Содержание

Вместо предисловия

Несмотря на то что почти всякая книга по программированию начинается с предисловия, оно почти всегда пишется последним. Для автора книги это последний шанс оправдать (хотя бы для самого себя) и мелкость темы, и убогость стиля, и отсутствие литературного таланта.

Будем честными: предисловия пишут не для того, чтобы их читали. Это единственное место, где можно попытаться убедить потенциального читателя потратить свое время и свои деньги на книгу до того, как он поймет, что ошибся. Косвенно переложив тем самым ответственность за выбор на него.

Автор убежден, что это не имеет ничего общего с моралью, и поэтому отказывается от ложных оправданий и пустых обещаний. Да, он писал книгу не без удовольствия, но вряд ли это может служить весомой рекомендацией. Единственный совет, который он может предложить, – обратиться к оглавлению, выбрать любой раздел и попробовать почитать. Первое впечатление обычно и самое верное.

Впрочем, одно достоинство у предисловия все-таки есть. В нем автор может выразить свою благодарность тем, кто вдохновил его на написание книги, кто помогал ему дельным советом или добрым участием, от кого он получал поддержку.

Не воспользоваться этой возможностью было бы верхом черствости. Автор признателен своей семье, в особенности великолепному Ламику, но, конечно, больше всех – крошке Ру.

Введение

Программирование богато алгоритмами и структурами данных, и современный программист должен владеть большинством из них. Или, по крайней мере, иметь о них ясное представление.

Да, нельзя объять необъятного. И все же есть некий минимум, которым, на наш взгляд, должен владеть всякий программист, чтобы считаться профессионалом.

В физике известен второй закон термодинамики. В самой примитивной формулировке он постулирует невозможность вечного двигателя: какие бы ухищрения нами ни предпринимались, невозможно направить всю имеющуюся энергию на полезную работу; часть (и порой значительная часть) энергии неизбежно будет теряться. Конечно, закона сохранения энергии никто не отменял: сама по себе энергия никуда не пропадает, она переходит в другие формы и рассеивается в виде тепла. Именно поэтому КПД любого двигателя всегда строго меньше единицы. Энергия, которой принципиально нельзя воспользоваться, увеличивает хаос во Вселенной. Количественной мерой этого хаоса (или неупорядоченности) в термодинамике служит энтропия. Задача инженеров – уменьшить энтропию, снизить, насколько можно, потери, не дать этой рассеянной энергии еще больше увеличить хаос.

Программирование – это дисциплина на стыке математики и инженерии. Как математики мы, программисты, призваны решать сложные задачи, переводить их на язык чисел и символов. Как инженеры – воплощать решения в форме программ, которые могут быть выполнены машиной. Кроме того, мы должны обеспечивать эффективность выполнения этих программ. Эффективность – это не только скорость, с которой задача будет решена, но также и объем занимаемой памяти. Разумеется, к инженерным «заботам» следует отнести масштабируемость, расширяемость, устойчивость, возможность внесения изменений и проч.

Математик работает так, как будто у него в запасе вечность и неограниченные ресурсы. Если для решения задачи программе придется проработать пусть даже миллион лет, математик будет считать свою миссию выполненной. Инженер, напротив, скован массой технических и экономических ограничений. Миллион лет его никак не устраивает:

хорошо – час, может быть – день, в крайнем случае – месяц. Если инженер не может обеспечить этого, то либо задача признается (в данных условиях) нерешенной, после чего она возвращается математику для дальнейших раздумий, либо возникают сомнения, и часто обоснованные, в том, что инженер вообще способен это сделать.

Программист – это, как правило, немного математик, а в основном инженер (многие, очень многие из тех, кто считает себя программистами, совсем не понимают самой необходимой математики; может, это и чересчур, но вряд ли имеют право так себя именовать). И большую часть времени программист занимается второй, т. е. инженерной, стороной деятельности.

Без надежных и удобных инструментов любая деятельность, а инженерная – в особенности, неэффективна и даже невозможна. Математика снабжает нас идеями, концепциями и рецептами (в программировании обычно говорят об алгоритмах), но именно инструментарий делает нас сильными, позволяет воплощать идеи, концепции и рецепты в работающие программы. У хорошего мастера, в какой бы области он не трудился, есть свой набор инструментов: у любого плотника мы найдем пилу, рубанок и стамеску; электрик не может обойтись без отвертки, плоскогубцев и изоленты, а строителю необходимы лопата, мастерок и рулетка. А всем им необходим молоток.

Программисту тоже необходимы инструменты. Понятно, первый из них – компьютер. Но без программ компьютер бесполезен (разве что обогревать пространство вокруг себя, увеличивая энтропию Вселенной). А программы – это не только машинные коды, способные выполняться процессором, но и данные, обычно организованные особым образом. Если программист действительно хочет быть профессионалом, а не поденщиком, он должен иметь ясное представление о способах представления данных, понимать, когда их применять, и уметь ими пользоваться.

Одной из таких структур данных – стеку – и посвящена эта небольшая книга. Стек давно известен и заслуженно пользуется уважением; без него сегодня немыслимо написать ни одной программы. Порой стек используется явно, но куда чаще он действует скрытно, в глубине. И надо признать, именно эта – неочевидная – часть возможностей остается для многих программистов terra incognita. Очень жаль...

Программисты в своей ежедневной работе полагаются на операционные системы, средства разработки, языки программирования. Это – часть их инструментария, в котором также используется стек. Конечно, можно было бы сказать: «Для чего мне знать что-то еще? До-

статочно того, что это работает и этому можно доверять». Да, такой взгляд имеет право на существование. До тех пор, пока мы решаем типовые задачи (а для большинства такие задачи – единственное, чем им приходится заниматься всю профессиональную карьеру), большего и не нужно. Но программирование неизмеримо богаче. В нем полным-полно задач нерешенных, решенных частично либо решенных плохо и ждущих своего часа. Стоит ли сознательно обеднять себя и обманываться тем, что можно обойтись имеющимся? Или ждать, когда кто-то более умный и отважный сделает то, что считалось невыполнимым?

Конечно, каждый решает этот вопрос по-своему. Но отвергнуть, даже не попробовав, никуда не годится.

Изложение в книге построено следующим образом. Стек не возникает неизвестно как, откуда и для чего. В книге рассматривается ряд задач нарастающей сложности. Все эти задачи рано или поздно приводят к стеку; таким образом, всякий раз появление стека подготавливается и обосновывается. Для того чтобы читатель мог составить себе наиболее полное представление о стеке и его возможностях, часто приводится решение исходной задачи другими способами. Такое сравнение всегда полезно, поскольку позволяет оценить достоинства и недостатки различных подходов и выбрать из них лучший.

Книгу лучше читать по порядку, поскольку последующий материал обычно опирается на предыдущий. Там, где это представлялось полезным, более ранний материал частично повторяется; это избавляет от необходимости возвращаться назад и позволяет поддерживать определенный ритм.

Структурно книга состоит из трех частей. Первая часть – основная. Именно здесь ставятся, обсуждаются и решаются задачи. Вторая часть посвящена описанию небольшой, но достаточно мощной, стековой машины и программированию для нее. Третья часть – это приложения, среди которых – исходные коды транслятора и интерпретатора стековой машины, описанной во второй части.

Завершается книга библиографией. Мы приложили все усилия к тому, чтобы перечислить в ней действительно ценные (на наш взгляд) и в то же время доступные источники, относящиеся к основной теме книги.

Эти источники позволят заинтересованным читателям двигаться дальше – к настоящим высотам.

В ежедневной практике программиста стек в явном виде встречается лишь иногда; большей частью стек «трудится» незаметно. Чаще всего он напоминает о себе тогда, когда работа программы внезапно

прекращается и на экран (либо в консоль) выводятся малопонятные непосвященному строчки сообщений, состоящие из адресов памяти и имен функций, вызовы которых привели к остановке. Как правило, такие сообщения свидетельствуют о том, что в программах имеются ошибки. Эти ошибки обычно невозможно отследить на этапе трансляции программы; они проявляют себя во время исполнения. Многие (возможно, большинство) не понимают ни причин их возникновения, ни того, что они означают. Прочитав книгу, вы начнете понимать, отчего это происходит и что нужно делать, чтобы программы стали более надежными и устойчивыми.

Основная область применения стеков – трансляция языков программирования и поддержка сред исполнения. Эти области информатики считаются сложными, и это действительно так. Но даже самые сложные вещи состоят из простых. Вот этими, относительно простыми вещами мы займемся и коснемся многого, что непосредственно используется при реализации языков программирования и работы программ после их запуска на исполнение. Если когда-нибудь вам придется (или захочется) реализовать язык программирования, то будьте уверены – стек станет вашим главным помощником.

И последнее. Стек в качестве главного «героя» был выбран не случайно. Он по-своему красив. Правда, для того чтобы это оценить, придется потрудиться. Нельзя понять музыку, читая рецензии; музыку надо слушать. То же и со стеком – с ним надо поработать. В основном тексте книги почти нет привычных задач и упражнений; упражнением в известном смысле является весь текст. Если читатель попробует свои силы в реализации задач, которые рассматриваются в книге, – это лучшее, чего можно пожелать. Польза будет несомненной.

Кое-где в тексте в качестве иллюстрации тех или иных понятий встречаются фрагменты программ, написанных на языке программирования Java. Все они очень просты, и даже если читатель незнаком с Java, мы уверены, что он без труда их поймет.

Мы нигде не используем примечаний: ни подстрочных, ни вынесенных в отдельный раздел. Многие склонны их рассматривать как малозначительные отступления и обычно пропускают. Кроме того, такие примечания прерывают процесс чтения и вынуждают отвлекаться от основной темы. Мы сделали примечания частью текста, *выделяя их курсивом*. Не стоит их игнорировать – в них много полезной информации.

Часть I

Задачи, приводящие к стеку

Скобочные структуры: элементарный случай

Мы начнем с того, что рассмотрим одну простую задачу, которая послужит отправной точкой всего последующего изложения. Эта задача встречается в разнообразных вариантах (и чуть позже мы расскажем об этом подробнее), но нам пока вполне достаточно такой формулировки: определить – правильно ли расставлены скобки в произвольном алгебраическом выражении. Несколько примеров даст представление о том, что мы имеем в виду и какое решение ищем:

```
a + (b * c /(d – e))
a + (b * c /d – e)
(a + b – c * (d / (e + f)))
```

Если всем входящим в эти выражения переменным (`a`, `b`, ...) придать определенные числовые значения, то эти выражения могут быть вычислены по правилам элементарной арифметики (конечно, исключая случаи, когда знаменатели дробей обратятся в 0). В этих примерах выражений скобки расставлены правильно. Чтобы сделать последнее утверждение более наглядным, давайте оставим только скобки, убрав все «лишнее»; в результате у нас получится следующее:

```
(())
()
(((())))
```

Такого рода выражения будем называть *скобочными структурами*; мы полностью игнорируем все то, что содержится внутри скобок, и сосредоточиваем внимание только на скобках и их взаимном расположении. Все приведенные выше скобочные структуры, очевидно, корректны, т. е. сформированы правильно. Теперь приведем несколько примеров некорректных (неправильно сформированных) скобочных структур:

```
(()
())
)(
```

Давайте немного остановимся на последних примерах и проанализируем, что именно делает эти скобочные структуры некорректными (ошибочными или неправильно сформированными). Для этого удобно называть левые скобки «(» открывающими, а правые «)» – закрывающими.

В первом примере число открывающих скобок больше числа закрывающих. Во втором примере ситуация противоположная – число открывающих скобок меньше числа закрывающих. В таких случаях говорят, что скобки *не сбалансированы* по числу вхождений. Третий пример демонстрирует еще один случай: число открывающих скобок совпадает с числом закрывающих (т. е. скобки сбалансированы по числу вхождений), но скобочная структура тем не менее ошибочна – выражение не может начинаться с закрывающей скобки или заканчиваться открывающей.

Наша ближайшая задача – найти способ (алгоритм) проверки любой заданной скобочной структуры, состоящей только из скобок одного типа (в данном случае скобочной структуры, состоящей только из круглых скобок).

Очевидно, что существует бесконечное количество как правильно, так и неправильно сформированных скобочных структур. Поэтому попытка описать все возможные варианты скобочных структур их простым перечислением не осуществима даже теоретически – рано или поздно встретится такая скобочная структура, которая не была нами предусмотрена. В таком случае программа, осуществляющая проверку скобочной структуры, будет не в состоянии определить ее правильность. Следовательно, нужен иной способ. Такой способ (и очень простой) существует, но, перед тем как описывать его, мы предлагаем читателю немного подумать и попытаться найти его самостоятельно.

Если последовательность скобок начинается с закрывающей или завершается открывающей скобкой (наш третий пример), то, бесспорно, такая скобочная структура сформирована неправильно. Но эти два предельных случая не позволяют решить задачу в общем виде, т. к. первые примеры неправильно сформированных скобочных структур начинаются и завершаются нужными видами скобок – соответственно «(» и «)», но которые тем не менее ошибочны.

Итак, в решении задачи мы пока почти никуда не продвинулись. Но давайте посмотрим на проблему чуть иначе. Отвлечемся на несколько минут от скобок. Представьте лифт. Во избежание аварии необходимо контролировать массу груза (включая людей), перевозимого лифтом. Для этого в лифт встраивается специальный датчик. Каждый внесенный груз или входящий человек увеличивает общую массу груза в кабине и нагрузку на тросы и лебедку лифта. Разумеется, показания датчика увеличиваются. Каждый выходящий из кабины лифта уменьшает массу груза в кабине, и показания датчика уменьшаются.

Наш «груз» – это два вида скобок. Открывающая скобка увеличивает «массу», а закрывающая – уменьшает. Конечно, как и всякая аналогия, наш пример страдает неточностями и упрощениями, но он тем не менее наводит на идею.

Припишем каждой скобке некий «вес»: открывающей скобке – положительное число (скажем, 1), а закрывающей – число, равное по абсолютной величине, но противоположное по знаку (т. е. –1). Датчиком пусть будет целочисленная переменная (назовем ее, например, count) с начальным значением, равным 0 (в примере с лифтом это означает, что изначально кабина пуста). Эта переменная будет играть роль счетчика. Промоделируем поведение нашей «системы» на самой простой скобочной структуре: `()`. Для компактности будем представлять отдельные состояния в виде таблицы, в левой колонке которой указывается уже прочитанная часть скобочной структуры, а в правой (в той же строке) – значение счетчика. Получаем следующий протокол разбора:

```
Прочитанная часть   Счетчик
                       0
(                      1
()                     0
```

В самом начале (когда еще ни одна скобка не прочитана) счетчик равен 0. После обнаружения открывающей скобки соответствующее

ей число (т. е. 1) прибавляется к счетчику. После обнаружения закрывающей скобки соответствующее ей число (т. е. –1) также прибавляется к счетчику. Ясно, что итогом арифметической операции 1 + (–1) является 0 (лифт пуст). Если скобочная структура сформирована правильно, то значение счетчика по окончании разбора будет равно 0. Для закрепления (и дополнительной проверки) разберем другую скобочную структуру (()). Вот протокол разбора:

```
Прочитанная часть    Счетчик
                       0
(                      1
((                     2
(()                    1
(())                   0
```

Похоже, что для правильно сформированных скобочных структур этот метод работает. А что можно сказать о неправильно сформированных? Рассмотрим это на примере скобочной структуры (():

```
Прочитанная часть    Счетчик
                       0
(                      1
((                     2
(()                    1
```

Вся скобочная структура прочитана, а значение счетчика положительно (в нашем случае равно 1). Если обратиться к аналогии с лифтом, то это означает, что кто-то вошел в лифт и не выходит из него. Из этого немедленно следует, что число открывающих скобок в скобочном выражении больше числа закрывающих и скобочное выражение в целом сформировано неверно. Противоположный случай ()) дает следующий результат:

```
Прочитанная часть    Счетчик
                       0
(                      1
()                     0
())                    -1
```

Вся скобочная структура прочитана, а значение счетчика отрицательно (в нашем случае равно –1). Из этого немедленно следует, что число открывающих скобок в скобочном выражении меньше числа закрывающих и скобочное выражение сформировано неверно (из лифта вышло больше людей, чем в него вошло, что, конечно, невозможно). Наконец, рассмотрим случай)(:

```
Прочитанная часть    Счетчик
                       0
)                      -1
```

Как только значение счетчика станет отрицательным, то оставшуюся часть скобочной структуры можно не проверять (если лифт пуст, то из него некому выходить) – она заведомо ошибочна, и, следовательно, вся эта скобочная структура сформирована неверно. Необходимо сразу прекратить разбор и отвергнуть эту скобочную структуру как недопустимую. Почему нужно прекратить разбор, не проверяя следующих скобок? Во-первых, это бессмысленно, а во-вторых, открывающая скобка установит значение счетчика в 0, и мы можем решить, что такая скобочная структура допустима, хотя, очевидно, это не так. В качестве легкого упражнения проведите разбор следующей скобочной структуры: `(()))`. Первые четыре скобки установят значение счетчика в 0, но последняя (закрывающая скобка) лишняя, и все скобочное выражение должно быть отвергнуто. Кстати, продолжая, что можно сказать о скобочных структурах `(()())`, `(())()` или `(()()`?

Итак, задача решена. Для скобок одного типа (в нашем примере круглых) существует простой и эффективный алгоритм проверки структуры скобок.

Тип скобок может быть и иным, например фигурные `{}`, угловые `<>` или квадратные `[]`. Более того, помимо указанных скобок, в языках программирования используются и другие конструкции, которые могут быть отнесены к скобкам, например begin-end или try-catch (правда, обычно такого рода конструкции называют блоками, но суть дела от этого принципиально не меняется). По-существу, скобками являются и многострочные комментарии вида `/* ... */`. Иными словами, не важно, что мы называем скобками и как они выглядят; скобкой может быть все, что выполняет функции скобки.

Последнее утверждение служит иллюстрацией т. н. «утиного теста»: если нечто похоже на утку, плавает как утка и крякает как утка, то это, скорее всего, утка. Иначе говоря, нет необходимости связывать себя только общепринятыми типами скобок; важна сущность, выполняемые функции.

Простые скобочные структуры, подобные рассмотренным только что, составляют лишь небольшую часть синтаксических структур в языках программирования. Обычно встречаются более сложные скобочные структуры, т. е. скобочные структуры, включающие разные типы скобок. Вот небольшой пример:

```
{
int [] keys;
ArrayList <String> aList;
...
for (int i = 0; i < keys.length(); i++) {
        aList.get (i) = “”;
    }
}
```

Перед нами фрагмент исходного кода (на языке программирования Java, но подобный пример можно встретить во многих других языках), в котором встречаются все четыре «стандартных» типа скобок. Задача все та же: определить, правильно ли сформирована скобочная структура в том случае, когда типов скобок не один, а несколько. Давайте, как и в первом случае, уберем все «лишнее» и оставим только скобки. Вот что у нас получится:

```
{[]<>(()){()}}
```

Как убедиться в правильности скобочной структуры такого вида? Сразу же приходит в голову мысль воспользоваться только что разработанным методом подсчета, приписывая каждой открывающей и каждой закрывающей скобке некий «вес», т. е. определенное число, характеризующее вид скобки – открывающая или закрывающая. Пусть, как и прежде, любой открывающей скобке соответствует 1, а любой закрывающей –1. Для нашего примера после просмотра скобок получится 0 (и при этом счетчик никогда не станет отрицательным), т. е. мы можем сделать вывод, что скобочная структура правильна.

Итак, задача решена? Как бы не так! Посмотрим на такую, например, скобочную структуру: `{(})`. Наш алгоритм даст в результате значение 0, при этом значение счетчика никогда не будет отрицательным. Но эта скобочная структура, совершенно очевидно, сформирована неверно. Вывод: алгоритм со счетчиком не работает. Хуже всего то, что он выдает результат, который выглядит правильным (т. к. счетчик по окончании разбора действительно равен 0), но которому нельзя верить. Надо понять – почему он не работает и можно ли его исправить так, чтобы он был пригоден в новой ситуации с различными типами скобок.

Вообще говоря, причина лежит на поверхности: мы увеличиваем (соответственно, уменьшаем) значение счетчика всякий раз, когда в последовательности встречается открывающая (закрывающая)

скобка. В примере {(}) мы увеличиваем значение счетчика последовательно дважды: сначала, когда встречается скобка {, и потом, когда встречается скобка (. Но ведь это разные типы скобок! Не тут ли корень проблемы? Нельзя ли изменить способ изменения значений счетчика так, чтобы он соответствовал типу скобки? Можно, конечно, и будет правильно, если читатель попробует поискать решение в этом направлении.

Методы анализа скобочных структур стали исследоваться еще в 50-х годах XX века. Если в выражении встречаются скобки только одного вида, то задача, как мы уже видели, решается элементарно. Но по мере усложнения и развития языков программирования скобочные структуры стали включать в себя скобки разных типов, и проблема стала актуальной. Был предложен ряд методов решения (в их числе весьма оригинальные), но все эти методы были сложными в реализации и медленными в работе. Если читатель уже попытался найти такой метод, он, безусловно, должен был убедиться, что проблема нетривиальна. Можно вводить отдельные счетчики для каждого вида скобок, но проблема согласования вложенности одних типов скобок в другие все равно оставалась. Метод со счетчиком, при всей его элегантности, для составных скобочных структур работает неудовлетворительно. Нужно что-то иное, и сейчас мы переходим к обсуждению метода, совершенно отличного от метода подсчета с использованием счетчика.

Вернемся еще раз к нашему примеру {(}). Он, как мы знаем, соответствует недопустимой скобочной структуре. Проанализируем скобки по порядку. Первой встречается открывающая { скобка. Чтобы скобочная структура стала допустимой, где-то в последовательности скобок должна быть закрывающая } скобка. Она, конечно, присутствует (в примере это третья скобка слева). Но, прежде чем мы до нее «доберемся», надо что-то сделать с другой открывающей скобкой, только теперь это (. Для того чтобы (третья по порядку) закрывающая фигурная скобка } соответствовала первой открывающей {, нужно обеспечить для открывающей круглой скобки парную ей закрывающую. Принято говорить, что круглая скобка (вложена в фигурную {, или, чуть более формально, что у открывающей круглой скобки *глубина вложенности* больше, чем у открывающей фигурной. Прежде чем закрывать внешние скобки (в данном случае { и }), нужно, чтобы были закрыты все внутренние. В нашем примере как раз это очевидное правило и не соблюдается: фигурная скобка }, относящаяся к паре с меньшей глубиной вложенности (внешней), появляется

раньше круглой), которая относится к скобкам большей глубины вложенности (внутренней).

Может быть, небольшой пример поможет лучше понять эти довольно абстрактные рассуждения. Представьте себе закрытую шкатулку, внутри которой лежит закрытый конверт. Нам нужно прочесть содержимое конверта, т. е. лежащее в нем письмо, а затем все вернуть в исходное состояние. Шкатулка соответствует скобкам { и }, а конверт – скобкам (и). Прежде чем мы доберемся до конверта, надо открыть шкатулку. Этому соответствует скобка {. Далее мы должны открыть конверт, т. е. скобку (. После этого мы достаем письмо, читаем его, прячем обратно в конверт и закрываем конверт; этому соответствует, разумеется, закрывающая) скобка. Наконец, мы закрываем шкатулку, используя скобку }. Результат, очевидно, такой: {()}, и это правильно сформированная скобочная структура с правильным порядком вложенности. Наш начальный пример {(}) соответствует ситуации, которую можно описать так: открыть шкатулку, открыть конверт, достать письмо, закрыть шкатулку, закрыть конверт. Но как раз последнее действие выполнить и невозможно – шкатулка закрыта, и до конверта, лежащего в ней, добраться уже невозможно!

Работая над этой задачей, мы после некоторого размышления рано или поздно приходим к выводу о том, что решение, основанное на использовании счетчика (или даже нескольких счетчиков), по-видимому, следует отбросить. Счетчик хорошо работал для первого случая, когда у нас были скобки только одного типа. Его было вполне достаточно для хранения всей информации об уже просмотренном участке скобочной структуры; последующие скобки только модифицировали состояние счетчика, но не меняли существа обрабатываемой информации – тип скобки оставался прежним, и мы могли его игнорировать. Как только типов скобок стало больше, метод перестал работать, он подошел к границам применимости. Поступающая информация уже не может быть сохранена только с использованием счетчиков – информации не только стало больше, сама информация стала другой.

Метод с использованием счетчика прост и понятен: нам привычно что-то подсчитывать. Но всему наступает предел, за который надо выходить.

А теперь, после этого несколько затянувшегося описания причин безуспешных попыток анализа скобочной структуры с различными типами скобок, мы переходим к главному действующему лицу – стеку. Но, прежде чем мы продолжим, нам придется ненадолго прерваться.

Следующие несколько страниц будут очень важными, поскольку мы собираемся рассказать, что такое стек, опишем относящуюся к стеку терминологию, введем обозначения, которые позволяют избавиться от многословных описаний, и, наконец, расскажем о некоторых основных операциях со стеком. Возможно, не сразу будет понятно – для чего мы на время отложили нашу задачу, но потерпите. Все это необходимо будет тщательно изучить, так что давайте приступим.

Стек: знакомство

Обратимся к классике и позаимствуем определение стека у Дональда Кнута:

Стек – это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются только на одном из концов списка.

Для знатока структур данных этого, скорее всего, будет достаточно. Но все-таки давайте уделим определению стека некоторое время. Самое важное в этом определении – последние слова, а именно «только на одном из концов». Наверное, проще всего проиллюстрировать, что такое стек, – привести несколько примеров, которые всем, надеемся, будут знакомы и понятны.

Первый пример стека мы обнаруживаем в детской игрушке. Всем, вероятно, знакома детская пирамидка: закрепленный на основании стержень, на который надеты разноцветные диски. Для того чтобы поместить на стержень новый диск, необходимо надеть его на свободный конец стержня. Чтобы убрать со стержня диск, его необходимо снять со свободного конца стержня. Таким образом, и добавление, и удаление дисков осуществляются всегда только с одного конца стержня; ни снимать, ни надевать диски со стороны основания пирамидки нельзя. Далее, невозможно снять диск, над которым есть другие диски, и невозможно добавить новый диск иначе, как поверх других. Диск, добавленный последним, можно будет снять первым; диск, добавленный первым, будет снят в последнюю очередь.

Еще один классический пример стека – стопка подносов в кафе самообслуживания. Мы можем взять только самый верхний поднос из стопки. Когда подносы заканчиваются, сотрудник кафе приносит новые. Поднос на самом верху стопки – это последний добавленный поднос; поднос на дне стопки – самый первый добавленный поднос.

Нет возможности (если, конечно, не стоит задача все поломать и испачкать) взять поднос из середины стопки. Равно как и нет возможности добавить поднос в середину стопки.

Чтобы, наконец, перейти к делу, приведем еще один пример: магазин для хранения патронов в огнестрельном оружии. Очевидно, что первым будет использован тот патрон, который был помещен в магазин последним. Патрон, добавленный в магазин первым, будет использован последним.

Во всех трех примерах мы видели один и тот же тип поведения: то, что было добавлено первым (диск, поднос, патрон), будет использовано последним. То же самое можно сказать и так: то, что было добавлено последним, будет использовано первым. Как раз для этой формы определения существует эквивалентная краткая «*Last In First Out*», или просто – *LIFO*.

Итак, операции вставки и удаления из определения, приведенного в начале раздела, подчиняются дисциплине LIFO. Такая структура данных и есть стек.

Теперь, когда общая идея стека стала (надеемся) понятной, настало время договориться о том, как стек можно изображать графически. Существуют три основных способа изображения стека, и все они широко используются в литературе по программированию и структурам данных. Первые два способа практически идентичны друг другу. Обсудим сначала их.

Поскольку стек, как и всякая другая структура данных (такая как массив, список, дерево и проч.), хранится в памяти, в основе первых двух способов представления стека лежит отображение стека как набора ячеек памяти. Напомним, что память – это последовательность ячеек. Каждая ячейка имеет номер (от 0 до некоторой величины), или *адрес*. Ячейки памяти с меньшими адресами считаются расположенными ближе к началу, а с большими – ближе к концу памяти. Все ячейки памяти совершенно равноправны, и нет никаких оснований считать те или иные из них более или менее важными.

Графически память изображается в виде последовательности смежных ячеек. Стек может расти как в направлении от меньших адресов к большим, так и наоборот – от больших адресов к меньшим. Если условиться, что ячейки с меньшими адресами (т. е. те, что расположены ближе к началу памяти) изображаются ниже ячеек с большими адресами, то первые два способа изображают стек так, как показано на следующем рисунке:

На левом рисунке стек растет в направлении от меньших адресов к большим (т. е. снизу вверх), на втором – от больших адресов к меньшим (т. е. сверху вниз). По поводу изображенных на рисунках стрелок с надписями «дно» и «вершина» мы вскоре поговорим подробнее.

В большинстве компьютерных архитектур стек, как правило, растет сверху вниз, т. е. от больших адресов к меньшим (правый рисунок, см. выше). В дальнейшем мы будем придерживаться именно такого способа изображения стека, но имейте в виду, что часто можно встретить и другой способ (тот, что слева). Выбор направления не принципиален, и программист вправе использовать тот, который кажется ему более подходящим.

Конечно, интуитивно левый вариант кажется более привлекательным, т. к. мы привыкли считать от меньших значений к большим, но повторимся, что направление счета – это вопрос соглашения.

Третий способ представления стека позволяет изображать стек не в виде рисунков, а в виде строки, которую можно читать привычным способом, т. е. слева направо. Он, конечно, несколько менее нагляден, чем рисунки, но у него имеется одно бесспорное преимущество – компактность. Этот способ представления стека выглядит следующим образом:

```
├ 1 4 109 -58
```

Здесь изображен стек с тем же самым содержимым, что и в первых двух случаях. Стек растет слева направо. Дно стека обозначается символом «├» (впрочем, способ обозначения непринципиален, и мы могли бы выбрать любой другой символ или вовсе ничего не исполь-

зовать). Вершина стека обычно никак не обозначается; считается, что вершина – это просто самый правый элемент в строке.

Преимущество последнего обозначения не только (и даже не столько) в его компактности, сколько в том, что он позволяет легко изображать изменения, происходящие в стеке, т. е. содержимое стека в различные моменты времени: ведь записывать в строку куда проще, чем рисовать. Допустим, мы провели над содержимым стека какие-то операции (об операциях речь пойдет чуть позже). Первые два способа изображения стека, конечно, наглядны и удобны, но трудоемки. Кроме того, их сложно сделать частью документации программного кода. Третий способ решает эти проблемы элементарно:

```
| 1 4 109 -58 → | 1 4 109 -58 49
```

Здесь показаны два состояния стека: до операции (слева от стрелки) и после операции (справа от стрелки). Очевидно, что такая линейная форма записи состояния стека позволяет записывать (и документировать) все основные операции над стеком. Первые два способа для этого чересчур громоздки.

Способ записи состояния стека в виде строки называется *стековой диаграммой* (или *диаграммой стека*) и является общепринятым. Мы будем применять как графическое изображение стека в виде вертикальных ячеек памяти (при этом будет считаться, что стек растет сверху вниз – от больших адресов памяти к меньшим, что соответствует правому рисунку, приведенному выше), так и стековые диаграммы, отдавая последним предпочтение.

Теперь, когда мы обсудили основные способы изображения стеков, можно, наконец, обратиться к понятиям дна и вершины стека, которые, скорее всего, уже и без того вполне понятны.

Неформально говоря, *дно* – это место, начиная с которого накапливаются элементы стека. В терминах адресов памяти дно – это адрес ячейки памяти, с которого начинается участок памяти, выделенной стеку.

Если стек растет снизу вверх, то его дно – ячейка памяти с наименьшим выделенным адресом (не обязательно 0); если стек растет сверху вниз, то его дно – ячейка памяти с наибольшим выделенным адресом (не обязательно последним). Для лучшего понимания рассмотрим пример. Пусть для стека выделен участок памяти с адресами от 1000 до 2000 (включительно). Если стек растет снизу вверх, то дно стека располагается в ячейке с адресом 1000. Новые элементы последовательно «попадают» в ячейки памяти с адресами 1000, 1001,

1002, ..., 2000. Если стек растет сверху вниз, то дно стека располагается в ячейке памяти с адресом 2000, и теперь новые элементы последовательно «попадают» в ячейки памяти с адресами 2000, 1999, 1998, ..., 1000.

Вершина стека – это ячейка памяти, в которую был добавлен или из которой будет удален последний добавленный элемент стека. Рассмотрим пример:

```
├ → ├ 1 → ├ 1 99 → ├ 1 99 -13
```

Пусть изначально стек был пуст (т. е. не содержал никаких элементов). Затем мы добавили в стек число 1. На него указывает *вершина стека*. Теперь последовательно добавим в стек число 99 и число –13 (см. рисунок). Очевидно, что стек содержит уже три элемента (в порядке добавления 1, 99, –13), т. е. стек не пуст. На вершине стека – последнее добавленное число (–13). Продолжим и удалим из стека один элемент. Так как стек – это структура данных, подчиняющаяся дисциплине LIFO, то удаляемым элементом будет последний добавленный (т. е. –13), число 99 теперь будет на вершине стека, и стек примет следующий вид:

```
├ 1 99
```

Вместо термина «вершина» программисты часто используют термин «голова» стека. Вместо термина «дно» встречается термин «основание» стека. Операция добавления нового элемента в стек обычно называется «проталкиванием» в стек (вспомните аналогию с магазином огнестрельного оружия). Операция удаления элемента часто называется «выталкиванием» из стека. Мы будем широко использовать все эти термины, т. к. они довольно точно отображают основные понятия стека и операций над ним.

В программах и описаниях алгоритмов для обозначения дна и вершины стека часто используются обозначения: соответственно bottom (или просто bot) и top (или иногда tail и head соответственно).

Стек предназначен для хранения в нем любой информации – не только чисел, но и символов и даже структур. В нашей книге стек будет использоваться преимущественно для хранения чисел.

Покажем небольшой пример использования стека: переворот строки. Например, если на входе имеется строка HelloWorld, то необходимо на выходе получить строку dlroWolleH. Используя стек, сделать это проще простого:

- поочередно, символ за символом, протолкнуть в стек все символы, составляющие входную строку (т. е. протолкнуть H, затем e, потом l и т. д. вплоть до d);
- поочередно, символ за символом, вытолкнуть из стека все находящиеся в нем символы (сначала d, затем l и т. д. вплоть до W).

Ну разве не просто?

Теперь нам осталось обсудить *основные операции* над стеком. Пока нам понадобятся только три из них (мы будем вводить новые операции постепенно, по мере необходимости). С двумя из них мы уже познакомились. Первая операция – добавление (проталкивание) нового элемента в стек: *push*. Вторая операция – удаление (выталкивание) элемента из стека: *pop*.

Третья операция проверяет стек на пустоту: *empty*.

Кажется, что эти операции не нуждаются в особых комментариях, но давайте посмотрим, так ли это.

Что, если мы попытаемся удалить элемент из пустого стека? Это, бесспорно, ошибка, и именно для предотвращения этой ошибки (она называется *underflow* – опустошение, исчерпание) и вводится операция empty. Пока все логично: если операция empty вернет логическое значение true (т. е. истина), то использовать операцию удаления (выталкивания) элемента pop нельзя; в противном случае операция pop будет выполнена успешно. Теперь представим, что объем памяти, выделенный стеку, исчерпан и стек полностью заполнен. Это означает, что все ячейки стека заняты и адрес вершины стека достиг предельного значения (в примере стека, растущего от адреса 2000 до адреса 1000, предельное значение соответствует адресу 1000). Попытка добавить в стек новый элемент операцией push приведет к ошибке переполнения стека (*overflow*). Если среда исполнения позволяет контролировать переполнение стека (что, как правило, и бывает), то работа программы, пытающейся выполнить такую операцию push, обычно завершается некорректно. Если среда исполнения, напротив, не контролирует переполнения стека, то новый элемент в стек будет добавлен (в нашем примере по адресу 999), но это гораздо хуже, чем ошибка в первом случае. Тогда мы хотя бы знали, что ошибка случилась, программа завершилась досрочно и мы должны придумать другое решение и устранить причину возникновения ошибки (например, слишком большая глубина рекурсивных вызовов). Во втором случае мы фактически портим содержимое памяти, которая не принадлежит стеку. Может случиться, что такая программа даже доработает до конца и нормально завершится. Но можно ли будет ве-

рить результатам работы такой программы? Конечно, нет. Подобная скрытая ошибка очень коварна, поскольку создает иллюзию того, что все идет хорошо. Так что ситуация переполнения должна обязательно контролироваться.

На этом сказанного о стеке и об основных операциях над стеком достаточно, и мы можем вернуться к нашей задаче – анализу скобочных структур, состоящих из различных типов скобок.

Скобочные структуры: общий случай

Теперь, наконец, мы практически готовы рассмотреть общий случай анализа скобочных структур, т. е. таких структур, в которых встречаются любые виды скобок и в любой комбинации. Для этого мы воспользуемся только что введенным понятием стека. Начнем с (правильной) скобочной структуры {()}. Хотя, как мы помним, эта структура правильно разбирается с использованием счетчика, в общем случае счетчик оказывается бессильным. Оставим попытки работать со счетчиком и посмотрим, чем нам может помочь стек.

Общий принцип действия будет таким: когда в последовательности встречается любая открывающая скобка (т. е. скобка (, [, { или <), то мы проталкиваем ее в стек. Когда в последовательности встречается любая закрывающая скобка (т. е. скобка),], } или >), мы действуем согласно следующему правилу, которое, для краткости, назовем «сопоставлением»: если тип закрывающей скобки совпадает с типом скобки на вершине стека (а в стеке, напоминаем, лежат только открывающие скобки), то мы выталкиваем из стека открывающую скобку и переходим к следующей скобке во входной последовательности. Если тип закрывающей скобки не совпадает с типом скобки на вершине стека, то разбор немедленно останавливается, и мы фиксируем ошибку.

Конечно, на словах это выглядит длинно и непонятно, так что давайте обратимся к нашему примеру. Вот так выглядит протокол разбора (последовательность шагов разбора скобочной структуры пронумерована для удобства ссылок):

```
Прочитанная часть   Стек
                    ├                   (1)
{                   ├ {                 (2)
{(                  ├ { (               (3)
{()                 ├ {                 (4)
{()}                ├                   (5)
```

Напомним, что символ ├ обозначает дно стека. Будем предполагать, что в начале разбора скобочной структуры стек пуст; очистку стека можно обеспечить разными способами, но здесь об этом мы говорить не будем. Изначально ни одна скобка не прочитана (шаг 1). Теперь мы считываем первую скобку; это – открывающая фигурная скобка. Поскольку скобка открывающая, она, согласно нашему алгоритму, проталкивается в стек (шаг 2). Читаем следующую скобку. Она, как и первая, – открывающая, хотя и другого типа (а именно – круглая). Ее тоже проталкиваем в стек (шаг 3). Теперь в стеке находятся две открывающие скобки.

Продолжаем считывать скобки из входной строки и встречаем закрывающую круглую скобку. Тут вступает в действие правило сопоставления. Первое, что мы должны сделать, – сопоставить тип только что прочитанной закрывающей скобки с типом скобки на вершине стека. Типы совпадают (обе скобки одного типа, а именно – круглые). Выталкиваем из стека скобку (очевидно, круглую открывающую); пока что считанная последовательность сформирована правильно (шаг 4). Наконец, считываем последнюю скобку. Это закрывающая скобка. Опять применяем правило сопоставления. Тип закрывающей скобки совпадает с типом скобки на вершине стека. Выталкиваем из стека открывающую скобку. А теперь – внимание. Скобочная структура полностью прочитана (иначе говоря – последовательность скобок закончилась), а стек – пуст. Вывод: скобочная структура {()} сформирована правильно. Вот и все. Согласитесь – элегантно. Давайте закрепим навык и рассмотрим более сложную структуру {([])<>}. Последовательно выпишем состояния стека:

```
Прочитанная часть   Стек
                    ├                   (1)
{                   ├ {                 (2)
{(                  ├ { (               (3)
{([                 ├ { ( [             (4)
{([]                ├ { (               (5*)
{([])               ├ {                 (6*)
{([])<              ├ { <               (7)
{([])<>             ├ {                 (8*)
{([])<>}            ├                   (9*)
```

Номера строк, в которых происходит сопоставление текущей прочитанной закрывающей скобки со скобкой на вершине стека, помечены звездочкой (*). Очень важно тщательно изучить этот пример и убедиться в том, что работа стека понята правильно. Мы предлагаем

самостоятельно придумать несколько примеров правильно сформированных скобочных структур и проверить алгоритм их разбора путем тщательного выписывания состояний стека.

Если скобочная структура состоит только из скобок одного типа (то, что ранее мы назвали элементарным случаем), то это, очевидно, частный случай общего типа скобочных структур. Наш алгоритм к этому случаю, разумеется, вполне применим. Иначе говоря, о методе с использованием счетчика можно вовсе забыть и всегда использовать метод с использованием стека.

Итак, мы убедились, что стек позволяет просто и корректно распознавать правильно сформированные скобочные структуры. Теперь надо убедиться в том, что этот метод позволяет распознавать и неправильные скобочные структуры. Рассмотрим уже знакомый нам пример {(}):

```
Прочитанная часть   Стек
                    |-                  (1)
{                   |- {                (2)
{(                  |- { (              (3)
{(}                 |- { (              (4*)
```

Давайте подробно рассмотрим этот протокол разбора. В первых строках происходят чтение открывающих скобок (фигурной и круглой) и их проталкивание в стек. Пока все идет как надо. Но вот мы читаем закрывающую скобку } (шаг 4). В игру вступает правило сопоставления: необходимо сравнить тип закрывающей скобки с типом скобки на вершине стека. И тут обнаруживается, что их типы не совпадают. Мы не можем вытолкнуть из стека открывающую скобку, поскольку для нее нет парной закрывающей того же типа. Дальше можно уже не анализировать – скобочная структура составлена неправильно, и мы фиксируем ошибку. Закрепим результат, для чего рассмотрим скобочную структуру (((}}):

```
Прочитанная часть   Стек
                    |-                  (1)
(                   |- (                (2)
((                  |- ( (              (3)
(((                 |- ( ( (            (4)
(((}                |- ( (              (5*)
(((}}               |- ( (              (6*)
```

Как и раньше, номера строк, в которых происходит сопоставление текущей прочитанной закрывающей скобки со скобкой на вершине

стека, помечены звездочкой (*). Поначалу все идет прекрасно – мы проталкиваем в стек три открывающие скобки, встречаем закрывающую скобку (круглую), сопоставляем ее со скобкой на вершине стека и выталкиваем (шаг 5) из стека открывающую скобку. Но вот мы считываем закрывающую скобку другого типа (а именно фигурную). Попытка сопоставления этой скобки со скобкой на вершине стека терпит неудачу (шаг 6), дальнейший разбор прекращается, и мы фиксируем ошибку.

Еще один простой пример:)(. Тут все элементарно. Мы считываем закрывающую скобку и пытаемся сопоставить ее тип с типом скобки на вершине стека. Но ведь стек пуст и сопоставлять нечего! Результат: скобочная структура)(ошибочна. Напоследок рассмотрим еще одну ситуацию: (){. Вот протокол разбора этой скобочной структуры:

```
Прочитанная часть   Стек
                    ├                   (1)
(                   ├ (                 (2)
()                  ├                   (3*)
(){                 ├ {                 (4)
```

Первые две скобки образуют правильную (допустимую) часть скобочной структуры (шаги 1, 2 и 3). Появление следующей скобки (шаг 4) обрабатывается, как и прежде, – поскольку скобка открывающая, то она проталкивается в стек. И тут скобочная структура заканчивается – скобок в ней больше нет. Разумеется, это ошибка, и скобочная структура (){ считается сформированной неверно. Для закрепления полученных знаний мы рекомендуем составить несколько скобочных структур (как правильно, так и неправильно сформированных) и тщательно проверить их только что рассмотренным способом.

Пора подводить некоторые итоги. Вспомните, сколько усилий мы потратили на то, чтобы проверять скобочные структуры, состоящие из скобок только одного типа. Мы должны были ввести счетчик и корректировать его значение всякий раз, когда считывалась очередная скобка. Но этот метод оказался слишком «слаб», как только скобочная структура стала включать в себя другие типы скобок. В чем причина этой «слабости»? Мы уже упоминали об этом, но тогда это было только предположение. Теперь мы готовы дать ответ.

Что нового дал нам стек? Он дал возможность *хранения* уже прочитанных и обработанных данных. Счетчик, при всей его простоте и привлекательности, не «помнил» предыстории скобок. Счетчик фиксировал только факты появления открывающих и закрываю-

щих скобок, но не их типы. Стек позволяет запоминать предыдущую информацию, сопоставлять эту сохраненную информацию с новой информацией и на основе результатов сопоставления принимать решение о дальнейших действиях. Иными словами, стек работает как *память*. Конечно, как память специфического вида, ведь в стеке доступ осуществляется только в одном месте – в его вершине. Информация, лежащая ниже вершины стека (иными словами, информация, добавленная раньше), не доступна. Но этого оказалось вполне достаточно для решения задачи.

Стек позволяет упорядочить информацию во времени: данные, поступившие раньше других (они хранятся ближе ко дну стека), будут доступны после данных, поступивших позже других (они хранятся ближе к вершине стека). Повторимся: стек хранит данные в порядке их поступления. Именно эта особенность – хранение информации в зависимости от времени ее поступления – делает стек поистине незаменимым, в чем мы не раз убедимся в дальнейшем.

Не будем откладывать и обратимся к следующей задаче, в которой стек предстанет перед нами в новом качестве.

Подпрограммы: постановка задачи

Важным и очень полезным средством разделения программ на отдельные и относительно независимые составляющие являются функции (function), процедуры (procedure), методы (method) и подпрограммы (subroutine). Они представляют собой набор действий, снабженный именем. Подпрограммы можно рассматривать двояко: как средство управления структурой программы и придания программе модульности путем разделения ее на небольшие части и как расширение языка программирования новыми операциями, определенными программистом. Каждая функция, процедура, метод или подпрограмма может быть многократно вызвана из различных частей программы, в том числе из других функций, процедур, методов и подпрограмм. В большинстве современных языков программирования возможны также их рекурсивные вызовы; об этой полезной, интересной, но достаточно непростой возможности мы поговорим позже, после того как детально разберемся с обычными (т. е. нерекурсивными) вызовами.

Принципиальных отличий между функциями, процедурами, методами и подпрограммами нет, и поэтому в дальнейшем мы будем использовать в качестве их общего имени один и тот же, исторически первый термин – *подпрограммы*.