Глава 1

Основные понятия программирования, примеры исполнителей и простейших программ

В этой главе...

Приступая к изучению программирования на языке С, мы начнем с обсуждения понятий, связь которых с самим языком С и программированием для новичка может показаться не вполне очевидной, но которые по сути являются основой основ и которые в той или иной форме определяли принципы программирования во все времена... Это исполнитель, программа, предписание (команда), задача... Конечно, формально можно было бы предположить, что все эти слова и так знакомы со школьного курса информатики и изложить язык С формально, просто как один из языков, на которых можно писать программы. Тогда получился бы учебник, напоминающий те многочисленные учебники иностранных языков, по которым можно изучить алфавит и грамматику, но фактически нельзя научиться разговаривать, для чего совершенно необходимо понимать собеседника и уметь выражать свои мысли на языке, понятном собеседнику. Если бы нужен был именно такой учебник, он был бы втрое тоньше. В принципе таких учебников довольно много. К этой категории, в первую очередь, можно отнести всевозможные краткие справочники, а также учебники, рассчитанные на то, чтобы вызубрить язык перед экзаменом и забыть его навсегда... Не отрицая пользы справочников, я довольно скептически отношусь к пользе зубрежки.

Поэтому данный самоучитель не является сводом правил; он предназначен именно для того, чтобы научиться программировать на языке С, а не просто изучить синтаксис языка С. По этой причине в центре внимания все время находятся понятия программирования и постоянно демонстрируются способы их реализации на языке С. Конечно, в основу изложения могут быть положены разные понятия. Часто по традиции, берущей начало в 30-х гг. прошлого века, в основу программирования кладут

понятие алгоритма. В принципе в этом нет ничего плохого. Однако чтобы при таком подходе по-настоящему научиться программировать, а не просто усвоить основы языка, требуется солидная подготовка по теории алгоритмов. Поэтому в настоящем самоучителе выбраны такие основные понятия программирования, которые позволяют не только освоить программирование на языке C, но и научиться объектно-ориентированным методам. А это значит, что вы без переучивания, зачастую неизбежного при других подходах, сможете профессионально освоить программирование на таких объектно-ориентированных языках, как C++ и C#.

Основные понятия программирования

Говорят, что современной наукой совершенно точно доказано, что существует ген лени. Как бы то ни было, но если не считать трудоголиков, то люди всегда хотели бы, чтобы их работу делал кто-нибудь другой. Иными словами, всегда нужен исполнитель.

Исполнитель — это человек, организация, механическое или электронное устройство, робот и т.п., умеющий выполнять некоторый вполне определенный набор действий. Каждое действие, которое способен выполнить исполнитель, называется *предписанием*, а вся совокупность таких действий — *системой предписаний* исполнителя. Если исполнитель представляет собой процессор или вычислительную машину, то предписания называются также командами, а их совокупность — *набором команд* или *системой команд*. В терминах теории языков программирования предписания часто называются также *операторами*. Приказ о выполнении действия, выраженный формальным, заранее фиксированным способом, называется *вызовом предписания*. В языке программирования такой приказ чаще всего выражается *исполняемым оператором*.

Размышляя об исполнителе как о механическом или электронном устройстве, удобно представлять себе его в виде некоего устройства с кнопками на передней панели — возле каждой кнопки написано имя предписания, которое будет выполнено при нажатии этой кнопки. (Исполнителя можно также представлять и в виде диалогового окна с различными элементами управления.) Система предписаний такого исполнителя совпадает с совокупностью кнопок на его передней панели. Чтобы составить список предписаний, достаточно переписать имена, написанные возле кнопок. Вызов предписания нужно представлять себе как нажатие кнопки, после которого исполнитель выполняет соответствующее действие.

Всякое действие производится над некоторыми объектами и состоит в изменении состояний этих объектов. Объекты могут быть *внутренними* или *внешними*.

Внутренние объекты исполнителя часто называются *глобальными* (для данного исполнителя). Глобальные объекты исполнителя образуют его память, хранящую некоторую информацию об истории работы исполнителя, — их состояния могут меняться в результате выполнения одних предписаний и сказываться на ходе выполнения следующих. Термин “глобальные” подчеркивает, что эти объекты относятся сразу ко всем предписаниям данного исполнителя, т.е. ко всему исполнителю целиком.

Внешние объекты называются *параметрами* предписания. Параметры связываются с исполнителем при вызове конкретного предписания на время выполнения этого предписания и делятся на *входные* и *выходные*. Состояние входного параметра влияет на выполнение предписания, но в результате выполнения не меняется. Состояние вы-

ходного параметра не влияет на выполнение предписания, но может измениться в результате выполнения. Некоторые параметры могут быть и входными, и выходными. Это надо понимать так: их состояния и сказываются на результате, и могут меняться в результате выполнения предписания.

Если представлять исполнителя в виде устройства с кнопками, то глобальные объекты — это внутренние части устройства, состояние которых меняется при нажатии кнопки. Как правило, пользователя они не интересуют, если он не любопытен и ему безразлично, как функционирует устройство. Возле некоторых кнопок на передней панели исполнителя есть отверстия, куда перед нажатием кнопки (перед вызовом предписания) следует поместить внешние объекты. Количество отверстий и их тип определяются конструкцией исполнителя. Сами внешние объекты можно представлять в виде магнитофонных кассет разных размеров, содержимое (состояние) которых может измениться, если объект (а точнее, соответствующее отверстие) является выходным. Эффект нажатия кнопки — эффект вызова предписания — зависит не только от того, какая кнопка нажата, но и от состояний входных объектов (содержимого кассет, вставленных в отверстия предписания), а также от того, что происходило с исполнителем раньше, т.е. от состояния глобальных объектов, памяти исполнителя.

Если вы хотите, чтобы исполнитель выполнял именно ту работу, которая вам нужна, как правило, он должен быть *детерминированным*. Поэтому результат выполнения предписания должен полностью определяться состоянием:

- глобальных объектов исполнителя;
- входных и выходных параметров предписания и состоит в изменении состояний.

Одно и то же предписание (например, команда) может быть выполнимым при одних состояниях глобальных объектов, входных и выходных параметров и невыполнимым при других. (По законам термодинамики, невыполнима, например, команда “запуск двигателя” при отсутствии какого-то бы ни было источника энергии.) В последнем случае возникает предусмотренная конструкцией исполнителя исключительная ситуация — отказ — и продолжение работы с исполнителем становится невозможным (в рамках приведенной выше модели отказ можно понимать как перегорание предохранителей).

Командуя исполнителями, мы обычно достигаем желаемого результата не одним вызовом какого-нибудь предписания, а определенной *последовательностью вызовов* разных предписаний. Иными словами, желаемый результат достигается не в результате выполнения одного вызова какого-нибудь предписания, а в результате выполнения определенной *последовательности* команд. Такая последовательность (а точнее, некоторое ее формальное описание) называется *программой*. А деятельность по составлению таких формальных описаний называется *программированием*. Как правило, программа гораздо короче, чем та последовательность действий, которую она описывает. Ведь программа (обычно неудачно составленная) может описывать даже бесконечную последовательность действий или последовательность, приводящую к отказу одного из исполнителей. Таким образом, программирование — это процесс записи (создания, разработки) программы на языке, понятном тому исполнителю, для которого разрабатывается программа. Поэтому язык, понятный некоему исполнителю, принято называть *языком программирования* (для данного исполнителя), или его *входным языком*. Понятно, что в принци-

пе языков програ ммирования существует столько, сколько существует исполнителей различных типов. Конечно, не все они равнозначны. Поэтому в настоящее время разными авторами насчитывается от 2 до 50 тыс. языков программирования. Трудно перечислить даже только самые распространенные в настоящее время языки программирования. Упомяну лишь такие языки, как Pascal (возможно, вы знакомы с ним со школы), FORTRAN (самый первый язык программирования высокого уровня и потому очень старый — и в то же время вечно молодой!), ALGOL (прадедушка многих современных языков программирования) и, конечно же, совершенно уникальное семейство языков C (сам C, C++, C# и их многочисленные версии, такие как Little C, Small C, RatC и C--). Правила, по которым записывается программа на языке программирования, называются *синтаксисом* данного языка программирования.

Рассмотрим несколько классических (обычно излагаемых в университетских курсах программирования) примеров исполнителей и простейших программ для них.

Исполнитель “Резчик металла” (РМ)

Описание исполнителя

Этот исполнитель представляет собой простейший станок с числовым программным управлением и состоит из прямоугольного стола, на который можно класть листы металла, и резака, расположенного над столом. Резак можно поднимать, опускать и перемещать параллельно краям стола. Рисуя стол и листы металла на бумаге (на доске), мы будем говорить о перемещениях резака вправо, влево, вверх и вниз. При опускании резака и при движении в опущенном состоянии на листе металла под резаком образуется прорезь.

Предположим, что резак имеет размер 0.5*0.5 мм и перемещается с шагом 0.5 мм[1]. Идеализируя ситуацию, можно считать стол Резчика металла клетчатым полем (с клетками размером 0.5*0.5 мм). Резак всегда находится в некоторой клетке этого поля; его можно перемещать в соседние клетки по горизонтали и по вертикали, но не по диагонали. При опускании или передвижении опущенного резака вырезается целиком вся клетка (или в некоторых, “особо интеллектуальных”, моделях нашего станка, определенная ее часть).

Резчик металла снабжен датчиками, которые позволяют определить, находится ли резак в клетке у верхнего, нижнего, левого или правого края стола и есть ли металл в клетке под резаком.

---

[1] *Заметьте, что мы в десятичной дроби отделяем дробную часть точкой, а не запятой. Это соответствует правилам записи в англоязычных странах, и в частности в США и Великобритании. Именно там были созданы почти все получившие широкое промышленное распространение языки программирования. Поэтому не удивительно, что почти во всех языках программирования дробная часть отделяется точкой, а не запятой. Обратите внимание также на то, что вместо крестика (знака умножения) мы используем звездочку. Это связано с тем, что в большинстве устройств ввода крестик отсутствовал, и в языках программирования он был заменен * (звездочкой). Оказалось, что каких-либо неудобств это не вызывает, и необходимость в крестике как особом знаке для умножения отпала. Нет такого знака и на современных клавиатурах. Впрочем, на клавиатурах некоторых машин (например, в машинах серии МИР) и их входных языках такой знак был, зато отсутствовала звездочка!*

Система предписаний Резчика металла состоит из следующих предписаний:

```
начать работу
поднять резак
опустить резак
шаг вправо
шаг влево
шаг вверх
шаг вниз
резак над металлом           : да/нет
резак у верхнего края стола  : да/нет
резак у нижнего края стола   : да/нет
резак у правого края стола   : да/нет
резак у левого края стола    : да/нет
закончить работу
```

Только что мы описали (частично, правда) синтаксис входного языка Резчика металла — ведь мы указали, как следует записывать его предписания. (Чтобы описание было полным, нужно еще указать, как из предписаний строить программы.) Сам язык, на котором описывается синтаксис данного языка, называется *метаязыком*.

Однако синтаксиса еще не достаточно для того, чтобы указать, что же фактически выполняется по тем или иным инструкциям или операторам. Описание смысла программ на языке программирования называется его *семантикой*. Приведем пример семантического описания инструкции.

> Запись “резак над металлом : да/нет” означает, что у предписания “резак над металлом” есть один выходной объект, и этот объект может принимать одно из двух состояний: либо да, либо нет. (Можно считать, что в кнопку “резак над металлом” встроена лампочка, которая после нажатия кнопки может либо загореться, либо не загореться.)

Предписания такого вида называются *вырабатывающими значение типа да/нет*.

Тип в языке программирования — это множество значений и допустимых операций над ними.

Покажем, что с помощью более сложного метаязыка систему предписаний Резчика металла можно написать более компактно. Действительно, условимся, что косая черта (/) является символом нашего нового метаязыка, который означает, что можно выбрать одно (любое) из значений (на самом деле слов), разделенных этим знаком. Тогда систему предписаний Резчика металла можно написать более компактно так:

```
начать работу;
поднять/опустить резак
шаг вправо/влево/вверх/вниз
резак над металлом                                  : да/нет
резак у верхнего/нижнего/правого/левого края стола : да/нет
закончить работу
```

Мы видим, что удачный выбор метаязыка позволяет записывать системы предписаний в более сжатой форме. Ведь теперь в строке можно описать целую группу предписаний.

Продолжим пояснение семантики операторов нашего языка. Иными словами, теперь мы собираемся пояснить, что происходит при выполнении отдельных предписаний Резчика металла. По предписанию “начать работу” резак оказывается в левом верхнем углу стола в поднятом положении. При попытке сделать шаг вверх, когда резак уже находится у верхнего края стола, возникает исключительная ситуация “отказ”.

Аналогично отказ происходит при попытке сделать шаг вправо, когда резак находится у правого края, и т.п. Предписания “поднять/опустить резак” переводят резак в требуемое положение независимо от того, был ли он поднят или опущен, и никогда не приводят к отказу. Резак можно двигать и в поднятом, и в опущенном состоянии независимо от того, есть под ним металл или нет.

Пример программы

Наконец, приведем схематический пример программы для исполнителя “Резчик металла”:

```
программа прямоугольник
•  дано    : | лист металла закрывает весь стол Резчика
•  получить: | от левого верхнего угла отрезан прямоугольник
•           | размером 2*3 шага Резчика металла
•--------------
•  РМ.начать работу
•  РМ.шаг вниз
•  РМ.шаг вниз
•  РМ.опустить резак
•  РМ.шаг вправо
•  РМ.шаг вправо
•  РМ.шаг вправо
•  РМ.шаг вверх
•  РМ.шаг вверх
•  РМ.поднять резак
•  РМ.закончить работу
конец программы
```

Эта программа описывает последовательность из 11 вызовов предписаний исполнителя “Резчик металла”. По окончании ее выполнения в листе металла будут прорезаны клетки, помеченные черными квадратиками, как показано на рис. 1.1.

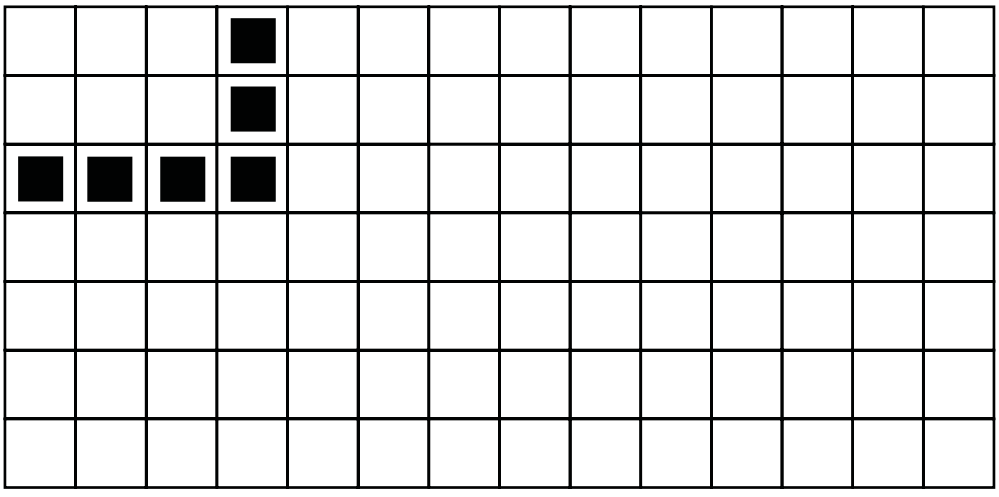

Рис. 1.1. Резчик металла отрезал уголок заданного размера от листа металла

Напомним: программа — это формальное описание последовательности вызовов предписаний (в нашем примере — предписаний Резчика металла). Слово “формальное” означает, что описание должно удовлетворять строгим, вполне определенным правилам (синтаксису входного языка). Мы не будем перечислять их, однако заметим, что в соответствии с этими правилами необходимо в нужном порядке записать слова **программа**, **дано**, **получить**, **конец программы**, провести составленную из точек вертикальную линию, соединяющую слово **программа** со словами **конец программы**, сделать отступ (т.е. сдвинуть имена предписаний вправо от вертикальной черты) и т.п.

Если в строке программы встречается знак |, то текст, начиная от этого знака и до конца строки, считается комментарием. *Комментарий* — это произвольный текст, который никак не влияет на последовательность вызовов предписаний при выполнении программы, а служит лишь для пояснения смысла программы при чтении ее человеком. Например, в программе “прямоугольник” комментарием является текст после слов **дано** и **получить**.

Исполнитель “Робот-разведчик” (РР)

Описание исполнителя

Этого исполнителя можно представлять в виде лунохода, которого Агент 007 забрасывает куда-то в далекую неизвестную местность (назовем ее квадрантом). Роботом-разведчиком Агент 007 может управлять (например, по радио) с помощью следующей системы предписаний:

```
начать работу
впереди/справа/слева свободно       : да/нет
впереди/справа/слева занято          : да/нет
сделать шаг
шагать до упора
повернуться направо/налево
повернуться на север/юг/запад/восток
закончить работу
```

Квадрант Робота-разведчика является прямоугольным клетчатым полем неизвестных размеров. Некоторые клетки квадранта могут быть заняты препятствиями. По краю квадрант обнесен сплошной стеной (ведь это секретный объект!).

По предписанию “начать работу” Робот-разведчик оказывается в северо-западной (левой верхней) клетке квадранта, лицом на восток (рис. 1.2).

На рисунках (а вы, конечно, понимаете, что на самом деле это сверхсекретные карты!) местоположение Робота-разведчика изображается стрелкой, указывающей его ориентацию. Например, стрелка ▸ указывает ориентацию на восток, а стрелка ▾ — на юг.

По предписанию “слева свободно” Робот-разведчик анализирует, что находится слева от него. Если слева стена или клетка с препятствием, то он отвечает **нет**. Если же слева пустая клетка квадранта, то он отвечает **да**. Предписание “слева занято” выполняется аналогично, В состоянии, изображенном на рис. 1.2, в ответ на “слева занято” Робот-разведчик ответит **да**, а на “слева свободно” — **нет**.

Если клетка впереди Робота-разведчика свободна, то он может переместиться в нее по предписанию “сделать шаг”. Если же впереди стена или препятствие, то попытка выполнить предписание “сделать шаг” приведет к отказу. По предписанию “шагать до упора” Робот-разведчик перемещается вперед до тех пор, пока не упрется в стену

или в препятствие. Из состояния, изображенного на рис. 1.2, по предписанию “шагать до упора” Робот-разведчик переместится в северо-восточную (правую верхнюю) клетку квадранта и остановится в ней лицом по-прежнему на восток (рис. 1.3).

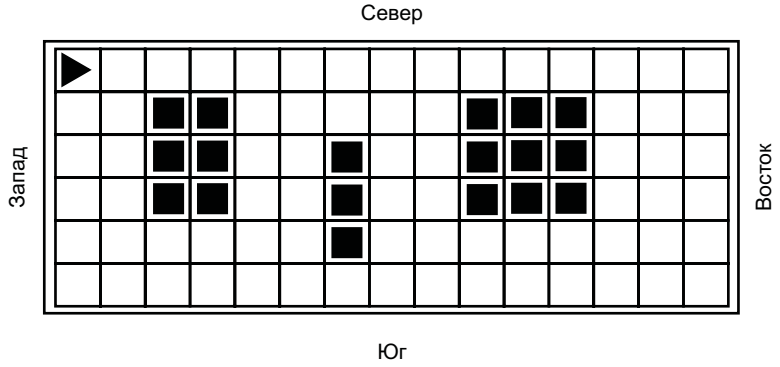

Рис. 1.2. Когда Робот-разведчик получит предписание “начать работу”, он переместится в северо-западную (на карте левую верхнюю) клетку квадранта и повернется лицом на восток

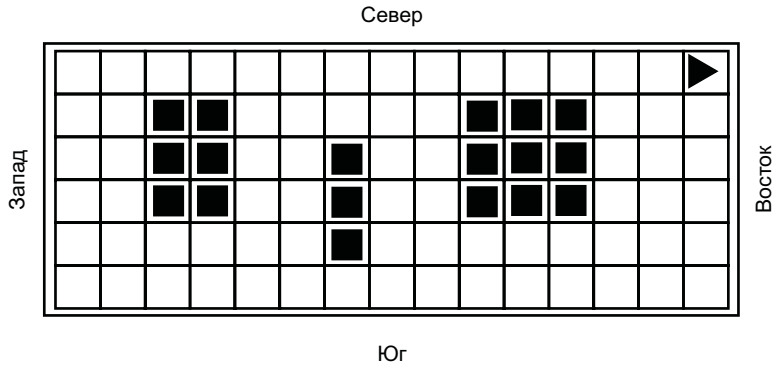

Рис. 1.3. Как вы помните, Робот-разведчик, получив предписание “начать работу”, переместился в северо-западную (на карте левую верхнюю) клетку квадранта и повернулся лицом на восток. А по новому предписанию “шагать до упора” Робот-разведчик переместился в северо-восточную (на карте правую верхнюю) клетку квадранта и остановился в ней лицом на восток

Если в этот момент Робот-разведчик получит предписание “шагать до упора” еще раз, то он никуда не переместится, однако и отказа не возникнет. Таким образом, вызов предписания “шагать до упора” никогда не приводит к отказу.

Наконец, по предписанию “повернуться ...” Робот-разведчик поворачивается. При этом ему говорится, либо куда он должен повернуться (налево — на 90° против часовой стрелки или направо — на 90° по часовой стрелке), либо в каком положении он должен оказаться (лицом на север, на запад и т.п.).

Программа “прогулка”

Сейчас давайте рассмотрим разработку несложной программы для нашего Робота-разведчика. Дело в том, что сам по себе Робот-разведчик, конечно, совершенно бесполезен. Ведь самостоятельно он ничего не делает. Прежде всего должна быть сформулирована задача, которую должен выполнить Робот-разведчик. Именно ради достижения цели, сформулированной в постановке задачи, и должна разрабатываться программа.

На практике именно тогда, когда *постановка задачи* давно забыта (если она вообще когда-либо существовала!), *заказчик* вспоминает о том, что неплохо было бы найти фактического *исполнителя*, который и напишет программу. (Вообще-то, тот, кто выполняет работу по созданию программ, называется *программистом*, но заказчики не всегда знают это слово.) Так что, как правило, перед тем как приступить к написанию программы, приходится сначала заняться постановкой задачи, причем в условиях острого дефицита времени, полной некомпетентности заказчика, надоедливых советов и многочисленных вопросов типа “Извините, я не знаю, *что* вы там будете делать, но я ни за что не отстану от вас, пока не узнаю, *как* вы это будете делать”. Поэтому ничуть не удивительно, что современное программирование часто называется *экстремальным*, и в настоящее время усиленно разрабатываются его технологии и принципы менеджмента. Последние часто заключаются в том, что одну задачу решает несколько человек: за компьютером находится не менее двух программистов, а в затылок им дышит “неназойливый” босс. (Это не шутка. Есть даже книги, в которых этот китайский метод разработки программ преподносится как последнее достижение в области технологии программирования. Всерьез обсуждается вопрос о количестве мышей и расстановки мебели на таком рабочем месте и приводятся соответствующие фотографии.)

Итак, когда заказчик давно и прочно забыл постановку задачи, программист в экстремальных условиях должен сформулировать *технические требования* (иногда их еще называют *спецификациями*) к программе. Предположим, Робот-разведчик был создан для разведки местности. Тогда необходима программа, которая позволяет разведать местность. Но что значит “разведать”? Провести топографическую съемку, геологическую разведку или собрать секретные сведения об объектах в квадранте? Едва ли заказчик (возможно, им будет сам Агент 007) скажет об этом программисту. Скорее всего, он скажет, что **Робот-разведчик должен совершить прогулку по квадранту**. Это и есть та постановка задачи, которую программист может использовать в качестве рабочей. Однако ее нужно уточнить, или, как часто говорят, *детализировать*. Действительно, что такое *прогулка*? Это слово нигде не встречается в системе предписаний Робота-разведчика. Поэтому это слово нужно определить через другие, связь которых с системой предписаний Робота-разведчика более очевидна. Предположим, что вы решили, что *прогулка* — это *обход квадранта*. Тем самым вы уточнили постановку задачи, причем такое уточнение необходимо зафиксировать (лучше всего письменно!) в договоре с заказчиком. Но что значит *обойти квадрант*? Облазить все его клетки? Это невозможно, если там есть препятствия. Посетить все свободные клетки? Возможно, но только в том случае, если из левой верхней клетки можно попасть во все свободные. Но возможно, что нужно всего лишь *обойти квадрант по периметру*. Если заказчик сказал, что это именно то, что нужно, опять не поленитесь и (письменно!) зафиксируйте это в договоре, ведь это новое уточнение технического задания. Но что такое **периметр квадранта**? Возможно, это **периметр прямоугольника**, ограничивающего квадрант. Если заказчик согласен с таким уточнением, не поленитесь в очередной раз (письменно!) зафиксировать такое уточнение в договоре. Теперь, как легко сообра-

зить, вся задача свелась к тому, что Робот-разведчик должен *пройти под стенами квадранта*. (Странно, если заказчик с этим не согласится, — ведь он подписал все необходимые бумаги!) Возможно, вы уже представляете, какой вид должна иметь программа. Но все же (особенно в более сложных ситуациях) необходимо предварительно выяснить возможность выполнения этой задачи. Иными словами, проверяйте *непротиворечивость технического задания* — оно должно быть понятным до начала разработки программы и труд по его составлению также должен оплачиваться... Во всяком случае нужно хотя бы убедиться в том, что руководство представляет, какое программное изделие ему нужно, а не скажет потом, что, конечно, *оно не помнит, что именно было заказано, но вы сделали совсем не то*... В нашем случае понятно, что **препятствия должны не прилегать к стенам квадранта**. Честно говоря, заказчик с этим условием должен согласиться автоматически, так как оно вытекает из *его* (подписанных *им*) требований. (Даже если он не согласен, он должен оплатить все выполненные работы и, возможно, уплатить неустойку, если таковая была предусмотрена в договоре.) Если заказчик согласен, можете приступать к разработке программы обхода квадранта. Программе уже можно дать и рабочее название: *прогулка*. Одновременно можно приступать и к разработке документации к программе. В документации уже можно написать, например, как называется программа, что она делает (обход квадранта по периметру) и что условием ее выполнения является следующее: **препятствия не прилегают к стенам квадранта**. Чтобы теперь написать нашу программу, достаточно понаблюдать и аккуратно записать все, что делает бабушка с внуком, который обходит песочницу по периметру.

```
программа прогулка
• дано    : | препятствия не прилегают к стенам квадранта.
• получить: | Робот-разведчик обошел квадрант по периметру
•-------------
•   Робот-разведчик.начать работу
•   Робот-разведчик.шагать до упора
•   Робот-разведчик.повернуться направо
•   Робот-разведчик.шагать до упора
•   Робот-разведчик.повернуться направо
•   Робот-разведчик.шагать до упора
•   Робот-разведчик.повернуться направо
•   Робот-разведчик.шагать до упора
•   Робот-разведчик.повернуться направо
•   Робот-разведчик.закончить работу
конец программы
```

Это уже хотя и совсем не сложная, но реальная программа, способная управлять Роботом-разведчиком. Выполнив ее всестороннее тестирование, можете вместе с необходимой документацией передать ее заказчику. (Только не забудьте предварительно подписать акт о проведенных испытаниях и выполненной работе, — без этого получить деньги за выполненные работы будет существенно сложнее, чем написать программу!)

Теперь давайте подумаем, почему программа получилась такой простой. В значительной степени это определяется наличием спецификации на каждом этапе разработки. Почему же тогда программисты не пишут спецификации? Причин несколько. Во-первых, слишком мало внимания в курсах программирования уделяется тому, как нужно писать спецификации. (Этому нужно учиться.) Во-вторых, руководители часто не знают, какими должны быть спецификации и потому вместо написания спецификаций

программисты часто заняты бесполезным бумаготворчеством, результатом которого является горный массив отписок и полное отсутствие каких бы то ни было спецификаций. В-третьих, по привычке часто программист полагает, что “данная программа абсолютно тривиальна и все сойдет и так...” Между тем, законы Мэрфи и падающего бутерброда ни в одной области человеческой деятельности не собирают столь богатый урожай, как в программировании. В-четвертых, даже если вы решили написать спецификации и они кажутся вам простыми, то, сев за стол (или за клавиатуру), вы обнаруживаете, что не можете кратко изложить суть того, что секунду назад казалось таким тривиальным. Но подумайте сами: какой прок от кода, если вы не можете толком объяснить, что он делает?! В-пятых, код может появиться раньше, чем спецификации. (Этим синдромом страдают многие суперпрограммисты, но не все страдающие этим синдромом — суперпрограммисты...) В-шестых, в условиях цейтнота возникает соблазн написать все потом... (Но потом все забывается, даже самые тривиальные вещи.) И действительно, если вы не напишете спецификации, все время (и даже больше отпущенного!) уйдет на отладку, причем отладка “съест” и время следующего проекта, и вы окажетесь в цейтноте еще до того, как приступите к нему. В следующем проекте вы опять пожертвуете спецификациями, стремясь побыстрее (уже давно пора!) представить хоть какой-нибудь код, чтобы убедить очередного заказчика, что все идет по плану, а отставание столь мизерное, что существенного значения не имеет. Цикл повторится. Поэтому не удивительно, что на отладку многие программисты тратят 90% времени!

Есть лишь один способ избежать цейтнота — писать подробные формализованные технические задания, обязательно проводить доказательство правильности программ и тщательного соблюдать дисциплину программирования. Технические задания даже на простые программы также приносят пользу — позже, когда вам понадобится восстановить в памяти, что же делает та или иная относительно тривиальная программка (одна из сотен или тысяч в разрабатываемой системе). У программистов, пишущих технические задания, обычно повышается качество программ, а время программирования вопреки всем ожиданиям существенно сокращается (в основном за счет отладки), причем существенно упрощается сопровождение разработанных систем и их модификация (подготовка новых версий). Так что хотя комментарии и не влияют на работу исполнителя, позвольте вам еще раз посоветовать тщательно документировать и комментировать программы, а строки **дано** (предусловие) и **получить** (постусловие), практически являющиеся комментариями, записывать до разработки программы. Разработка программы должна вестись так, чтобы с помощью комментариев и документации можно было без труда *проверить* (программисты часто говорят *верифицировать*) программу и *доказать ее правильность*. О том, как это делается, мы еще поговорим, и не раз...

Заметьте, что нам удалось избежать ошибок еще и потому, что мы применили специальный метод: запись того, что делает исполнитель (бабушка с внуком). Такой метод широко применяется при создании простых программ и называется записью *макросов*, или *скриптов*. Для этого во многих приложениях (в качестве примеров могут служить пакеты Microsoft Office, CorelDraw, Adobe PhotoShop, Microsoft Visual C++ и др.) предусмотрены специальные панели инструментов. Средства записи макросов — это лишь небольшая (хотя и наиболее часто применяемая) часть *инструментальных средств*, умелое применение которых может существенно облегчить написание программы и зачастую позволяет избежать многих досадных описок.

Теперь, в свете всего сказанного, перечитайте программу еще раз и ответьте на следующие вопросы.

1. Достаточно ли четко сформулировано постусловие?
2. Нужна ли предпоследняя команда? Если считаете что да, то зачем?
3. Вытекает ли ее необходимость из постановки задачи (из постусловия)?
4. Будет ли программа правильной, если предпоследнюю команду повторить еще четыре раза? А если пять?
5. Четко ли сформулировано предусловие?

Вот какие ответы предложил бы я.

Постусловие сформулировано нечетко. Ведь четко не сформулировано, что значит "Робот-разведчик обошел квадрант по периметру". В частности, не ясно, как он должен быть ориентирован. В случае с внуком предпоследняя команда может быть и не важна, просто бабушка хочет ему показать пройденный им путь. В случае же лунохода пропуск предпоследней команды может быть критическим: если луноход начнет передавать изображения, то он будет снимать совсем не то, что надо, а при попытке продвинуться вперед расшибется о стену или свалится в пропасть. Таким образом, запись постусловия "Робот-разведчик обошел квадрант по периметру" разные люди могут трактовать по-разному, и в зависимости от этого одни будут считать, что предпоследняя команда необходима, а другие — что она не нужна. Если же ее повторить еще четыре раза, то хотя формально результат выполнения программы (положение и ориентация Робота-разведчика) не изменится, программа станет длиннее. Кроме того, Робот-разведчик совершит лишний оборот. Это может быть как желательным (осмотрел местность), так и нежелательным (закружилась голова, дополнительный расход энергии). И так как мнения у разных людей могут разойтись, то необходимо признать, что постусловие сформулировано нечетко.

Наконец, давайте попытаемся ответить на вопрос: *была ли постановка задачи совершенно четкой и можно ли было приступать к ее решению?*

Несмотря на то, что постановка задачи была весьма далека от идеальной, она была понятной (хотя и несколько нечеткой). Что касается нечеткости как таковой, то наличие ее на начальном этапе в каком-то смысле неизбежно. Однако важно, чтобы программист смог провести анализ (возможно, не без помощи заказчика!) и уточнить детали, знание которых необходимо для написания программы. Ведь мы смогли мысленно смоделировать задачу и решить ее. Во всяком случае, мы нашли первое (если честно, нулевое) приближение. Конечно, прежде чем использовать найденное решение для управления луноходом, АЭС, подводной лодкой или детским манежем, необходимо уточнить техническое задание, проверить (верифицировать) правильность программы и всесторонне ее *протестировать* (сюда входят многочисленные испытания — как в лабораторных условиях, так и у заказчика).

Программа "назад до упора"

Следующая программа короче и проще. Вот она:

```
программа назад до упора
• дано     : | Робот-разведчик где-то в квадранте
• получить: | Робот-разведчик сместился назад до упора,
•           | ориентация Робота-разведчика не изменилась
•-------------
•   Робот-разведчик.повернуться налево
•   Робот-разведчик.повернуться налево
```

- `Робот-разведчик.шагать до упора`
- `Робот-разведчик.повернуться налево`
- `Робот-разведчик.повернуться налево`

конец программы

Проанализируйте эту программу самостоятельно и ответьте на следующие вопросы.

1. Обязательно ли после выполнения данной программы Робот-разведчик вернется в ту же клетку, в которой он находился перед ее выполнением?
2. Предположим, что Робот-разведчик может передвигаться в квадранте только с помощью его предписаний. (Иными словами, его нельзя забросить как десантника в произвольную клетку квадранта с самолета.) Как в квадранте должны быть расположены препятствия, чтобы после выполнения данной программы Робот-разведчик обязательно возвращался в ту же клетку, в которой он находился перед ее выполнением?
3. Предположим, что с помощью сверхъестественных средств (черная и белая магия, самолет для заброски диверсантов, а также команда "Робот-разведчик.переместиться в свободную клетку с координатами (x, y)") Робот-разведчик может попасть в любую свободную клетку квадранта. Как в квадранте должны быть расположены препятствия, чтобы после выполнения данной программы Робот-разведчик обязательно возвращался в ту же клетку, в которой он находился перед ее выполнением?

Вот решение этих задач.

Прежде всего отметим следующее свойство данной программы: после ее выполнения за спиной у Робота-разведчика оказывается препятствие. (Действительно, после выполнения команды "Робот-разведчик.шагать до упора" Робот-разведчик окажется лицом к стене, а после выполнения двух следующих команд повернется на полоборота.) Поэтому если Робот-разведчик займет такую клетку, что за спиной у него не будет препятствия, то после выполнения данной программы он окажется в другой клетке. Таким образом, ответ на первый вопрос отрицательный.

Если предположить, что Робот-разведчик может передвигаться в квадранте только с помощью его предписаний, то начать движение Робот-разведчик сможет только из северо-западной клетки. Если он сможет сделать хотя бы один шаг, то, сделав его, он окажется в такой клетке, что сзади него не будет препятствия, следовательно, после выполнения данной программы он окажется уже в другой клетке. Поэтому чтобы после выполнения данной программы Робот-разведчик обязательно возвращался в ту же клетку, в которой он находился перед ее выполнением, северо-западная клетка квадранта должна быть огорожена препятствиями, либо она должна быть занята, тогда робот вообще не сможет попасть в квадрант из-за отказа в предписании "Робот-разведчик.начать работу". Это и есть ответ на второй вопрос.

Наконец, если предположить, что Агент 007 может использовать черную и белую магии, самолет для заброски диверсантов, а также не менее магическую команду "Робот-разведчик.переместиться в свободную клетку с координатами (x, y)", то Робот-разведчик может попасть в любую свободную клетку квадранта. Если из нее можно сделать хотя бы один шаг, то, сделав его, робот окажется в такой клетке, что сзади него не будет препятствия, значит, после выполнения данной программы он окажется уже в другой клетке. Поэтому чтобы после выполнения данной программы Робот-разведчик обязательно возвращался в ту же клетку, в которой он находился перед ее

выполнением, все свободные клетки квадранта должны быть огорожены препятствиями. Это и есть ответ на третий вопрос.

Мораль: программу нужно читать не затем, чтобы проверить синтаксис (это сделает компилятор!), а чтобы найти ее свойства. Именно они могут помочь ответить на вопросы о ее поведении и правильности обработки данных.

Читая программу, мы увидели, что данные могут иметь довольно сложную организацию (квадрант разделен на клетки, некоторые из них свободны, а в некоторых могут находиться препятствия). Чтобы получить представление о принципах организации данных, мы сейчас познакомимся с одной из наиболее любимых программистами структур данных — со стеком.

Стек

Стек — одна из простейших структур данных и, несомненно, одна из наиболее часто используемых в программировании. Прежде чем перейти к примерам, введем понятие стека. *Стеком* называется любая структура, в которой могут накапливаться какие-то элементы и для которой выполнено следующее основное условие: *элементы из стека можно доставать только в порядке, обратном порядку добавления их в стек*.

Конечно, столь важная структура данных, как стек, многократно "переизобреталась" и потому имеет и другие названия: *магазин*, *стековая память*, *магазинная память*, *память магазинного типа*, *запоминающее устройство магазинного типа*, *стековое запоминающее устройство*. Переформулируя данное выше определение, можно сказать, что это структура данных, в которой новый элемент всегда записывается в ее начало (вершину), а очередной читаемый элемент также всегда выбирается из ее начала. Таким образом, стек представляет собой список, добавление элементов к которому и удаление их из него осуществляется по принципу "последним пришел — первым вышел" (LIFO, Last In — First Out). Эта упорядоченная структура данных используется программистами для записи и извлечения данных в соответствии с принципом "последним пришел — первым вышел". Такой структурой данных очень часто приходится пользоваться при программировании на языке Ассемблер. Дело в том, что для программ выделяется специальная область памяти, функционирующая по принципу "последним пришел — первым вышел". Эта область, называемая также стеком, используется для временного хранения адресов возврата при вызовах функций и процедур, а также для хранения передаваемых параметров и локальных переменных. Информация размещается в стеке с помощью команды *push* (*затолкнуть в стек*). Извлечение информации из стека происходит с помощью команды *pop* или *pull* (*вытолкнуть* или *извлечь из стека*). При этом из стека "выталкивается" элемент, который занимает самую верхнюю позицию, — вот почему стек известен как структура, действующая по принципу "последним пришел — первым вышел". Примеры стеков могут быть следующими.

- Обыкновенная детская пирамидка (стержень, на который надевают колечки, рис. 1.4): большое колечко, которое было надето раньше, нельзя снять, не сняв предварительно маленькое, которое было надето позже. Детские пирамидки используются в игре "Ханойские башни", которая фактически состоит из трех пирамидок (рис. 1.5).

- Железнодорожный тупик, в который загоняют вагоны: вагон, загнанный ранее, нельзя выгнать из тупика, не выгнав предварительно вагон, который был загнан позже.
- Труба с одним запаянным концом, куда помещают разноцветные цилиндрические "бочонки" (диаметр которых чуть меньше диаметра трубы). Последний пример в наибольшей степени соответствует программистскому понятию стека: заглядывая в трубу, мы сможем увидеть, что бочонков в трубе нет, или увидеть цвет верхнего бочонка, но не сможем увидеть, есть ли бочонки под верхним, сколько их и каких они цветов.

Элемент стека, который в данный момент можно взять, т.е. самый "верхний" (верхнее колечко на пирамидке, последний загнанный вагон в тупике и т.п.), называется *вершиной стека*. Если число элементов в стеке не может превышать некоторой величины, то стек называют *ограниченным*, а максимальное число элементов в нем — *глубиной стека*. Стек, в котором нет ни одного элемента, называется *пустым*.

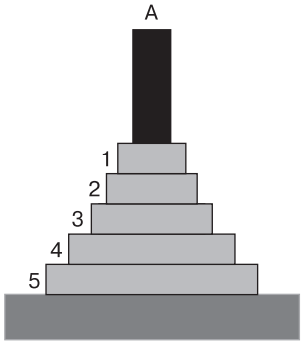

Рис. 1.4. Детская пирамидка — первый прибор для серьезного ознакомления со стеком

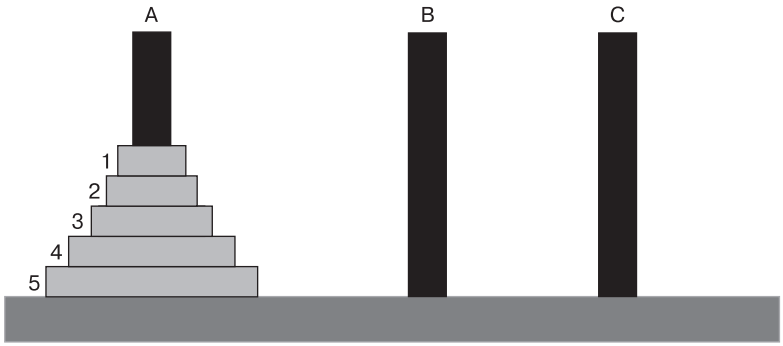

Рис. 1.5. Игру "Ханойские башни", фактически состоящую из трех детских пирамидок, любят и дети, и взрослые, а программисты просто обожают разрабатывать алгоритмы этой игры и писать программы, демонстрирующие эту игру

Исполнитель “Стековый калькулятор” (СК)

Программисты обнаружили, что в стек удобно складывать не только материальные объекты, такие как диски, тарелки, железнодорожные вагоны, но и абстрактные объекты, например числа. Оказалось, что, благодаря стеку, в арифметических выражениях, например, можно обойтись без скобок и забыть о приоритете выполнения операций, т.е. с использованием стека операнды можно расположить так, чтобы арифметические действия (числа) можно было выполнять в том порядке, в каком они записаны. Именно это умеет делать исполнитель “Стековый калькулятор” (СК). Сейчас мы рассмотрим его систему предписаний.

Система предписаний исполнителя “Стековый калькулятор”

Система команд исполнителя “Стековый калькулятор” состоит из следующих команд:

```
начать работу
добавить <вх: число> в стек
сложить/вычесть/умножить/разделить
показать результат
закончить работу
```

Запись “добавить <**вх**: число> в стек” означает, что это предписание имеет один входной объект (входной параметр), который является числом.

Исполнитель “Стековый калькулятор” имеет два глобальных объекта: стек чисел (т.е. стек, элементами которого являются числа) и табло, на котором по предписанию “показать результат” изображается вершина стека (сам стек при этом не меняется).

После вызова предписания “начать работу” стек калькулятора пуст, а на табло ничего не изображено. Предписание “добавить <**вх**: число> в стек” добавляет указанное при вызове число в стек. Предписание “показать результат” изображает на табло вершину стека. После этого табло не меняется до следующего вызова предписания “показать результат”. При выполнении арифметических операций (сложить, вычесть, умножить, разделить) Стековый калькулятор достает из стека последовательно сначала правый, а затем левый аргументы операции, выполняет операцию и полученный результат помещает в стек (рис. 1.6).

Рис. 1.6. Чтобы выполнить арифметическое действие, Стековый калькулятор достает из стека последовательно сначала правый, а затем левый аргументы операции, выполняет операцию и полученный результат помещает в стек

Порядок выборки аргументов из стека легко запомнить: если мы хотим вычислить с помощью калькулятора разность 22 − 5, то аргументы операции "−" надо поместить в стек в том порядке, в котором мы их читаем, — сначала 22, а потом 5.

При выполнении любой арифметической операции число элементов в стеке уменьшается на 1. Попытки выполнить операцию, когда в стеке меньше двух элементов, или показать результат, когда стек пуст, приводят к отказу.

Пример программы для вычисления выражения с помощью Стекового калькулятора

Предположим, нам нужно вычислить 17*(7+19)+52. Вот возможная версия программы для вычисления этого выражения с помощью Стекового калькулятора:

```
программа вычисление
• дано     :
• получить: | значение формулы 17*(7+19)+52 на табло СК
•-------------
•   СК.начать работу
•   СК.добавить <17> в стек
•   СК.добавить <7> в стек
•   СК.добавить <19> в стек
•   СК.сложить
•   СК.умножить
•   СК.добавить <52> в стек
•   СК.сложить
•   СК.показать результат
•   СК.закончить работу
конец программы
```

Последовательные состояния стека при выполнении этой программы изображены на рис. 1.7.

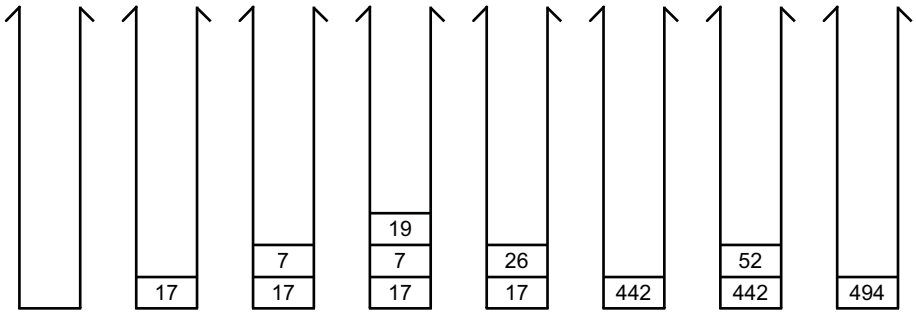

Рис. 1.7. Состояния стека при вычислении значения формулы 17*(7+19)+52. После вычисления на табло Стекового калькулятора будет отображаться число 494 (результат вычисления)

Исполнитель "Счетчик"

Исполнитель "Счетчик" кажется тривиальным. Тем не менее он находит широкое применение. Он используется, например, туристами при подсчете количества шагов, на заводах при подсчете количества вышедших с конвейера единиц продукции и т.д. В Windows он представлен специальным элементом управления.

Система предписаний исполнителя “Счетчик”

Простейшая система предписаний исполнителя “Счетчик” состоит из следующих команд:

```
начать работу
установить в нуль
увеличить на единицу
показать значение
закончить работу
```

Исполнитель имеет два глобальных объекта: внутренний, состоянием которого может быть неотрицательное целое число, и табло, на котором по предписанию “показать значение” это число отображается. Семантика предписаний очевидна из их названий.

Программа для исполнителя “Счетчик”

В качестве примера программы для исполнителя “Счетчик” рассмотрим программу, после выполнения которой будет отображаться число 5.

```
программа пять
• дано    :
• получить: | на табло Счетчика число 5
•-------------
•   Счетчик.начать работу
•   Счетчик.установить в нуль
•   Счетчик.увеличить на единицу
•   Счетчик.увеличить на единицу
•   Счетчик.увеличить на единицу
•   Счетчик.увеличить на единицу
•   Счетчик.увеличить на единицу
•   Счетчик.показать значение
•   Счетчик.закончить работу
конец программы
```

Задачи и упражнения

1. Напишите программу для исполнителя “Резчик металла”, вырезающую в листе металла год вашего рождения.
2. Напишите программу для исполнителя “Стековый калькулятор”, которая вычисляет значение формулы 15 − (16 + 18/(17 − 15))+31.
3. Напишите программу для Стекового калькулятора, которая показывает на табло год вашего рождения. А теперь решите эту же задачу при условии, что в программе в предписаниях “добавить <**вх**: число> в стек” можно использовать только однозначные числа от 0 до 9. Наконец, решите эту же задачу при условии, что в программе в предписаниях “добавить <**вх**: число> в стек” можно использовать только число 1. Можно ли на табло Стекового калькулятора отобразить 2003 при условии, что в программе в предписаниях “добавить <**вх**: число> в стек” можно использовать только число 2003002? Обязательно ли в программе в этом случае должна присутствовать команда деления? Можно ли составить программу, которая на табло Стекового калькулятора отображает любое наперед

заданное целое число при условии, что в программе нельзя использовать деление и в предписаниях "добавить <**вх**: число> в стек" можно использовать только числа 2003 и 1997?

4. Придумайте какого-нибудь исполнителя. Опишите его систему предписаний, глобальные объекты. Напишите две существенно разные программы для этого исполнителя.

5. Предположим теперь, что предписание "начать работу" должно встречаться только один раз, причем в самом начале программы, а предписание "закончить работу" должно встречаться тоже только один раз, причем в самом конце программы. Можете ли вы придумать исполнителя с такой системой предписаний, для которого можно написать только одну корректно работающую программу (иными словами, для этого исполнителя существует одна и только одна программа, при выполнении которой не возникает отказа)? А можете ли вы придумать исполнителя с такой системой предписаний, для которого можно написать только определенное количество (например, 22051921050419251707195 2) корректно работающих программ (иными словами, при выполнении этим исполнителем всех остальных программ возникнет отказ)?