элементов данных из одной системы в другую. В стандарте ANSI C оговорено, что в char может храниться как минимум 8 битов, 16 битов может храниться в sho rt и 32 бита — в long, так что мы, не мудрствуя лукаво, будем использовать именно эти типы для представления наших данных. Типов пакетов может быть много: пакет первого типа содержит однобайтовый спецификатор типа, двухбайтовый счетчик, однобайтовое значение и четырехбайтовый элемент данных:



Пакет второго типа может состоять из одного короткого и двух длинных слов данных:



Один из способов — написать отдельные функции упаковки и распаковки для каждого типа пакета:

```
    int pack_type1(unsigned char
 *buf, unsigned short count,
unsigned char val, unsigned long data) {
 unsigned char *bp;
 bp = buf; *bp++ = 0x01; *bp++ = count »
 8; *bp++ = count; *bp++ = val; *bp++ = data »
24; *bp++ = data » 16; *bp++ = data » 8;
 *bp++ = data; return bp - but; }
```

Для настоящего протокола потребовалось бы написать не один десяток сих функций — на все возможные варианты. Можно было бы несколь-упростить процесс, используя макросы или функции для обработки ювых типов данных (short, long и т. п.), но и тогда подобный повторяющийся код было бы трудно воспринимать, трудно поддерживать, 5 итоге он стал бы потенциальным источником ошибок.

Именно повторяемость кода и является его основной чертой, и здесь-то и может помочь грамотно подобранный способ записи. Позаимствовав идею у printf, мы можем определить свой маленький язык спецификации, в котором каждый пакет будет описываться краткой строкой, дающей информацию о размещении данных внутри него. Элементы пакета даруются последовательно: с обозначает 8-битовый символ, s — 16-битовое короткое целое, а 1 — 32-битовое длинное целое.Таким образом, на-зимер, пакет первого типа (включая первый байт определения типа) моет быть представлен форматной строкой cscl. Теперь мы в состоянии :пользовать одну-единственную функцию pack для создания пакетов обых типов; описанный только что пакет будет создан вызовом

```
pack(buf, "cscl", 0x01, count, val, data);
```

В нашей строке формата содержатся только описания данных, поэтому ам нет нужды использовать какие-либо специальные символы — вроде в printf.

На практике о способе декодирования данных могла бы сообщать риемнику информация, хранящаяся в начале пакета, но мы предположим, что для определения формата данных используется первый байт пакета. Передатчик кодирует данные в этом формате и передает их; приемник считывает пакет, анализирует первый байт и использует его для (екодирования всего остального.

Ниже приведена реализация pack, которая заполняет буфер buf кодированными в соответствии с форматом значениями аргументов. Мы сделали значения беззнаковыми, в том числе байты буфера пакета, чтобы избежать проблемы переноса знакового бита. Чтобы укоротить описания, мы использовали некоторые привычные определения типов:

```
typedef unsigned char uchar;
typedef unsigned short ushort;
typedef unsigned long ulong;
```

Точно так же, как sprinth, strcopy и им подобные, наша функция предполагает, что буфер имеет достаточный размер, чтобы вместить результат; обеспечить его должна вызывающая сторона. Мы не будем делать попыток определить несоответствия между строкой формата и списком аргументов.

```
    #include <stdarg.h>
 /* pack: запаковывает двоичные элементы в буфер, */
 /* возвращает длину */
 int pack(uchar *buf, char *fmt, ..'.)
 {
 va_list args; char *p; uchar *bp; ushort s; ulong 1;
 bp = buf;
 va_start(args, fmt); for (p = fmt; *p != '\0'i P++)
 ( switch (*p) { case 'c': /* char */
*bp++ = va_arg(args, int); break; , case 's'.: /*
 short */ s = va_arg(args, int); *bp++ = s »
8; *bp++ = s; break; case '!': /* long */
 1 = va_arg(args, ulong); *bp++ = 1 »
24; *bp++ = 1 » 16; *bp++ = 1 » 8; *bp++ = 1;
 break;
default: /* непредусмотренный тип */
 va_end(args); return -1; }
} va_end(args);
 return bp - buf;
 }
```

Функция pack использует заголовочный файл stda rg . h более активно, чем функция eprintf в главе 4. Аргументы последовательно извлекаются с помощью макроса va_arg, первым операндом которого является переменная типа va_list, инициализированная вызовом va_start; а в качестве второго операнда выступает тип аргумента (вот почему va_arg — то именно макрос, а не функция). По окончании обработки должен быть осуществлен вызов va_end. Несмотря на то что аргументы для ' c ' ; ' s ' представлены значениями char и short соответственно, они должны извлекаться как int, поскольку, когда аргументы представлены многоточием, C переводит char и short в int.

Теперь функции pack_type будут состоять у нас всего из одной строки, которой их аргументы будут просто заноситься в вызов pack:

```
    /* pack_type1: пакует пакет типа 1 */
 int pack_type1(uchar *buf, ushort count, uchar val, ulong data)
 {
return pack(buf, "cscl", 0x01, count, val, data); }
```

Для распаковки мы делаем то же самое и вместо того, чтобы писать тдельный код для обработки каждого типа пакетов, вызываем общую функцию unpack с соответствующей форматной строкой. Это централизирует преобразования типов:

```
     /* unpack: распаковывает элементы из buf,
 возвращает длину */
 int unpack(uchar *buf, char *fmt, ...)
   {
va_list args;
char *p;
uchar *bp, *pc;
ushort *ps;
ulong *pl;
bp = buf;
 va_start(args, fmt);
 for (p = fmt; *p !='\O'; p++) {
switch (*p) {
 case 'c': /* char */
pc = va_arg(args, uchar*);
 *pc = *bp++;
break; case 's': /* short */
ps = va_arg(args, ushort*);
 *ps = *bp++ « 8;
*ps |= *bp++;
break; case '!' /* long */
 pi = va_arg(args, ulong*);
*pl = *bp++ « 24;
*pl |= *bp++ « 16;
 *pl |= *bp++ « 8;
*pl |= *bp++;
break; default: /* непредусмотренный тип */
 va_end(args);
return -1; } }
va_end(args); return bp - buf;
 }
```

Так же как, например, scanf, функция unpack должна возвращать вызвавшему ее коду множество значений, поэтому ее аргументы являются указателями на переменные, где и будут храниться требуемые результаты. Значением функции является количество байтов в пакете, его можно использовать для контроля.

Все значения у нас беззнаковые. Мы придерживались размеров, которые ANSI C определяет для типов данных, и поэтому наш код можно переносить даже между машинами, имеющими разные размеры для типов short и long. Если только программа, использующая pack, не будет пытаться переслать как long (к примеру) значение, которое не может быть представлено 32 битами, то значение будет передано корректно; на самом деле мы передаем младшие 32 бита числа. Если же потребуется передавать более длинные значения, то нужно придумать другой формат.

Благодаря использованию unpack, функции для распаковки пакетов в зависимости от их типа стали выглядеть гораздо проще:

```
      /* unpack__type2: распаковывает и
 обрабатывает пакет типа 2 */
 int tinpack_type2(int n, uchar *buf)
{
uchar c; ushort count; ulong dw1, dw2;
  if (unpack(buf, "call'", &c, Scount, &dw1, &dw2)
 != n)
 return -1; assert(c == 0x02);
return process_type2(count, dw1, dw2); }
```

Перед тем как вызывать unpack_type2, мы должны сначала убедиться,что имеется пакет именно 2-го типа; распознаванием типа пакетов занимается цикл получателя, примерно такой:

```
    while ((n = readpacket(network, buf, BUFSIZ)) >
           0)
{ switch (buf[OJ)
{ default:
 eprintf("неправильный тип пакета Ox%x", buf[0]);
break; case 1:
unpack_type1(n, buf);
break; case 2:
unpack_type2(n, buf);
break;
 } }
```

Подобный стиль описания функций довольно размашист. Можно более мпактно определить таблицу указателей на распаковывающие функции, причем номер в таблице будет типом пакета:

```
    int (*unpackfn[])(int, uchar *) = {
 unpack_typeO,
unpack_type1,
 unpack_type2, };
```

Каждая функция в таблице разбирает пакет своего типа, проверяет ре-пьтат и инициирует дальнейшую обработку этого пакета. Благодаря эй таблице работа приемника получается более прямолинейной:

```
    /* receive: читает пакеты
 из сети, обрабатывает их */
 void receive(int network)
 {
uchar type, but[BUFSIZ];
  int n;
 while ((n = readpacket(network, buf, BUFSIZ)) > 0)
 { type = buf[0]; if (type >= NELEMS(unpackfn))
eprintf("неправильный тип пакета Ox%x", type);
if ((*unpackfn[type])(n, buf) < 0)
 eprintf("ошибка протокола, тип %x длина %d",
 type, n); } }
```

Итак, теперь код для обработки каждого пакета стал компактен; основная часть всей обработки происходит в одной функции и потому поддерживать такой код нетрудно. Код приемника теперь мало зависит от самого протокола; код его также прост и однозначен.

Этот пример основан на некоем реальном коде настоящего коммерческого сетевого протокола. Как только автор осознал, что этот, в общем, не-хитрЦй подход работоспособен, в небытие ушли несколько тысяч строк повторяющегося кода, напичканного ошибками (скорее даже, опискаМи), и вместо него появились несколько сот строк, поддерживать которые можно без всякого напряжения. Итак, хороший способ написания существенным образом улучшил как сам процесс работы, так и ее результат.

Упражнение 9-1

Измените pack и unpack так, чтобы можно было передавать и значения со знаком, причем даже между машинами, имеющими разные размеры short и long. Как вы измените форматную строку для обозначения элемента данных со знаком? Как можно протестировать код, чтобы убедиться, что он корректно передает, например, число -1 с компьютера с 32-битовым long на компьютер с 64-битовым long?

Упражнение 9-2

Добавьте в pack и unpack возможности обработки строк. (Есть вариант включать длину строки в форматную строку.) Добавьте возможность обработки повторяющихся значений с помощью счетчика. Как это соотносится с кодировкой строк?

Упражнение 9-3

Вспомните таблицу указателей на функции, которую мы применили олько что в программе на С. Такой же принцип лежит в основе механизма виртуальных функций С++. Перепишите pack, unpack и receive на ;++, чтобы прочувствовать все удобство этого способа.

Упражнение 9-4

Напишите версию printf для командной строки: пусть эта функция ечатает свой второй и последующие аргументы в формате, заданном первом аргументе. Надо отметить, что во многих оболочках имеется строенный аналог такой функции.

Упражнение 9-5

Напишите функцию, реализующую спецификации.формата, .исполь-/емого в какой-нибудь программе работы с электронными таблицами ли в Java-классе Decimal Format, где числа отображаются в соответствии некоторым заданным шаблоном, указывающим количество обязательных и возможных символов, положение десятичной точки и тысячных шятых и т. п. Для иллюстрации рассмотрим строку

##,##0.00

Эта строка задает число с двумя знаками после десятичной точки, э крайней мере одним знаком перед десятичной точкой, запятой в качестве разделителя тысяч и

заполняющими пробелами до позиции 10 000. аким образом, число 12345.67 будет представлено как 12, 345. 67, а .4 — ж **,**0.40 (для наглядности вместо пробелов мы вставили звездоч-i). Для получения полной спецификации можете обратиться к описанию DecimalFormat или программам работы с электронными таблицами.

Регулярные выражения

Спецификаторы формата для pack и unpack — достаточно простой юсоб записи, описывающий компоновку пакетов. Следующая тема iniero обсуждения — несколько более сложный, но гораздо более фазительный способ записи -- регулярные выражения (regular pressions), определяющие шаблоны текста (patterns of text). В этой книге мы время от времени использовали регулярные выражения, не вая им точного описания; они достаточно знакомы, чтобы понять их без особых пояснений. Хотя регулярные выражения широко распространены в средах программирования под Unix, в других системах они применяются гораздо реже, поэтому в данном разделе мы решили показать некоторые их преимущества. На случай, если у вас нет под рукой библиотеки с регулярными выражениями, мы приведем ее простейшую реализацию.

Существует несколько разновидностей регулярных выражений, но суть у них у всех одинакова — это способ описания шаблона буквенных символов, включающего в себя повторения, альтернативы и сокращения для отдельных классов символов вроде цифр или букв. Примером такого шаблона могут служить всем знакомые символы замещения (wildcards), используемые в процессорах командной строки или оболочках для задания образцов поиска имен файлов. Как правило, символ * используется для обозначения "любой строки символов", так что команда

```
C:\> del *.exe
```

использует шаблон, которому соответствуют все имена файлов, оканчивающиеся на ".-. exe". Как это часто бывает, детали меняются от системы к системе и даже от программы к программе.

Причуды отдельных программ могут навести на мысль, что регулярные выражения являются механизмом? создаваемым ad hoc, для каждого случая отдельно, но на самом деле регулярные выражения — это язык с формальной грамматикой и точным значением каждого выражения. Более того, при должной реализации конструкция работает очень быстро: благодаря соединению теории и практического опыта; вот пример в пользу специализированных алгоритмов, упоминавшихся нами в главе 2.

Регулярное выражение есть последовательность символов, которая определяет множество образцов поиска. Большинство символов просто-напросто соответствуют сами себе, так что регулярное выражение abc будет соответствовать именно этой строке символов, где бы она ни появлялась. Но, кроме того, существует несколько метасимволов, которые обозначают повторение, группировку или местоположение. В принятых в Unix регулярных выражениях символ ~ обозначает начало строки, а $ — конец строки, так что шаблон ~x соответствует только символу x в начале строки, x$ — только x в конце строки. Шаблону ~x$ соответствует только строка, содержащая единственный символ x, и, наконец, ~$ соответствует пустая строка.

Символ . (точка) обозначает любой символ, так что выражению x. y будут соответствовать xay, x2y и т. п., но не xy или xaby; выражению ~. $ соответствует строка из одного произвольного символа.

Набору символов внутри квадратных скобок соответствует любой ия этих символов. Таким образом, [0123456789] соответствует одной цифре, это выражение можно записать и в сокращенном виде: [0-9]1.

Эти строительные блоки комбинируются с помощью круглых скобок для группировки, символа — для обозначения альтернатив, * — для обозначения нуля или более совпадений, + — для обозначения одного или более совпадений и ? — для обозначения нуля или одного совпадения. Наконец, символ \ применяется в качестве префикса перед метасимволом для исключения его специального использования, то есть * обозначает просто символ *, а \\ — собственно символ обратной косой черты \.

Наиболее известным инструментом работы с регулярными выраже-йиймн является программа g rep, о которой мы уже несколько раз упоминали. Эта программа — чудесный пример, показывающий важность нотации. В ней регулярное выражение применяется к каждой строке вводимого файла и выводятся строки, которые содержат образцы поиска, описываемые этим выражением. Эта простая спецификация с помощью регулярных выражений позволяет справиться с большим количеством ежедневной рутинной работы. В приведенных ниже примерах обратите внимание на то, что синтаксис регулярных выражений, используемых в качестве аргументов g rep, отличается от шаблонов поиска, применяемых для задания набора имен файлов; различие обусловлено разным назначением выражений.

Какой исходный файл использует класс Regexp?

```
% grep Regexp *.java
```

В каком файле этот класс реализован?

```
% grep 'class.*Regexp' *.java
```

Куда я подевал это письмо от Боба?

```
% grep '"From:.* bob@' mail/*
```

Сколько непустых строк кода в этой программе?

```
grep *. c++ 1 we
```

Благодаря наличию флагов, позволяющих выводить номера соответствующих выражению строк, подсчитывать количество соответствий, осуществлять сравнение без учета регистра, инвертировать смысл выражения (то есть отбирать строки, не соответствующие шаблону) и т. п., программа grep используется сейчас настолько широко, что стала уже классическим примером программирования с помощью специальных инструментов.

К сожалению, не во всех системах имеется grep или ее аналог. Некоторые системы включают в себя библиотеку регулярных выражений, которая называется, как правило, regex или regexp, и ее можно использовать для создания собственной версии g rep. Если же нет ни того, ни другого, то на самом деле не так трудно реализовать какое-то скромное подмножество языка регулярных выражений. Здесь мы представляем реализацию регулярных выражений и grep, чтобы работать с ними; для простоты мы ограничимся метасимволами $. и * , причем * обозначает повторение предыдущей точки или обычного символа. Выбор этого подмножества

обеспечивает почти все возможности регулярных выражений, тогда как его программировать значительно проще, чем в исходном общем случае.

Начнем с самой функции, осуществляющей проверку на соответствие. Ее задача — определить, соответствует ли строка текста регулярному выражению:

```
    /* match: ищет regexp в text */
int match(char *regexp, char *text) {
if (regexp[0] == '',')
return matchhere(regexp+1, text);
do { /* должны попробовать,
даже если строка пуста */ if (matchhere(regexp, text))
return 1;
 } while (*text++ != '\0'); return 0; }
```

Если регулярное выражение начинается с ", то текст должен начинаться с символов, соответствующих остальной части выражения. При другом начале мы проходим по тексту, используя matchhere для выяснения, соответствует ли текст каждой позиции выражения. Как только мы находим соответствие, миссия наша завершена. Обратите внимание на использование do-while: выражениям может соответствовать пустая строка (например, шаблону $ соответствует пустая строка, а шаблону . * — любое количество символов, включая и ноль), поэтому вызвать matchhere мы должны даже в том случае, если строка текста пуста.

Большая часть работы выполняется в рекурсивной функции matchhere:

```
    /* matchhere: ищет regexp в начале text */
int matchhere(char *regexp, char *text)
{
if (regexp[0] == "\0')
 return 1; if (regexp[1] == '*')
return matchstar(regexp[0], regexp+2, text); if
 (regexp[0] == '$' && regexp[1] == 'ДО')
return *text == '\O'; if (*text!='\0' &&
 (regexp[0]==',' || regexp[0]==*text))
 return matchhere(regexp+1, text+1); return 0; }
```

Если регулярное выражение пусто, это означает, что мы дошли до его конца и, следовательно, нашли соответствие. Если выражение оканчивается символом $, оно имеет соответствие только в том случае, если текст также расположен в конце. Если выражение начинается с точки, то первый символ соответствующего текста может быть любым. В противном случае выражение начинается с какого-то обычного символа, который) в тексте соответствует только самому себе. Символы ~ и $, встречающиеся в середине регулярного выражения, рассматриваются как простые литеральные, а не метасимволы.

Отметьте, что matchhe re, убедившись в совпадении одного символа из; шаблона и подстроки, вызывает себя самое, таким образом, глубина ре] курсии может быть такой же, как и длина шаблона (при полном соответветствии).

Единственный сложный случай представляет собой выражение, начинающееся с символа и звездочки, например x*. В этом случае мы осуществляем вызов matchstar, где первым аргументом является операнд звездочки (то есть x), а следующими аргументами — шаблон после звездочки и сам текст.

```
    /* matchstar: ищет oregexp в начале text */
int matchstar(int c, char *regexp, char *text)
{
do { /* знак * означает ноль или более вхождений */
if (matchhere(regexp, text)) return 1;
" ' ---- i- -\n- ss fi,toYt++ __ c || c __ '))
 return 0; }
```

Здесь мы опять используем do-while — из-за того, что регулярному выражению x* может соответствовать и ноль символов. Цикл проверяет, совпадает ли текст с остатком регулярного выражения, пытаясь сопоставить их, пока первый символ текста совпадает с операндом звездочки.

Наша реализация получилась довольно простенькой, однако она работает и занимает всего лишь три десятка строк кода — можно утверждать, что для того, чтобы ввести в обращение регулярные выражения, не нужно применять никаких сложных средств.

Скоро мы представим некоторые соображения по расширению возможностей нашего кода, а пока напишем свою версию g rep, использующую match. Вот как выглядит основная часть:

```
    /* grep main: ищет regexp в файлах */
 int main(int argc, char *argv[]) {
 int i, nmatch;
 FILE *f;
 setprogname("grep"); if (argc < 2)
eprintf("usage: grep regexp [file ...]");
 nmatch = 0; if (argc == 2) {
  if (grep(argv[1], stdin, NULL))
 nmatch++; } else {
 for (i = 2; i < argc; i++)
 { f = fopen(argv[i], "r">; if (f == NULL) {
 weprintf("can't open %s:", argv[i]); continue;
 } if (grep(argv[1], f,
  argc>3 ? argv[i] : NULL) > 0)
 nmatch++; fclose(f); } }
   return nmatch == 0; }
```

По соглашению программы на С возвращают 0 при успешном заверше-и и ненулевое значение — при различных сбоях. Наша программа Ф, так же как и Unix-версия, считает выполнение успешным, если ла найдена строка, соответствующая шаблону. Поэтому она возвращает 0, если было найдено хотя бы одно соответствие; 1 — если соответствий найдено не было, и 2 (посредством eprintf) — если произошла шбка. Эти значения можно протестировать, использовав в качестве олочки какую-то другую программу.

Функция grep сканирует один файл, вызывая match для каждой его строки:

```
    /* grep: ищет regexp в файле */
int grep(char *regexp, FILE *f, char *name)
 {
 int n, nmatch;
  char buf[BUFSIZ];
```

```
 nmatch = 0;
 while (fgets(buf, sizeof but, f) != NULL)
 { n = strlen(buf); if (n > 0 && buf[n-1] == '\n')
 buf[n-1] = '\0'; if (match(regexp, buf))
{ nmatch++; if (name != NULL)
 printf ("%s:", name); printf("%s\n", buf); }
 } return nmatch;
 i /
```

Если открыть файл не удается, то программа не прекращает работу. Такое поведение было выбрано для того, чтобы при вызове

```
% grep herpolhode *.*
```

даже если какой-то файл в каталоге не может быть открыт, g rep уведомляла об этом и продолжала работу; если бы на этом она останавливалась, пользователю пришлось бы вручную набивать список всех файлов в каталоге, кроме недоступного. Обратите внимание и на то, что grep выводит имя файла и строку, соответствующую шаблону, однако имя файла не выводится, если чтение происходит из стандартного потока ввода или из одного файла. Может быть, такое поведение покажется кому-то старомодным, однако оно отражает идиоматический стиль, в основе которого лежит опыт. Когда задан только один источник ввода, миссия дгер состоит, как правило, в выборе соответствующих строк, и имя файла никого не интересует. А. вот если поиск происходит в нескольких файлах, то, как правило, задачей является найти все совпадения с какой-то строкой, и здесь имя файла будет весьма информативно. Сравните

```
% strings markov.exe | grep 'DOS mode'
```

и

```
% grep grammer chapter*.txt
```

Продуманность нюансов — одна из тех черт, что сделали grep столь популярным инструментом; здесь мы видим, что для создания хороших программ нотация обязательно должна учитывать человеческую психологию.

В нашей реализации match завершает работу сразу же, как только найдено соответствие. Для g rep это вполне подходит в качестве поведения по умолчанию, но для реализации оператора подстановки (найти и заменить) в текстовом редакторе больше подошел поиск крайне левого самого длинного совпадения (leftmost longest). Например, если нам задан текст "аааaa", то шаблону a* соответствует, в частности, и пустая строка в начале текста, однако более естественным было бы включить в совпадение все пять a. Для того чтобы match искала крайне левую и самую длинную строку, matchstar надо переписать так, чтобы добиться жадного (greedy) поведения1 алгоритма: вместо того чтобы просматривать каждый символ текста слева направо, он должен каждый раз пытаться пройти до конца самой длинной строки, соответствующей оператору-звездочке, и откатываться назад, если оставшаяся часть обрабатываемого текста не соответствует оставшейся части шаблона. Другими словами, она должна выполняться справа налево. Вот как выглядит версия matchstar, осуществляющая поиск крайне левого максимального совпадения:

```
/* matchstar: поиск в начале теста c*гедехр */ int matchstar
(int c, char *regexp, char *text)
```

```
{
char *t;
for (t = text; *t != ДО' && (*t == c || c == '.'); t++)
do { /* знак * соответствует
 нулю или более вхождений */ if
 (matchhere(regexp, t))
return 1;
} while (t-- > text); return 0;
X
;
```

Для g rep абсолютно неважно, какое именно совпадение будет обнаруже-ю, поскольку эта программа проверяет лишь наличие соответствий во-эбще и выводит всю строку, содержащую соответствие. Таким образом, поиск крайне левого максимального совпадения, жизненно необходимый для оператора замещения, хоть и делает некоторую ненужную для g rep работу, все же может быть применен без изменений и для этой программы.

Наша версия g rep вполне сравнима с версиями, поставляемыми с различными системами (неважно, что синтаксис наших регулярных выражений несколько скуднее). Существуют некоторые патологические выражения, которые приводят к экспоненциальному возрастанию трудоемкости поиска, — например, выражение a*a*a*a*a*Ь при введенном тексте aaaaaaaaac, однако экспоненциальный поиск присутствует и в некоторых коммерческих реализациях. Вариант g rep, применяемый в Unix (он называется eg rep), использует более сложный алгоритм поиска соответствий; этот алгоритм гарантирует линейный поиск, избегая отката, когда не подходит частичное соответствие.

А как насчет того, чтобы match умела обрабатывать полноценные регулярные выражения? Для этого надо добавить возможность сопоставления классов символов типа [a-zA-Z] конкретному символу алфавита, возможность исключать значение метасимволов (например, для поиска точки нужно обозначить этот символ как литеральный), предусмотреть скобки для группировки, а также ввести альтернативы (аос или clef). Первый шаг здесь — облегчить match работу, скомпилировав шаблон в некое новое представление, которое было бы проще сканировать: слишком накладно производить разбор класса символов для каждого нового сравнения его с одним символом; предварительно вычисленное представление, основанное на битовых массивах, сделает классы символов гораздо более эффективными. Полная реализация регулярных выражений, со скобками и альтернативами, получится гораздо более сложной; облегчить написание помогут некоторые средства, о которых мы поговорим далее в этой главе.

Упражнение 9-6

Как соотносится быстродействие match и strstr при поиске простого текста (без метасимволов)?

Упражнение 9-7

Напишите версию matchhere без рекурсии и сравните ее быстродействие с рекурсивной версией.

Упражнение 9-8

Добавьте в g rep несколько ключей командной строки. К наиболее популярным ключам относятся -y для инвертирования смысла шаблона, -i для поиска без учета регистра и ~n для вывода номеров строк. Как должны выводиться номера строк? Должны ли они печататься на той же строке, что и совпавший текст?

Упражнение 9-9

Добавьте в match операторы + (один или более) и ? (ноль или один). Шаблону a+bb? соответствует строка из одного или более a, за которыми следует одно или два b.

Упражнение 9-10

В нашей реализации match специальное значение символов " и $ отключается, если они не стоят, соответственно, в начале или конце выражения, а звездочка * рассматривается как обычный символ, если она не стоит непосредственно после литерального символа или точки. Однако более стандартным поведением является~отключение значения метасимвола постановкой перед ним символа обратной косой черты (\). Измените match так, чтобы она обрабатывала этот символ именно таким образом.

Упражнение 9-11

Добавьте в match классы символов. Класс символов определяет соответствие любому из символов, заключенных в квадратные скобки. Для Удобства лучше ввести диапазоны, например [a-z] соответствует любой строчной букве (английского алфавита!), а также определить способ инвертирования смысла, — например, [~0-9] соответствует любому символу, кроме цифры.

Упражнение 9-12

Измените match так, чтобы в ней использовалась версия matchstar, (которой ищется крайне левое максимальное соответствие. Кроме того, :ледует добиться возвращения позиции символов начала и конца текста, ;оответствующего шаблону. Теперь создайте новую версию дгер, кото- j рая вела бы себя как старая, но выполняла замену текста, соответству-. ющего шаблону, заданным новым текстом и выводила полученные строчки. Пример вызова:

```
grep 'homoiousian' 'homoousian' mission.stmt
```

Упражнение 9-13

Измените match и дгер так, чтобы они-работали со строками символов Unicode формата UTF-8. Поскольку UTF-8 и Unicode являются расширением набора 7-битового ASCII, такое изменение будет совместимо с предыдущей версией. Регулярные выражения, так же как и текст, в котором происходит поиск, должны корректно работать с UTF-8. Как в этом случае должны быть реализованы классы символов?

Упражнение 9-14

Напишите программу для автоматического тестирования регулярных выражений: она должна генерировать тестовые выражения и строки, в которых будет

происходить поиск. Если можете, используйте уже существующую библиотеку как прототип для ответов, — возможно, вы найдете какие-то ошибки и в ней.

Программируемые инструменты

Множество инструментов группируется вокруг языков специального.'! назначения. Программа g rep — всего лишь одна из целого семейства инструментов, которые используют регулярные выражения или другие языки для разрешения программистских проблем.

Одним из первых подобных инструментов стал командный интерпретатор, или язык управления заданиями. Достаточно быстро стало понятно, что последовательность команд можно поместить в отдельный файл, а потом запустить экземпляр интерпретатора команд, или оболочки /(shell), указав этот файл в качестве источника ввода. Следующим шагомстало уже добавление параметров, условных выражений, циклов, переменных и т. п., то есть всего того, из чего в нашем представлении состоит нормальный язык программирования. Единственное отличие заключалось в том, что существовал только один тип данных — строки, а операторами в программах-оболочках являлись различные программы, осуществлявшие порой достаточно интересные вычисления. Несмотря на то что программирование под конкретные оболочки сейчас уже уходит в прошлое, уступая место работе с инструментами типа Perl в командных средах или щелканью по кнопкам в графических средах, этот "старинный" подход по-прежнему остается простым и достаточно эффективным способом создания каких-то сложных операций из простых частей.

Еще один программируемый инструмент, Awk, представляет собой небольшой специализированный язык, который предназначен для отбора и преобразования элементов входного потока; он ищет в этом потоке соответствия заданным шаблонам, а когда находит, то выполняет связанные с шаблонами действия. Как мы уже видели в главе 3, Awk автоматически читает входные файлы и разделяет каждую прочитанную строку на поля, обозначаемые от $1 до $NF, где NF есть количество полей в строке. Его поведение "по умолчанию" удобно для выполнения большого числа типовых действий, поэтому многие полезные программы пишутся на Awk в одну строчку. Так, например, программа

```
# split.awk: расщепляет текст на отдельные слова { for (i = 1; i <=
NF; i+3-) print $i }
```

выводит "слова" по одному в строке. И напротив, вот программа f mt, которая заполняет каждую выводимую строку словами (до 60 символов); пустая строка означает конец параграфа:

```
    # fmt.awk: форматирует в 60-символьные строки
/./ { for (i = 1; i <= NF; i++) addword($i) }
  # непустая строка /~$/ { printlineO; print "" }
 # пустая строка
 END { printlineO }
 function addword(w) {
 if (length(line) + 1 + length(w) > 60)
 printlineO if (length(line) == 0)
 line = w else
 line = line " " w
```

```
}
function printline() {
if (len'gth(line) > 0) { print line line = "." }
 }
```

Мы часто используем f mt для того, чтобы заново разбить на абзацы почтовые сообщения и другие короткие документы; ее же мы использовали главе 3 для форматирования вывода программы markov.

Программируемые инструменты часто берут свое начало от малых ыков программирования, созданных для более простого выражения шений проблем в какой-то узкой предметной области. Прелестным шмером является инструмент Unix под названием eqn, который обра-тывает математические формулы. Язык, применяемый в нем для вво-., очень похож на обычный: например, выражение f мы прочитали бы лух как "пи на два" (pi over two), и в этом языке оно так и записывается - pi over 2. Тот же подход применяется и в TEX, в нем это выраже-ie было бы записано как \pi \over 2. Если для проблемы, которую вы пытаетесь разрешить, уже существует естественный или привычный :особ записи, используйте его или, на худой конец, попробуйте его дотировать: не пытайтесь писать с нуля.

Awk развился из программы, которая использовала регулярные выражения для выявления аномалий в записях телефонного трафика; теперь же] ffk содержит в себе переменные, выражения, циклы и т. п., — и это делает о полноценным языком программирования. Perl и Tel первоначально проектировались с целью совместить удобство и выразительность малых языков с помощью полноценных языков программирования. В результате получились действительно удобные языки программирования общего назначения, хотя, конечно, чаще всего их используют для обработки текста.

Для подобных инструментов применяют общий термин — языки скриптов (scripting languages). Такое название объясняется их происхождением ранних командных интерпретаторов, вся "программируемость" которых ограничивалась исполнением заранее записанных "сценариев" cript) программ. Языки скриптов позволяют использовать регулярные выражения более творчески: не только для поиска соответствий шаблону — простого обнаружения соответствия, но и для определения участ-iB текста, которые должны быть изменены. Именно это осуществляется в двух командах regsub (от regular expression substitution — замена с помощью регулярных выражений), реализованных в приводимой ниже программе на языке Tel. Программа эта является несколько более общей формулой программы из главы 4, получающей биржевые котировки; новая версия выполняет это, получая данные из URL, передаваемого ей в качестве первого аргумента. Первая замена удаляет строку http://, если она присутствует; вторая — удаляет символ / и заменяет его пробелом, разбивая, таким образом, аргумент на два поля. Команда lindex получает поля из строки (начиная с индекса 0). Текст, заключенный в квадратные скобки, выполняется как команда Tel и заменяется результирующим текстом; последовательность $x заменяется значением переменной x.

```
    # geturl.tcl: получает документ из URL
# ввод имеет вид [http://Jabc.def.com[/whatever...]
# если присутствует, удалить http://
regsub "http://" $argv "" argv ;
 # в начале строки заменить
символ / пробелом regsub ','/" largv " " argv ;
# выполнить сетевое соединение
```

```
 set so [socket [lindex $argv 0] 80] ;
 set q "/[lindex $argv 1]"
puts $so "GET $q HTTP/1.0\n\n" ;
# послать запрос
flush $so
ft пропустить заголовок
 while {[gets $so line] >= 0 && $line != ""} {} ;
# прочесть и вывести весь ответ
 puts [read $so] ;
```

Этот скрипт, как правило, производит весьма объемистый вывод, большую часть которого составляют тэги HTML, заключенные между < и >. Perl удобен для текстовых подстановок, так что нашим следующим инструментом станет скрипт на Perl, который использует регулярные выражения и подстановки для удаления тэгов:

```
    # unhtmi.pl: delete HTML tags
 while (<>)
{ tt собирает весь ввод в одну строку
 $st r .'=$_; # накапливая вводимые строки }
 $str =~ s/<p>]*>//g; # удалить <...>
$str =- s/ / /g; n заменить   пробелом
$str =~ s/\s+/\n/g; it сжать свободное место print
$str:
```

Для тех, кто не знаком с Perl, этот код будет загадкой. Конструкция

```
     $str =" s/regexp/repl/g
```

строке str подставляет строку repl вместо текста, соответствующего регулярному выражению readexp (берется крайне левое максимальное соответствие); завершающий символу (от "global") означает, что действия надо произвести глобально, для всех найденных соответствий, и не только для первого. Последовательность метасимволов \s является сокращенным обозначением символа пустого места (пробел, знак табуляции, символ перевода строки и т. п.); Это означает перевод строки. Строка " " — это символ HTML, как и те, о которых мы упоминали I главе 2, он означает non-breakable space — неразрывный пробел.

Собрав все написанное воедино, мы получим совершенно идиотский, но функционирующий web-браузер, реализованный как скрипт командного интерпретатора, состоящий из одной строки:

```
    # web: получает web-страницу и
форматирует ее текст,
# игнорируя HTML
geturl.tcl $1 | unhtml.pl | fmt.awk
```

Такой вызов получит web-страницу, отбросит все управление и форма-гирование и отформатирует текст по своим собственным правилам. Получился быстрый способ достать страницу текста из web.

Отметьте, что мы неспроста использовали сразу несколько Жыков (Tel, Perl, Awk) и в каждом из них — регулярные выражения. Собственно говоря, мощь различных нотаций и состоит как раз в подборе наилучшего варианта для каждой проблемы.

Тем особенно хорош для получения текста из сети, Perl и Awk прекрасно редактируют и форматируют текст, а регулярные выражения — отличный способ определения фрагментов текста, которые должны быть найдены и изменены. Объединение всех этих языков получается гораздо более мощным, чем любой из них в отдельности. Целесообразно разбить задачу на части, если можнс выиграть за счет составления правильной нотации в каждой из них.

Интерпретаторы, компиляторы и виртуальные машины

Какой путь проходит программа от исходного кода до исполнения? Если язык достаточно прост, как в printf или в наших простейших регулярных выражениях, то исполняться может сам исходный код. Это несложно; таким образом можно запускать программу сразу же по написании.

Нужно искать компромисс между временем на подготовку к запуску и скоростью исполнения. Если язык не слишком прост, то для исполнения желательно преобразовать исходный код в некое приемлемое и эффективное внутреннее представление. На это начальное преобразование исходного кода тратится определенное время, но оно вполне окупается более быстрым исполнением. Программы, в которых преобразование и исполнение объединены в один процесс, читающий исходный текст, преобразующий его и его исполняющий, называются интерпретаторами (interpreter). Awk и Perl, как и большая часть других языков скриптов и языков специального назначения, являются именно интерпретаторами.

Третья возможность — генерировать инструкции для конкретного типа компьютера, на котором должна исполняться программа; этим занимаются компиляторы. Такой подход требует наибольших затрат времени на подготовку, однако в результате ведет и к наиболее быстрому последующему исполнению.

Существуют и другие комбинации. Одна из них — мы рассмотрим ее подробно в данном разделе — это компиляция программы в инструкции для воображаемого компьютера (виртуальной машины — virtual machine), который можно имитировать на любом реальном компьютере. Виртуальная машина сочетает в себе многие преимущества обычной интерпретации и компиляции.

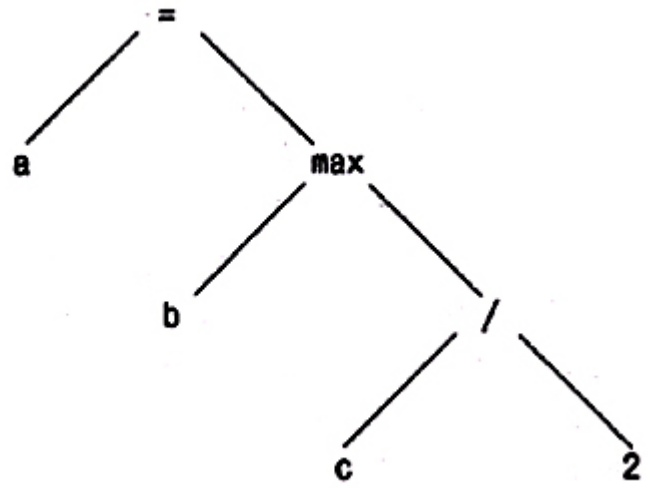
Если язык прост, то какой-то особо большой обработки для определения структуры программы и преобразования ее во внутреннюю форму не требуется. Однако при наличии в языке каких-то сложных элементов — определений, вложенных структур, рекурсивно определяемых выражений, операторов с различным приоритетом и т. п. — провести синтаксический разбор исходного текста для определения структуры становится труднее.

Синтаксические анализаторы часто создаются с помощью специальных автоматических генераторов, называемых также компиляторами компиляторов (compiler-compiler), таких как yacc или bison. Подобные программы переводят описание языка, называемое его грамматикой, как правило, в программу на C или C++, которая, будучи однажды скомпилирована, переводит выражения языка во внутреннее представление. Конечно же, генерация синтаксического анализатора непосредственно из грамматики языка является еще одной впечатляющей демонстрацией мощи хорошей нотации.

Представление, создаваемое анализатором, — это обычно дерево, в котором внутренние вершины содержат операции, а листья — операнды. Выражение

a = max(b, c/2);

может быть преобразовано в такое синтаксическое дерево:



Многие из алгоритмов работы с деревьями, описанных в главе 2, вполне] огут быть применимы для создания и обработки синтаксических деревьев.

После того как дерево построено, обрабатывать его можно множеством способов. Наиболее прямолинейный метод, применяемый, кстати, в Awk, — это прямой обход дерева с попутным вычислением узлов.Упрощенная версия алгоритма такого вычисления для языка, основанного на целочисленных выражениях, может включать в себя восходящий обход типа такого:

```
    typedef struct Symbol Symbol;
 typedef struct Tree Tree; \j
struct Symbol {
 int value;
 char *name; };
struct Tree {
 int op; /* код операции */
 int value; ' /* значение, если это число */
Symbol «symbol; /* имя, если это переменная */
Tree *left;
Tree * right;
 };
/* eval: версия 1: вычисляет выражение,
заданное деревом */
 int eval(Tree *t)
 {
 int left, right;
 switch (t->op) { case NUMBER:
 return t->value; case VARIABLE:
 return t->symbol->value; case ADD:
 return eval(t»left) + eval(t->right); case DIVIDE:
 left = eval(t->left);
 right = eval(t->right);
```

```
 if (right == 0) .
 eprintf("divide %d by zero", left);
return left / right; case MAX:
 left = eval(t->left);
right = eval(t->right);
return left>right ? left : right; case ASSIGN:
 t->left->symbol->value = eval(t->right);
return t->left->symbol->value;
  /*...*/ } }
```

Первые несколько выражений case вычисляют простые выражения вроде констант или значений; следующие вычисляют арифметические выражения, а дальше может идти обработка специальных случаев, условных выражений и циклов. Для реализации управляющих структур нашему дереву потребуется не показанная здесь дополнительная информация, которая представляет собой поток управления.

Подобно тому, как мы делали в pack и unpack, здесь можно заменить явный переключатель таблицей указателей на функции. Отдельные операции при этом будут выглядеть практически так же, как и в варианте с переключателем:

```
    /* addop: суммирует два выражения,
заданных деревьями */
int addop(Tree *t)
 {
 return eval(t->left) + eval(t->right); }
```

Таблица указателей сопоставляет операции и функции, выполняющие операции:

```
    enum { /* коды операций, Tree.op */
NUMBER, VARIABLE, ADD, DIVIDE, /* ... */
 };
/* optab: таблица функций для операций */ int
 (*optab[])(Tree *) = {
pushop, /* NUMBER */
pushsymop, /* VARIABLE */
 addop, /* ADD */
 divop, /* DIVIDE */
 /* ... */
 };
```

Вычисление использует операции для индексирования в таблице указанной на функции для вызова этих функций; в этой версии другие функции вызываются рекурсивно.

```
    /* eval: версия 2: вычисляет выражение */ /*
по таблице операций */ int eval(Tree *t) {
return (*optab[t->op])(t); }
```

Обе наши версии eval применяют рекурсию. Существуют способы устранить ее — в частности, весьма хитрый способ, называемый шитым кодом ireaded code), который практически не использует стек вызовов. Самым ящным способом избавления от рекурсии будет сохранение функций в массиве, с последующим выполнением этих

функций в записанном порядке. Таким образом, этот массив становится просто последовательностью инструкций, исполняемых некоторой специальной машиной.

Для представления частично вычисленных значений из нашего выражения мы, так же как и раньше, будем использовать стек, так что, несмотря на изменение формы функций, преобразования отследить будет трудно. Фактически мы изобретаем некую стековую машину, в которой инструкции представлены небольшими функциями, а операнды хранятся в отдельном стеке операндов. Естественно, это не настоящая машина, но мы можем писать программу так, как будто подобная машина все же существует: в конце концов, мы можем без труда реализовать ее в виде интерпретатора.

При обходе дерева вместо подсчета его значения мы будем генерировать массив функций, исполняющих программу. Массив будет также содержать значения данных, используемых командами, такие как константы и переменные (символы), так что тип элементов массива должен быть объединением:

```
    typedef union Code Code;
 union Code {
void (*op)(void); /* если операция - то функция
 */ int value; /* если число - его значение */ Symbol *symbol; /*
если переменная - то символ */
};
```

Ниже приведен блок кода, генерирующий указатели на функции и помещающий их в специальный массив code. Возвращаемым значением функции generate является не значение выражения — оно будет подсчитано только после выполнения сгенерированного кода, — а индекс следующей операции в массиве code, которая должна быть сгенерирована:

```
     /* generate: обходя дерево, генерирует команды */
 int generate(int codep, Tree *t)
 {
 switch (t->op) { case NUMBER:
 code[codep++].op = pushop;
 code[codep++].value = t->value;
 return codep; case VARIABLE:
 code[codep++]. op = pushsymop;
 code[codep++].symbol = t->symbol;
 return codep; case ADD:
 codep = generate(codep, t->left);
codep = generate(codep, t->right);
code[codep++].op = addop;
 return codep;
case DIVIDE:
 codep = generate(codep, t->left);
codep = generate(codep, t->right);
code[codep++].op = divop;
 return codep; case MAX:
 # ... */ } }
```

Для выражения a - max(b , c/2) сгенерированный код будет выглядеть так:

```
    pushsymop
b
pushsymop
 c
 pushop
2
 divop
maxop
  storesymop
 a
```

Функции-операции управляют стеком, извлекая из него операнды и загружая результаты.

Код интерпретатора организован в виде цикла, в котором командный :счетчик pc обходит массив указателей на фрикции:

```
    Code code[NCODE];
 int stack[NSTACK];
int stackp;
int pc; /* счетчик программы */
 /* eval: версия 3: вычисляет выражение из */ /*
сгенерированного кода */ int eval(Tree *t) {
 pc = generate(0, t);
 code[pc].op = NULL;
 stackp = 0; pc = 0;
 while (code[pc].op != NULL)
(*code[pc++].op)(); return stack[0]: }
```

Этот цикл моделирует в программном виде на изобретенной нами стековой машине то, что происходит на самом деле в настоящем компьютере:

```
    /* pushop:
    записывает число в стек; */
/* значение - следующее слово в потоке code */
 void pushop(void)
{
 stack[stackp++] = code[pc++].value; }
 /* divop: частное двух выражений */
void divop(void)
{
 int left, right;
 right = stack[--stackp]; left = stack[--stackp];
 , if (right == 0)
eprintf("divide %d by zero\n", left);
stack[stackp++] = left / right; }
```

Обратите внимание на то, что проверка делимого на ноль осуществляется в divop, а не в generate.

Условное исполнение, ветвление и циклы модифицируют счетчик программы внутри функции-операции, осуществляя тем самым доступ к массиву функций с какой-то новой точки. Например, операция goto всегда переустанавливает значение

переменной pc, а операция условного ветвления — только в том случае, если условие есть истина.

Массив code является, естественно, внутренним для интерпретатора, однако представим себе, что нам захотелось сохранить сгенерированную программу в файл. Если бы мы записывали адреса функций, то результат получился бы абсолютно непереносимым, да и вообще ненадежным. Однако мы могли бы записывать константы, представляющие функции, например 1000 для addop, 1001 для pushop и т. д., и переводить их обратно в указатели на функции при чтении программы для интерпретации.

Если внимательно посмотреть на файл, который создает эта процеду-, можно понять, что он выглядит как поток команд виртуальной машины. Эти команды реализуют базовые операции нашего рабочего языка, рункция generate — это компилятор, транслирующий язык на виртуальную машину. Виртуальные машины — стародавняя идея, обретшая госледнее время новую популярность благодаря Java и виртуальной шгане Java (Java Virtual Machine, JVM); виртуальные машины предоставляют простой способ создавать переносимые и эффективные реали-, ции программ, написанных на языках высокого уровня.

Программы, которые пишут программы

Возможно, самым примечательным в функции generate является то, о она представляет собой программу, которая пишет другую програму. ее вывод есть поток исполнимых команд для другой (виртуальной) шшны. Собственно, подобная идея нам привычна, именно это делают е компиляторы, только команды они генерируют для реальных машин. На самом деле формы, в которых появляются программы, пишущие программы, очень разнообразны.

Один обычный пример дает динамическая генерация HTML для web-раниц. HTML — это язык, хоть и достаточно ограниченный; кроме то, в себе он может содержать и код JavaScript. Web-страницы часто нерируются "на лету" программами на Perl или C, содержание таких раниц (например, результаты поиска или реклама, нацеленная на определенную аудиторию) зависит от приходящих запросов. Мы использовали специализированные языки для графиков, картинок, таблиц, математических выражений и индекса этой книги. Еще одним примером эжет служить PostScript — язык программирования, тексты на котором создаются текстовыми процессорами, программами рисования и множеством других источников; на финальном этапе обработки книга, которую вы держите в руках, представлена как программа на PostScript, держащая 57 000 строк.

Документ — это статическая программа, однако идея использования языка программирования как способа записи любой проблемы является весьма многообещающей. Много лет назад программисты мечтали о том, что компьютеры когда-нибудь смогут сами писать для себя программы, энечно же, эта мечта никогда не осуществится в полной мере, однако годня можно сказать, что машины нередко пишут программы за нас и иногда в таких областях, где совсем недавно это трудно было себе представить.

Наиболее распространенные программы, создающие программы, — это компиляторы, которые переводят программу с языка высокого уровня в машинный код. Однако нередко оказывается удобным переводить код программы сначала на один из широко известных языков высокого уровня. В предыдущем параграфе мы упоминали о том, что генератор синтаксического анализатора преобразует

определение грамматики языка в программу на C, которая и занимается синтаксическим разбором языка. Язык C достаточно часто используется подобным образом -в качестве "языка ассемблера высокого уровня". Modula-3 и C++ относятся к тем языкам общего назначения, для которых первые компиляторы создавали код на C, обрабатывавшийся затем уже стандартным компилятором. У такого подхода есть ряд преимуществ — одним из главных является эффективность, поскольку получается, что в принципе программа может выполняться так же быстро, как и программы на C. Еще один плюс — переносимость: такие компиляторы могут быть перенесены на любую систему, имеющую компилятор C. Подобный подход сильно помог этим языкам на ранних стадиях их внедрения.

В качестве еще одного примера возьмем графический интерфейс Visual Basic. Он генерирует набор операторов присваивания Visual Basic для инициализации объектов. Этот набор пользователь выбирает из меню и располагает на экране с по'мощью мыши. Во множестве других языков есть "визуальная" среда разработки и "мастера" (wizard), которые синтезируют код пользовательского интерфейса по щелчку мыши.

Несмотря на мощь программ-генераторов и на большое количество примеров их удачного применения, этот подход еще не признан так широко, как он того заслуживает, и нечасто используется программистами. Однако существует уйма простых возможностей программного создания кода, и вы вполне можете использовать их преимущества. Ниже приведены несколько примеров генерации кода на C или C++.

Операционная система Plan 9 генерирует сообщения об ошибках по заголовочному файлу, содержащему программные имена ошибок и комментарии к ним; эти комментарии механически конвертируются в строки, заключенные в кавычки, они помещаются в массив, который может быть индексирован перечислимым значением. В приведенном ниже фрагменте показана структура такого заголовочного файла:

```
    /* errors.h: стандартные сообщения об ошибках
 enum {
Eperm, /* Доступ запрещен */
 Eio, /* Ошибка ввода/вывода */
Efile, /* Файл не существует */
Emem, /* Переполнение памяти */
Espace, /* Нет места для файла */
Egreg /* Гришина ошибка */ };
```

Имея такой фрагмент на входе, несложная программа сможет произнес* и следующий набор деклараций для сообщений об ошибках:

```
    /* machine-generated;
do not edit. */
char *errs[] = {
"Доступ запрещен", /* Eperm */
"Ошибка ввода/вывода", /* Eio */ "Файл не существует",
     /*
Efile */ "Переполнение памяти", /* Emem */
"Нет места для файла", /* Espace */
"Гришина ошибка", /* Egreg */
 };
```

У такого подхода есть несколько достоинств. Во-первых, соотношение лежду значениями enum и строками, которые они представляют, получается самодокументированным, и его нетрудно сделать независимым от родного языка пользователя. Во-вторых, информация хранится только в однтом месте, в одном "истинном месте", из которого генерируется весь эстальной код, так что и всё обновление информации выполняется лишь в одном месте. В случае, когда таких мест несколько, после ряда обновлений они неизбежно начнут друг другу противоречить. И наконец, нетрудно сделать так, чтобы файлы программ на С создавались заново и перекомпилировались при каждом изменении заголовочного файла. Когда потребуется изменить сообщение об ошибке, все, что надо будет сделать, — это изменить заголовочный файл и компилировать таким способом операционную систему, и тогда сообщения автоматически обновятся.

Программа-генератор может быть написана на любом языке. Особенно просто это сделать на языке, специально предназначенном для обработки строк, таком как Perl:

```
      # enum.pl: генерирует строки сообщений no
 enurn + комментарии
  print "/* machine-generated; do not edit.
 */\n\n"; print "char *errs[] = {\n";
 while (<>) I
 # удалить перевод строки
 if (/~\s*(E[a-zO-9]+),?/)
 { » первое слово - E. . . $name = $1; * сохранить имя
 s/ *\\* *//;  УДалить до /*
s/ *\*///;  удалить
print "\t\"$_\", /'•* $name */\n"; } }
 print "};\n";
```

Опять в дело пошли регулярные выражения. Выбираются строки, у которых первое поле выглядит как идентификатор, после которого стоит запятая. Первая замена удаляет все символы до первого непустого символа строки комментария, а вторая — символы конца комментария и все предшествующие им пробелы.

Среди прочих способов для тестирования компилятора Энди Кёниг (Andy Koenig) разработал метод написания кода C++, позволяющий проверить, нашел ли компилятор ошибки в программе. Фрагменты кода, которые должны вызвать реакцию компилятора, снабжаются специальными комментариями, в которых описываются ожидаемые сообщения. Каждая строка такого комментария начинается с /// (чтобы их можно было отличить от обычных комментариев) и регулярного выражения, которое должно соответствовать диагностике компилятора, выдаваемой для этой строки. Таким образом, например, следующие два фрагмента кода должны вызвать реакцию компилятора:

```
    int f() {}
/// warning.* non-void function .* should return a value
void g() {return 1;}
/// error.* void function may not return a value
```

Если мы пропустим второй тест через компилятор C++, то он напечатает ожидаемое сообщение, вполне соответствующее регулярному выражению:

```
    % CC x.c
```

```
    "x.c", line 1: error(321): void function may
not return a value
```

Каждый такой фрагмент кода пропускается через компилятор, и вывод сравнивается с прогнозируемой диагностикой, этим процессом управляет схемная оболочка и программа на Awk. Сбоем считается ситуация, гда вывод компилятора не совпадает с ожидаемым. Поскольку комментарии представлены регулярными выражениями, у них получается которая свобода в оценке вывода: ее можно делать более или менее эогой к отклонениям в зависимости от надобности.

Идея использования семантических комментариев не нова. Такие комментарии используются в языке PostScript, где они начинаются с символа %. Комментарии, начинающиеся с %%, могут содержать дополнительную информацию о номерах страниц, окаймляющем прямоугольнике ounding Box), именах шрифтов и т. п.:

```
    %%PageBoundingBox:
126 307 492 768 %%Pages: 14
%%DocumentFonts:
 Helvetica Times-Italic Times-Roman
LucidaSans-Typewriter
```

В языке Java комментарии, которые начинаются с /** и заканчиваются */, пользуются для создания документации для следующего за ними опи-НЕИЯ класса. Глобальным вариантом самодокументации кода является к называемое грамотное программирование (literate programming), и котором программа и ее документация интегрируются в один документ, и при одной обработке документ готовится для чтения, а при другой программа готовится к компиляции.

Во всех рассмотренных выше примерах важно отметить роль нотации, смерения языков и использования инструментов. Их комбиниро-ние усиливает и подчеркивает мощь отдельных компонентов.

Упражнение 9-15

В программировании давно известна забавная задачка: написать про-шму, которая при выполнении точно воспроизводила бы саму себя — в виде исходного кода. Это такой гипертрофированный случай программы, пишущей программу. Попробуйте выполнить это — на своем любимом языке.

Использование макросов для генерации кода

Опустившись на пару уровней, можно говорить о макросах, которые шут код во время компиляции. На протяжении всей книги мы предостегали вас от использования макросов и условной компиляции, по-зльку этот стиль программирования вызывает множество проблем, щако же они все равно существуют, и иногда текстуальная подстановка — это именно то, что нужно для решения данной проблемы. Один из таких примеров - использование препроцессора C/C++ для компоновки программы с большим количеством повторяющихся частей.

Например, программа из главы 7, оценивающая скорость конструкций простейшего языка, использует препроцессор C для компоновки тестов, составляя из них последовательность стереотипных кодов. Суть теста — заключить фрагмент кода в цикл, который запускает таймер, выполняет этот фрагмент много раз, останавливает

таймер и сообщает о результатах. Весь повторяющийся код заключен в пару макросов, а измеряемый код передается в качестве аргумента. Первичный макрос имеет такой вид:

```
    #define LOOP(CODE) { \
 tO = clock(); \
 for (i=0; i < n; i++) { CODE; } \
 printf("%7d ", ciock() - tO); \
}
```

Обратная косая черта (\) позволяет записывать тело макроса в нескольких строках. Этот макрос используется в "операторах", которые имеют такой вид:

```
    LOOP(f1 = f2)
LOOP(f1 = f2 + f3)
  LOOP(f1 = f2 - f3)
```

Для инициализации иногда применяются и другие операторы, но основная часть, производящая замеры, представлена в этих одноаргументных фрагментах, которые преобразуются в значительный объем кода.

Иногда макросы могут использоваться и для генерации нормального коммерческого кода. Барт Локанти (Bart Locanthi) однажды написал эффективную версию оператора двумерной графики. Этот оператор, называемый bitblt, или rasterop, трудно было сделать быстрым, поскольку он использовал большое количество аргументов, которые могли комбинироваться самыми хитрыми способами. Проведя тщательный разбор вариантов, Локанти уменьшил комбинации до независимых циклов, которые можно было оптимизировать по отдельности. Затем каждый случай был воссоздан с помощью макроподстановки, аналогичной той, что показана в примере на тестирование производительности, и все варианты были перебраны в одном большом выражении switch. Оригинальный исходный код представлял две-три сотни строк, после выполнения макроподстановок он разрастался до многих тысяч строк. Этот конечный код был не самым оптимальным, но, учитывая сложность задачи, весьма эактйчным и простым в написании. И кстати, как и весь код самого высокого уровня, неплохо переносимым.

Упражнение 9-16

В упражнении 7-7 вам предлагалось написать программу, оценивающую траты на различные операции в C++. Используя идеи, изложенные в последнем параграфе, попробуйте написать новую версию этой программы.

Упражнение 9-17

В упражнении 7-8 надо было построить модель оценки затрат для tva, а в этом языке нет макросов. Попробуйте решить эту проблему, написав другую программу — на любом другом языке (или языках), которая создавала бы Java-версию и автоматизировала бы запуск тестов на эоизводительность.

Компиляция "на лету"

В предыдущем разделе мы говорили о программах, которые пишут программы. В каждом примере программы генерировались в виде исходного ада и, стало быть, для

запуска должны были быть скомпилированы или нтерпретированы. Однако возможно сгенерировать код, который можно шускать сразу, создавая машинные инструкции, а не исходный текст, акой процесс известен как компиляция "налету" (on the fly) или "как раз :вовремя" (just in time); первый термин появился раньше, однако последний — включая его акроним JIT — более популярен.

Очевидно, что скомпилированный код по определению получается епереносимым — его можно запустить только на конкретном типе процессора, зато он может получиться весьма скоростным. Рассмотрим такое выражение:

```
max (b, c/2)
```

Здесь нужно вычислить c, поделить его на 2, сравнить результат с b и вы-рать большее из значений. Если мы будем вычислять это выражение, используя виртуальную машину, которую мы в общих чертах описали в начале этой главы, то хотелось бы избежать проверки деления на ноль в divop: поскольку 2 никогда не. будет нулем, такая проверка попросту бессмысленна. Однако ни в одном из проектов, придуманных нами для реализации этой виртуальной машины, нет возможности избежать этой проверки — во всех реализациях операции деления проверка делителя на ноль осуществляется в обязательном порядке.

Вот здесь-то нам и может помочь динамическая генерация кода. Если мы будем создавать непосредственно код для выражения, а не использовать предопределенные операции, мы сможем исключить проверку деления на ноль для делителей, которые заведомо отличны от нуля. На самом деле мы можем пойти еще дальше: если все выражение является константой, как, например, тах (3*3, 2/2), мы можем вычислить его единожды, при генерации кода, и заменять константой-значением, в данном случае числом 9. Если такое выражение используется в цикле, то мы экономим время на его вычисление при каждом проходе цикла. При достаточно большом числе повторений цикла мы с лихвой окупим время, потраченное на дополнительный разбор выражения при генерации кода.

Основная идея состоит в том. что способ записи позволяет нам вообще выразить суть проблемы, а компилятор для выбранного способа записи может специально оптимизировать код конкретного вычисления, Например, в виртуальной машине для регулярных выражений у нас, скорее всего, будет оператор для сравнения с символом:

```
    int matchchar(int literal, char *text)
 {
 return *text == literal;
}
```

Однако, когда мы генерируем код для конкретного шаблона, значение этого literal фиксировано, например ' x ' , так что мы можем вместо показанного выше сравнения использовать оператор вроде следующего:

```
    int matchx(cbar
     *text)
 {
 return *text == 'x';
}
```

И затем, вместо того чтобы предварительно определять специальный оператор для значения каждого символа-литеры, мы можем поступить проще: генерировать код для операторов, которые нам будут действительно нужны для данного выражения. Применив эту идею для полного набора операторов, мы можем написать JIT-компилятор, который будет анслировать заданное регулярное выражение в специальный код, оп-мизированный именно под это выражение.

Кен Томпсон (Ken Thompson) именно это и сделал в 1967 году для реализации регулярных выражений на машине IBM 7094. Его версия генерировала в двоичном коде небольшие блоки команд этой машины для разных операторов выражения, сшивала их вместе и затем запускала шучившуюся программу, просто вызвав ее, совсем как обычную функцию. Схожие технологии можно применить для создания специфических юледовательностей команд для обновлений экрана в графических системах, где может быть так много различных случаев, что гораздо более эффективно создавать динамический код для каждого из них, чем расписать с все заранее или включить сложные проверки в более общем коде.

Демонстрация того, что же включает в себя создание полноценного Т-компилятора, неизбежно вынудит нас обратиться к деталям конкретных наборов команд, однако все же стоит потратить некоторое время, гобы на деле понять, как работает такая система. В оставшейся части этого параграфа для нас важно будет понимание сути, идеи происходящего, а не деталей конкретных реализаций.

Итак, вспомним, в каком виде мы оставили нашу виртуальную машину, — структура ее выглядела примерно так:

```
    Code code[NCODE];
 int stack[NSTACK];
 int stackp;
int pc; /* программный счетчик */
 Tree *t;
t = parse();
pc = generate(0, t);
 code[pc].op = NULL;
stackp = 0; pc = 0;
while (code[pc].op != NULL) (*code[pc++].op)();
 return stack[0];
```

Для того чтобы адаптировать этот код для JIT-компиляции, в него [адо внести некоторые изменения. Во-первых, массив code будет теперь» re массивом указателей на функции, а массивом исполняемых команд.

Будут ли эти команды иметь тип char, int или long — зависит только от того процессора, под который мы компилируем; предположим, что это будет int. После того как код будет сгенерирован, мы вызываем его как функцию. Никакого виртуального счетчика команд программы в новом коде не будет, поскольку обход кода за нас теперь будет выполнять собственно исполнительный цикл процессора; по окончании вычисления результат будет возвращаться — совсем как в обычной функции. Далее, мы можем выбрать — поддерживать ли нам отдельный стек операндов для нашей машины или воспользоваться стеком самого процессора. У каждого из этих вариантов есть свои преимущества; мы решили остаться верными отдельному стеку и сконцентрироваться на деталях самого кода. Теперь реализация выглядит таким образом:

```
    typedef int Code; Code code[NCODE];
  int codep; int stack[NSTACK]; int stackp;
 ...
Tree *t;
 void (*fn)(void);
 int pc;
 *
 t = parse(); pc = generate(0, t);
genreturn(pc); /^генерация последовательности */ /*
 команд для возврата из функции */ staokp = 0;
 flushcaches(); /* синхронизация памяти с
 процессором */ fn = (void(*)(void)) code;
 /* преобразование массива */
/* в указатель на функцию */
(*fn)(); /* вызов полученной функции */
 return stack[0];
```

После того как generate завершит работу, gen return вставит команды, которые обусловят передачу управления от сгенерированного кода к eval.

Функция f lushcaches отвечает за шаги, необходимые для подготовки процессора к запуску свежесозданного кода. Современные машины работают быстро, в частности благодаря наличию кэшей для команд и данных, а также конвейеров (pipeline), которые отвечают за выполнение сразу нескольких подряд идущих команд. Эти кэши и конвейеры исходят из предположения, что код не изменяется; если же мы генерируем этот код непосредственно перед запкуском, топроцессор может оказаться в затруднении: ему нужно обязательно очистить свой конвейер и кэши для исполнения новых команд. Эти операции очень сильно зависят от энкретного компьютера, и, соответственно, реализация f lushcaches будет в каждом случае совершенно уникальной.

Замечательное выражение (void (*)(void)) code преобразует адрес :ассива, содержащего сгенерированные команды, в указатель на функцию, который можно было бы использовать для вызова нашего кода.

Технически не так трудно сгенерировать сам код, однако, конечно, для ого чтобы сделать это эффективно, придется позаниматься инженерной .еятельностью. Начнем с некоторых строительных блоков. Как и раньше, [ассив code и индекс внутри него заполняются во время компиляции. Для [ростоты мы повторим свой старый прием — сделаем их оба глобальными. Затем мы можем написать функцию для записи команд:

```
    /* emit: добавляет команду к потоку кода */
 void emit(Code inst)
 {
code[codep++] = inst; }
```

Сами команды могут определяться макросами, зависящими от процессора, или небольшими функциями, которые собирали бы код, заполняя поля в командном слове инструкции. Гипотетически мы могли бы завести функцию pop reg, которая бы генерировала код для выталкивания значения из стека и сохраняла его в регистре процессора, и функцию push reg, которая бы генерировала код для получения значения, хранящегося в регистре процессора, и заталкивания его в стек. Наша обновленная функция addop будет использовать некие их аналоги, применяя

некоторые предопределенные константы, описывающие команды (вроде ADDINST) и их расположение (различные позиции сдвигов SHIFT, которые определяют формат командного слова):

```
    /* addop: генерирует команду ADD */
 void addop(void)
 {
 Code inst;
popreg(2); /* выборка из стека в регистр 2 */
 popreg(l); /* выборка из стека в регистр 1 */
 inst = ADDINST « INSTSHIFT;
 inst |= (R1) « OP1SHIFT;
inst = (R2) « OP2SHIFT;
 »«m+/i.'ne4-.V /* выпопнить ADD R1, R2 */
 pushreg(2); /* загрузить значение R2 в стек */
  }
```

Это, однако, только самое начало. Если бы мы писали настоящий JIT-ком-пилятор, нам бы пришлось заняться оптимизацией. При прибавлении константы нам нет нужды грузить ее в стек, вынимать оттуда и после этого прибавлять: мы можем прибавить ее сразу. Должное внимание к подобным случаям помогает избавиться от множества излишеств. Однако даже в теперешнем своем виде функция addop будет выполняться гораздо быстрее, чем в наших более ранних версиях, поскольку различные операторы уже не сшиты воедино вызовами функций. Вместо этого код, исполняющий их, располагается теперь в памяти в виде единого блока команд, и для нас все сшивается непосредственно счетчиком команд процессора.

Теперешняя реализация функции generate выглядит весьма похоже на реализацию виртуальной машины, но на этот раз она задает реальные машинные команды вместо указателей на предопределенные функции. Чтобы генерировать более эффективный код, следовало бы потратить усилия на избавление от лишних констант и другие виды оптимизации.

В нашем ураганном экскурсе в генерацию кода мы бросили лишь самый поверхностный взгляд на некоторые из технологий, применяемых в настоящих компиляторах, некоторые же вообще обошли молчанием. При этом мы лишь попутно затронули ряд аспектов, связанных со сложностью организации современных процессоров. Но мы показали, как программа может анализировать описание проблемы, чтобы создать специальный код, который наиболее эффективен для конкретной задачи. Изложенные выше идеи вы можете использовать для создания быстрой версии grep, для реализации небольшого языка программирования имени себя, для проектирования и создания виртуальной машины, оптимизированной для решения специфических задач, или даже для создания компилятора для какого-нибудь интересного языка.

Между регулярным выражением и программой на C++ есть, конечно, немалая разница, но суть у них одна — это всего лишь нотации для решения проблем. При правильной нотации многие проблемы становятся гораздо более простыми. А проектирование и реализация выбранной нотации может дать массу удовольствия.

Упражнение 9-18

JIT-компилятор сгенерирует более быстрый код, если сможет заменить выражения, содержащие только константы, такие как тах(3*3, 4/2), их значением. Опознав такое выражение, как он должен вычислять его значение?

Упражнение 9-19

Как бы вы тестировали JIT-компилятор?

Дополнительная литература

В книге Брайана Кернигана и Роба Пайка "The Unix Programming nvironment" (Brian Kernighan, Rob Pike. The Unix Programming nvironment. Prentice Hall, 1984) широко обсуждается инструментальный подход к программированию, который так хорошо поддерживается! Unix. В восьмой главе ее содержится полная реализация — от грамматики уасс до выполнимого кода — простого языка программирования.

Дон Кнут в своей книге "TEX: The Program" (Don Knuth. TEX: The\ rogram. Addison-Wesley, 1986) описывает этот сложнейший текстовый эоцессор, приводя всю программу целиком — около 13 000 строк на ascal — в стиле "грамотного программирования", при котором пояснения комбинируются с текстом программы, а для форматирования документами и выделения соответствующего кода используются соответствующие эограммы. То же самое для компилятора ANSI C проделано у Криса Фрейзера и Дэвида Хэнсона в книге "A Retargetable C Compiler" (Chris raser, David Hanson. Л Retargetable C Compiler. Addison-Wesley, 1995).

Виртуальная машина Java описана в книге Тима Линдхольма и Франк Еллина "Спецификация виртуальной Java-машины", 2-е издание fim Lindholm, Frank Yellin. The Java Virtual Machine Specification. 2"d ed. ddison-Wesley, 1999).

Кен Томпсон (Ken Thompson) описал свой алгоритм (это один из самых первых патентов в области программного обеспечения) в статье legular Expression Search Algorithm" в журнале Communications of the CM (11, 6, p. 419-422, 1968). Работа с регулярными выражениями весьма подробно освещена в книге Джеффри Фридла "Mastering Regular xpressions" (Jeffrey Friedl. Mastering Regular Expressions. O'Reilly, 1997).

JIT-компилятор для операций двумерной графики описан в статье Эба Пайка, Барта Локанти (Bart Locanthi) и Джона Рейзера CJonn eiser) "Hardware/Software Tradeoffs for Bitmap Graphics on the Blit", опубликованной в Software — Practice and Experience (15, 2, p. 131-152, ;bruary 1985).

Эпилог

Если бы люди могли учиться у истории, какие уроки она могла бы нам преподать! Но страсть и компания слепят наши очи, и свет, даваемый опытом, — как кормовой огонь, освещает только волны позади нас.

Сэмюэль Тейлор Колридж. Воспоминания

Компьютерный мир постоянно изменяется, причем во все возрастающем темпе. Программисты вынуждены овладевать новым: новыми языками, новыми инструментами и новыми системами и, конечно, изменениями, не совместимыми со старым. Программы становятся все объемнее, интерфейсы все сложнее, а сроки все жестче.

Однако есть некоторые неизменные моменты, позволяющие осмыслить уроки прошлого, что может помочь в будущем. Основные вопросы, освещенные в этой книге, базируются как раз на этих неизменных концепциях.

Простота и ясность — первые и наиболее важные среди этих основ, поскольку все остальное так или иначе из них следует. Делайте все самым простым способом. Выбирайте наиболее простой алгоритм, который выглядит достаточно быстрым, и наиболее простую структуру данных, позволяющую решить поставленную задачу; объединяйте их простым, понятным кодом. Не усложняйте ничего до тех пор, пока в этом не возникнет настоятельной необходимости, — например, пока замеры производительности не покажут неудовлетворительных результатов. Интерфейсы должны быть ограничены и лаконичны, по крайней мере до тех пор, пока не станет совершенно очевидно, что преимущества от нововведений перевесят недостатки дополнительного усложнения.

Универсальность нередко идет рука об руку с простотой, поскольку является, по сути, возможностью решить проблему раз и навсегда (так сказать, "в общем виде"), а не возвращаться к ней снова и снова для рассмотрения специфических случаев. Зачастую это является и хорошим подспорьем для обеспечения переносимости: лучше найти одно общее решение, которое будет работать во всех системах, чем разбираться в отличительньгх особенностях каждой системы.

Далее стоит назвать эволюцию. Невозможно написать совершенную трограмму с первой попытки. Глубокое и всестороннее осознание проН элемы рождается только из сочетания размышлений и опыта; с помощыо чисто умозрительных заключений не удастся создать хорошей сиетемы, как не удастся сделать этого и чистым хакерством. Очень важна реакция пользователей; наиболее эффективным будет цикл развития системы, включающий в себя создание прототипа, экспериментирование, обратную связь с пользователем и дальнейшие усовершенствования. Программы, которые мы пишем для себя, часто не развиваются; достаточным образом; большие программы, которые мы покупаем у других, изменяются слишком быстро без нужного улучшения.

Интерфейсы являются одним из камней преткновения в программировании, и аспекты, связанные с ними, проявляются во многих местах. Наиболее очевидный случай — библиотеки, но существуют еще интерфейсы между программами и между пользователями и программами. В проектировании интерфейсов идеи простоты и универсальности имеют особое значение. Делайте интерфейсы как можно более

последовательными, чтобы их можно было без труда выучить и использовать; к этому надо подойти очень тщательно. Эффективным способом является абстрагирование: представьте себе идеальный компонент, библиотеку или программу; постарайтесь, чтобы ваш интерфейс как можно более полно соответствовал такому идеалу; спрячьте детали реализации — от греха подальше.

Роль автоматизации часто недооценивается. Гораздо эффективнее заставить компьютер выполнять вашу работу, чем делать ее вручную. Мы увидели примеры применения автоматизации при тестировании, отладке, анализе производительности и, что особенно важно, в написании кода, когда для решения конкретной задачи программы могут писать программы, которые было бы трудно написать вручную.

Нотация также зачастую недооценивается; и не только в качестве способа, которым программист сообщает компьютеру, что надо сделать,). Способ записи представляет собой основу для реализации широкого: спектра инструментов, а также обусловливает структуру программы, предназначенной для написания программ. Мы все хорошо владеем большими языками общего назначения, которые служат нам для создания основной части наших программ. Однако, как только задания становятся настолько специализированными и хорошо понятными, что запрограммировать их можно почти механически, стоит задуматься о том, чтобы создать нотацию, в которой задание выражалось бы естественно, и язык, реализующий эту нотацию. Регулярные выражения — один из наших любимых примеров, но возможностей создания рабочих языков для специализированных приложений существует бесчисленное множество. Для того чтобы дать хорошие результаты, языки эти вовсе не должны быть сложны.

У рядового программиста может иногда возникнуть ощущение, что он является винтиком в огромной машине: ему приходится использовать навязанные ему языки, системы и инструменты, выполняя задания, которые должны были бы быть уже сделаны для него и за него. Однако, по большому счету, наша работа оценивается по тому, как много мы можем сделать с помощью того, что у нас есть. Применяя идеи, изложенные в этой книге, вы обнаружите, что с вашим кодом стало удобнее работать, отладка стала гораздо менее болезненным процессом, да и в своих программах вы стали более уверены. Мы надеемся, что эта книга дала вам что-то, что сделает программирование более продуктивным и успешным для вас.

Приложение: свод правил

- Стиль
- Интерфейсы
- Отладка
- Тестирование
- Производительность
- Переносимость

> Каждая открытая мной истина становилась правилом, которое служило мне в дальнейшем для поиска 'других истин.
>
> Рене Декарт. Рассуждение о методе

Во многих главах были выделены правила или какие-то основные моменты, подводящие итог обсуждению. Для удобства поиска правила собраны здесь воедино. Не забывайте, что в соответствующих частях книги объясняется назначение и способы применения этих правил.

Стиль

Используйте осмысленные имена для глобальных переменных и короткие — для локальных.
Будьте последовательны.
Используйте активные имена для функций.
Будьте точны.
Форматируйте код, подчеркивая его структуру.
Используйте естественную форму выражений.
Используйте скобки для устранения неясностей.
Разбивайте сложные выражения.
Будьте проще.
Будьте осторожны с побочными эффектами.
Будьте последовательны в применении отступов и фигурных скобок.
Используйте идиомы для единства стиля.
Используйте else-if для многовариантных ветвлений.
Избегайте макрофункций.
Заключайте тело макроса и аргументы в скобки.
Давайте имена загадочным числам.
Определяйте числа как константы, а не как макросы.
Используйте символьные константы, а не целые.
Используйте средства языка для определения размера объекта.
Не пишите об очевидном.
Комментируйте функции и глобальные данные.
Не комментируйте плохой код, а перепишите его.
Не противоречьте коду.
Вносите ясность, а не сумятицу.

Интерфейсы

Прячьте детали реализации.
Ограничьтесь небольшим набором независимых примитивов.

Не делайте ничего "за спиной" у пользователя.
Всегда делайте одинаковое одинаково.
Высвобождайте ресурсы на том же уровне, где выделяли их.
Обнаруживайте ошибки на низком уровне, обрабатывайте на высоком.
Используйте исключения только для исключительных ситуаций.

Отладка

Ищите знакомые ситуации.
Проверьте самое последнее изменение.
Не повторяйте дважды одну и ту же ошибку.
Не откладывайте отладку на потом.
Пользуйтесь стеком вызова.
Читайте код перед тем, как исправлять.
Объясните свой код кому-либо еще.
Сделайте ошибку воспроизводимой.
Разделяй и властвуй.
Изучайте нумерологию ошибок.
Выводите информацию, локализующую место ошибки.
Пишите код с самоконтролем.
Ведите журнальный файл.
Постройте график.
Используйте инструменты.
Ведите записи.

Тестирование

Тестируйте граничные условия кода.
Тестируйте пред- и постусловия.
Используйте утверждения.
Используйте подход защитного программирования.
Проверяйте коды возврата функций.
Тестируйте по возрастающей.
Тестируйте сначала простые блоки.
Четко опредеоите, чего вы хотите на выходе текста.
Проверяйте свойства сохраности данных.
Сравните независимые версии.
Оценивайте охват тестов.
Автоматизируйте возвратное тестирование.
Создайте замкнутые тесты.

Производительность

Автоматизируйте замеры времени.
Используйте профилировщик.
Концентрируйтесь на критических местах.
Постройте график.
Улучшайте алгоритм и структуру данных.
Используйте оптимизацию компилятора.
Выполните тонкую настройку кода.
Не оптимизируйте то, что не имеет значения.
Объединяйте общие выражения.

Замените дорогостоящие операции на более дешевые.
Избавьтесь от циклов или упростите их.
Кэшируйте часто используемые значения.
Напишите специальную функцию захвата
памяти (аллокатор).
Буферизуйте ввод и вывод.
Специальные случаи обрабатывайте отдельно.
Используйте предварительное вычисление результатов.
Используйте приближенные значения.
Перепишите код на языке более низкого уровня.
Используйте минимально возможный тип данных.
Не храните то, что можете без труда вычислить.

Переносимость

Придерживайтесь стандарта.
Следуйте основному руслу.
Избегайте неоднозначных конструкций языка.
Попробуйте несколько компиляторов.
Используйте стандартные библиотеки.
Используйте только то, что доступно везде.
Избегайте условной компиляции.
Выносите системные различия в отдельные файлы.
Прячьте системные различия за интерфейсами.
Используйте текст для обмена данными.
Используйте при обмене данными
фиксированный порядок байтов.
При изменении спецификации изменяйте и имя.
Поддерживайте совместимость с существующими
программами и данными.
Не рассчитывайте на ASCII.
Не ориентируйтесь только на английский язык.