Робин Джонс Ян Стюарт

Откройте для себя язык Си

rebus_x 2017
Неофициальный перевод на русский язык
Не является изданием книги
Не предназначено для продажи
Не преследует коммерческой выгоды
Во имя свободы образования, мышления и распространения информации

Робин Джонс (Robin Jones)
Ян Стюарт (Ian Stewart)
Откройте для себя язык Си
(исх.: "The Art of C Programming", 1987)
Неофициальный перевод-редакция, 2017

Формат листа — A4

ОГЛАВЛЕНИЕ

Предисловие от авторов

Язык программирования Си занял необычное промежуточное место между “ассемблером” и высокоуровневыми языками, вобрав одновременно преимущества обеих сторон. Эта книга предназначена как для начала освоения языка, так и для приобретения мышления, которое бы существенно упрощало работу с ним. Вероятно, желающие изучить Си уже имели опыт работы с высокоуровневыми языками наподобие Бейсика или Паскаля, и потому постараемся учесть этот опыт и извлечь из него пользу. Так что предполагаем, что обозначения в коде программы простейших арифметических действий будут читателю знакомы и понятны, а также — что читатель знаком с такими понятиями, как циклы и ветвления (операторы условия), которые используются в большинстве языков высокого уровня.

Но конечно, этого будет мало. Если вы желаете заговорить на иностранном языке, то даже самый лучший перевод по словарю полноценно вам не поможет, поскольку сам целевой язык построен иначе, чем ваш родной. Подобно этому, пусть даже имея на руках некий “словарь” языка Си, нельзя начать программировать на Си с мышлением, по привычке оставшимся от Бейсика. Однако, если изучаемый язык и ваш родной относятся к одному языковому древу, то опора на такое родство станет, особенно поначалу, существенным подспорьем; оттолкнувшись от того, что знакомо, вы обретёте решимость и опыт, необходимые для постижения тех сторон изучаемого языка, которые уже не имеют с вашим родным ничего общего.

Наш подход к изучению Си будет подобен вышеописанному. Мы не будем бросаться сразу изучать все-все возможности Си, пусть даже это позволило бы нам сразу писать краткий и изящный код. Как правило, краткую запись кода труднее читать, и она может сильно смутить новичка. Полноценное использование таких возможностей языка начнётся тогда, когда читатель освоится и будет к этому готов.

Наша книга условно состоит из двух частей. Главы с 1-й по 13-ю повествуют об основных возможностях языка — с примерами кода, задачами, а также (где это уместно) более обширными заданиями, призванными проверить качество понимания прочитанного. В главах с 14-й по 17-ю, на основе предыдущего содержания, рассказано о разработке самых настоящих приложений[1]. В конце концов, компьютерные программы призваны упрощать жизнь человека, а Си по праву рождения — язык системного программирования. (В частности, он был разработан для написания операционной системы “UNIX”. Так что этот язык прекрасно подходит для разработки прикладных программ.) И задачи, выбранные нами, требуют создания таких программ: это арифметика с рациональными числами[2], так называемая черепашья графика[3], и выдача случайных чисел.

Ещё одна причина выбора таких задач в том, что они позволяют показать разные применения языка. Для арифметики с рациональными числами используются арифметические преобразования битов[4], а для получения случайных чисел — логические преобразования битов[4]. Для “черепашьей графики” потребуется применение сложной математики.

Книга содержит два приложения. Первое бегло рассматривает некоторые оставшиеся вопросы возможностей языка. Второе представляет собой краткий справочник. Наша книга скорее является пошаговым руководством с объяснениями, а не справочником; этот раздел предусмотрен на случай, если что-либо из рассмотренного в книге потребуется освежить в памяти.

---

1 На слове “приложений” делайте своим ожиданиям поправку на время написания книги. Ни о каких даже простейших графических интерфейсах речи не идёт. (прим. перев.)

2 Целые числа, а также те, которые можно записать в виде обыкновенной дроби. (прим. перев.)

3 Устоявшийся дословный перевод понятия “turtle graphics”. Подробнее — в главе 15. Однако учтите, что авторы прибегают к сторонним средствам создания графики, которых не предусмотрено в языке Си по стандарту. (прим. перев.)

4 Понятия разъяснены далее в книге, в главе 6. (прим. перев.)

Избранный нами подход к повествованию делает неизбежным то, что в первых главах код может кому-нибудь показаться неуклюжим. Напоминаем, что мы умышленно ограничим себя в количестве тех средств языка, которыми будем пользоваться поначалу. Но к концу книги читатель окажется неплохо осведомлённым о возможностях Си.

Фолкстон, гр. Кент Робин Джонс
Ковентри, гр. Уорикшир Ян Стюарт

Предисловие от переводчика

Я начал освоение языка Си именно с этой книги, и на мой взгляд, она хорошо справляется с задачей первоначального введения в язык, знакомства читателя с ним. Даже несмотря на время её написания (1987 г.) она устарела меньше, чем можно было бы ожидать. Браться за Кернигана-Ритчи, по моему мнению, лучше уже имея представления о языке, а не с полного нуля.

Увы, при чтении этой книги (которая сейчас перед вами) по мере удаления от начала усиливается впечатление, что авторы дописывали её во всё бóльшей спешке. Как следствие, многое пришлось домысливать, а затем дописывать за них; но так как при переводе это занятие весьма неблагодарное, то какие-то места в книге могут всё равно показаться незаконченными, а какие-то мысли авторов — неразгаданными. Поэтому я бы не назвал данную книгу источником, повествующим о мышлении, о подходах к программированию на Си, способах решения тех или иных задач; скорее это хорошее пособие, показывающее именно правила самого языка и имеющиеся в нём средства — с примерами. Именно этой стороне повествования я бы советовал уделить основное внимание.

Я бы даже не называл данный файл только переводом книги, скорее это её неофициальная русскоязычная редакция. Были выброшены за ненадобностью иллюстрации и эпиграфы из произведений Кэрролла, исправлены многочисленные опечатки и ошибки в примерах кода[1], тут и там вставлены важные (на мой взгляд) уточняющие примечания, внесены необходимые поправки и уточнения там, где речь шла о чём-либо устаревшем.

На время появления этой книги стандарт "ANSI C" (C89) ещё только разрабатывался, поэтому о стандартах языка в книге ничего не говорится. Я тоже не стал ничего добавлять по этому поводу, чтобы не забивать стандартами голову читателя; с ними можно ознакомиться при дальнейшем самостоятельном обучении.

Совет. Лучше не пробовать сразу же по ходу первого чтения предлагаемые примеры кода. Пусть первое чтение будет ознакомительным, и пусть это знакомство будет по возможности полным, чтобы сначала вы научились читать код. Книга написана сравнительно простым языком, поэтому при внимательном чтении всё будет понятно.

С надеждой на то, что данный перевод окажется полезен ещё кому-то, кроме меня,

rebus_x

---

1 За время перевода я твёрдо убедился, что большинство авторов книг по программированию соблюдает негласный заговор, о котором нигде не говорится и нигде невозможно прочитать. Заключается он в том, что в некоторых примерах кода намеренно делаются опечатки и ошибки, приводящие к неработоспособности или неправильной работе этого примера. Есть точка зрения, что подобные вещи — мол, потрудись прежде чем хоть что-то заработает — психологически готовят программиста к будущим танцам с бубном. (Как будто компьютерщикам мало этих танцев без всякого программирования.) С моей точки зрения, это вопиюще неверный подход, потому что цель книг, как правило — объяснить и показать, как работают в языке те или иные средства. Программирование — не самая простая вещь для освоения, и дополнительное запутывание и смятение читателя никак не способствуют пониманию.

ГЛАВА 1

Компиляторы и интерпретаторы

Любая программа, написанная на высокоуровневом языке, неизбежно подлежит переводу на машинный код, понятный тому процессору, на котором эта программа будет выполняться. Для решения этой задачи в основном используют два подхода (на самом деле их больше, но остальные подходы кроме означенных двух сейчас для нас несущественны).

Первый. Программа хранится в виде исходного кода на запоминающем устройстве компьютера. Затем, при запуске этой программы, она считывается построчно; считанная строка кода преобразуется в машинный код, после чего машинный код сразу же выполняется. При преобразовании в машинный код следующей строки область памяти с машинным кодом от предыдущей строки затирается. Выходит, что если какая-то строчка кода находится внутри цикла, выполняемого 200 раз, то каждый из этих двухсот раз эта одна и та же строчка будет переведена. Действующая таким образом переводящая программа называется интерпретатором[1], и этот подход должен быть хорошо знаком каждому[2], кто пользовался компьютерами "Commodore 64", Макинтош, "IBM PC", а также любым другим домашним компьютером.

Понятно, что этот подход медленный и неоправданно затратный с точки зрения мощностей компьютера. К тому же перевод кода происходит прямо во время работы программы, что ещё больше замедляет её выполнение. Правда, одно достоинство всё же есть. Так как программа (скажем, на Бейсике) выполняется интерпретатором непосредственно из исходного кода, то есть возможность быстро внести изменения в этот код, и снова запустить программу. (Возможно, вы думаете — "чего же ещё желать?". Сейчас узнаете, а заодно увидите новые сложности.)

Другой путь очевиден: переводить весь исходный код в целую законченную программу на машинном коде, после чего запускать этот машинный код. Уже ясно, что это намного быстрее. В этом случае переводящая программа называется компилятором (по-русски — сборщиком).

Как правило, для Бейсика применяются интерпретаторы, а для Си компиляторы. Нельзя сказать, что не существует компиляторов для Бейсика и интерпретаторов для Си; они есть, но встречаются редко.

Интерпретаторы отлично подходят для разработки, поскольку изменения вносятся в код быстро; по этой же причине они хороши для новичков. Однако для опытного ~~программиста медленное выполнение кода~~ компьютерщика медленная работа компьютера хуже, чем что-либо другое на свете. Пожалуй, лучший случай — это когда среда разработки предоставляет как интерпретатор, так и компилятор для данного языка. Тогда можно отлаживать программу с помощью интерпретатора, а после отладки собрать в исполняемый вид. Однако и здесь поджидает подлянка: чаще всего интерпретатор и компилятор данного языка, соседствующие на одном компьютере, переводят код по-разному, воспринимая его как разные диалекты[3] данного языка программирования, вследствие чего различия получаемого машинного кода могут быть впечатляющими. И потому то, что видится безошибочным интерпретатору, для компилятора может оказаться переполненным ошибками кошмаром.

---

1 Справедливости ради, сегодняшние интерпретаторы не настолько тупы, но в корне они не поменялись. (прим. авт.)

2 Компьютеры тех времён часто приветствовали пользователя экраном, на котором сразу можно было писать программу — потому что без прочих уже готовых программ больше ничего делать было нельзя. (прим. перев.)

3 *Разновидности одного и того же языка программирования, в которых различается работа тех или иных обозначений. (прим. перев.)

Программа для набора текста

Писать исходный код в чём-то придётся. Интерпретатор Бейсика предоставляет встроенный текстовый редактор, позволяющий правильно хранить набранный код, править его и прочее. У компиляторов, как правило, такого не предусмотрено, но если вы где-то раздобыли компилятор — возможно, там же неподалёку есть текстовый редактор. Так или иначе, без него вы работать не сможете.

Как выглядит работа при использовании компилятора

Последовательность будет приблизительно такая:

1. Загрузить программу, в которой вы предпочитаете набирать код.

2. Набрать его.

3. Сохранить на запоминающем устройстве вашего компьютера.

4. Загрузить компилятор и сообщить ему, где взять исходный код (то есть подать файл с исходным кодом ему на ввод). После чего компилятор займётся переводом. Скорее всего, он будет делаться в несколько проходов (исходный код будет просмотрен компилятором несколько раз). В зависимости от дружелюбности, трудолюбия, настроения вашего компьютера — он или самостоятельно запустит все эти проходы, или каждый из них вам придётся вызывать вручную. В конце концов на запоминающее устройство будет записана ваша программа в виде машинного кода. Однако если в вашей программе используются какие-либо библиотечные функции, то их кода в записанной программе не окажется. Поэтому список действий продолжается.

5. Загрузить построитель и подать ему на ввод только что полученный файл с машинным кодом программы. Построитель будет искать в файле использование тех функций, которые сами в нём не определены (тела самой функции в программе нет, а есть только её использование). Если такие есть — построитель ищет функции с этим именем в библиотеке, и делает в программе ссылку на эту библиотеку в случае успеха. Конечным действием также будет запись файла программы в виде машинного кода на запоминающее устройство компьютера.

6. И вот у вас есть файл программы, который можно запустить. Если в качестве запоминающего устройства используется жёсткий диск, то всё вышеописанное происходит довольно быстро (и всё равно нудно. Я предупреждал о сложностях, ожидающих при работе с компиляторами. — прим. авт. 1987). Но если вы используете кассету — то вам скорее исполнится двести лет, чем у вас хоть что-то успеет заработать. Если разработчик компилятора смилостивился над вашими нервами, то он предусмотрел возможность записи получаемых файлов в оперативную память, а не на кассету. Но часто это не делается по той причине, что очень объёмные программы в выделяемую область оперативной памяти не помещаются.

Если вы не всё поняли из вышеизложенного — это не страшно. Пока вы не освоились с имеющимся у вас компилятором Си, вам может быть неясно, зачем вообще нужны все эти действия.

Как я уже писал, для каждого компилятора последовательность действий будет своя. Ниже приведён список действий, цель которых — скомпилировать и запустить программу `FRED.C` (это файл с исходным кодом на Си — прим. перев.) при помощи компилятора "BD Software C", под управлением операционной системы "CP/M":

Приглашение ко вводу	**Ввод пользователя**	**Описание**
`A>`	`CC1 FRED.C`	Файл с исходным кодом `FRED.C`, находящийся на диске `A`, подан на ввод первой части компилятора (`CC1`). Вследствие этого создаётся файл `FRED.CCI`.

Приглашение ко вводу	Ввод пользователя	Описание
A>	CC2 FRED	Файл FRED.CC1 подан на ввод второй части компилятора. Вследствие этого создаётся файл FRED.CRL.
A>	CLINK FRED	Вызван построитель, задача которого — добавить в файл FRED.CRL ссылки на библиотечные функции. Полученное записывается в файл FRED.COM.
A>	FRED	Выполняется запуск полученной программы в виде файла FRED.COM.

В свежих (1987 — прим. перев.) версиях этого компилятора его "половинка" CC1 сама передаёт работу CC2, если не возникло никаких ошибок, так что количество действий стало меньше на единицу. На случай если вы не работали с "CP/M": приглашение A> указывает на текущее расположение — диск A, — и означает, что система ожидает ввод.

Вышенаписанный пример подразумевает, что всё получилось без каких-либо ошибок. Ошибки могут случиться на любом этапе компиляции, в случае чего появятся соответствующие сообщения. В этом случае следует понять, в чём именно дело, загрузить исходный код в текстовый редактор и внести исправления. Теперь видно, что отладкой заниматься не так просто, как в случае с интерпретатором. Необходимо гораздо более внимательно отслеживать вами написанное, ведь даже замена в каком-то месте кода запятой на точку с запятой может занять несколько минут.

Всё это — не попытка отговорить вас от использования компиляторов; полученные преимущества оправдают вложенные усилия. Просто будьте готовы, что внимания к Си с компилятором потребуется много больше, чем к Бейсику с интерпретатором.

Всем, кто читает в 21-м веке и позже

Выше авторы расписали работу при раздельном использовании компилятора, построителя и текстового редактора. Сегодня работа происходит в средах разработки (IDE), которые чаще всего сочетают в себе текстовый редактор и построитель, а также, если повезёт, ещё и компилятор с отладчиком.

Из опыта могу посоветовать среду "Code::Blocks" (свободное ПО, есть под все распространённые ОС), создатели которого также предлагают сборку со встроенным компилятором "GCC"; примеры кода из этой книги мною проверены именно в этой среде. Если использовать отдельный текстовый редактор, то отлично показывает себя программа-блокнот "Notepad++", в которой можно включать режимы подсветки синтаксиса для множества разных языков, включая Си.

Кстати

Думаю, вы заметили, что выше мы назвали себя "я". Мы посчитали, что слово "мы" какое-то напыщенное, мешает непринуждённости. Далее в книге слово "мы" будет означать — "я и вы (читатель)". Если кому-нибудь из читателей слово "я" от лица двух авторов книги видится неуместным, то они могут мысленно призвать возможности Си и так же мысленно выполнить следующее:

```
#define Я Мы
#define я мы
```

(Объяснение, что это значит, вы встретите в главе 6.)

ГЛАВА 2

Костяк программы на Си

Рассмотрим теперь, как выглядит написанная на Си программа. Чаще всего программа на Си — это последовательность функций. Функция (по-русски можно назвать преобразователем) — это примерно то же самое, что подпрограмма в Бейсике. Основное различие в том, что функция в Си умеет передавать код возврата в ту программу или другую функцию, которая её вызвала, а также — принимать на ввод значения от вызывающей программы или функции.

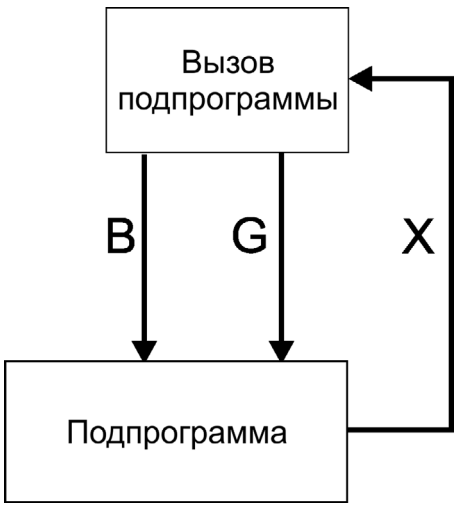
Например, взглянем на подпрограмму в коде Бейсика, задача которой — вычислить неизвестный катет `A` в треугольнике с углом 90 градусов, при условии что величина гипотенузы `G` и другого катета `B` известны:

```
1000 A = G*G - B*B
1010 IF A < 0 THEN PRINT "Треугольник не складывается": END
1020 X = SQR(A)
1030 RETURN              возврат из подпрограммы
```

Видно, что для возможности работы этой подпрограммы требуются подготовить два значения в виде переменных `G` и `B`. Также следует помнить, что искомое значение будет записано в переменную `X`. Учитывая всё это, для использования подпрограммы и выдачи искомого значения я могу написать следующее:

```
100 G = 7: B = 3
110 GOSUB 1000           переход в подпрограмму
120 PRINT X
```

Подпрограмма используется точно так же, как функция, поскольку мы передали ей значения (`G` и `B`), и она тоже вернула значение (`X`).



Не по-русски говоря, каждая функция на вход принимает аргументы (в данном случае `G` и `B`), и возвращает какое-то значение. Здесь важно подчеркнуть, что именно сама функция хранит то значение, которое она вернёт (а не какая-то сторонняя переменная).

Функция на Си, выполняющая ту же задачу что и вышеприведённая подпрограмма на Бейсике, может выглядеть так:

```
int katet_tr(g, b)
int g, b;
{
    int a, x;
    a = g*g - b*b;
    if (a < 0)
    {
         printf("Треугольник не складывается");
         exit(1);
    }
    x = sqrt(a);
    return x;
}
```

(Законченная программа с этой функцией, которую можно опробовать, встретится в главе далее. — прим. перев.)

Это знакомо

В целом вышенаписанный кусок кода на Бейсик непохож, но некоторые обозначения — почти те же. Скажем,

```
a = g*g - b*b;
```

отличается от такой же записи в Бейсике только точкой с запятой в конце. Условие

```
if(a < 0)
```

довольно похоже на

```
IF a < 0 THEN
```

, так же как и запись

```
printf("Треугольник не складывается");
```

напоминает

```
PRINT "Треугольник не складывается"
```

Заметные перемены

Теперь обсудим отличия. В первую очередь записано имя создаваемой функции `katet_tr()`, а номер строки кода вообще отсутствует. В Си номеров строчек кода нет. Имя функции, как и другие условные имена, которые даются в коде чему-либо, может состоять из (исключительно) латинских букв (желательно строчных; причину прочитаете в главе 6), знака нижнего подчёркивания _ , а также цифр 0-9 — но имя не может начинаться с цифры.

Количество читаемых[1] знаков в имени отличается в разных средах. Пределом является 6 знаков[2], поэтому в имени создаваемой функции будем считать значимой только часть `katet_`.

После имени создаваемой функции в скобках через запятую перечисляются её аргументы (входные данные).

Последующие две строки, начинающиеся со слова `int`, мы пока обойдём своим вниманием. Однако между ними стоит фигурная скобка `{` , о которой и поговорим через несколько абзацев.

Простейшие и составные действия

Вы должно быть заметили, что каждая строчка в коде завершается точкой с запятой. Но вовсе необязательно, чтобы точка с запятой приходилась именно на конец строчки кода. Например, вот способ присвоить значения в одной строке сразу трём переменным:

```
p = 3;  r = 7;   z = 0;
```

И это очень напоминает Бейсик:

```
20 P = 3:        R=7:      Z=0
```

В Си точка с запятой является указанием конца действия. В Бейсике такими разделителями действий являются двоеточие и разрыв строки. А в Си перенос строки не имеет для кода вообще никакого значения. Например:

```
p = 3;
```

или

```
p =
3;
```

или

```
p
=
3
;
```

— все эти записи означают одно и то же и работают одинаково успешно.

Как следствие, сложные действия можно для зрительного удобства разбивать на несколько строчек. В Бейсике такое не пройдёт. Представьте, что вы пишете следующее:

`2010 IF a > 7 OR b = 3 OR c > 0 THEN` (и далее последовательность действий, которая не умещается в одной строке)

---

1 Если имя выходит за пределы количества читаемых знаков, то из него читается предельно допустимое их количество, а остальная часть такого сверхдлинного имени отбрасывается. Компилятор будет работать только с оставшейся прочитанной частью. (прим. перев.)

2 Авторы книги осторожничают. Такая краткость (6 читаемых знаков) предусмотрена только для имён, объявленных в каком-то другом файле исходного кода, и к которым делается обращение из данного файла; такие имена помечаются ключевым словом `extern` (о нём в конце книги). В именах, которые используются в пределах только данного файла исходного кода, в большинстве случаев читаемым является 31 знак (в остальных случаях больше; всё зависит от используемой среды и компилятора). (прим. перев.)

В конце концов, когда вы понимаете что ничего из этого не выйдет, приходится писать подпрограмму.

Как же тогда написать оператор условия? Ведь выходит, что между собственно условием и действием, которое оператор должен выполнить, можно поставить сколько угодно разрывов строк. Всё просто! Простейшие действия (которые завершаются точкой с запятой) можно объединять в ряд — это называется составное действие.

Составное действие — это ряд простейших действий, заключённый в фигурные скобки. Составные действия можно писать везде, где можно написать простейшие. Любая функция на Си представляет собой составное действие, поэтому содержимое функции заключается в фигурные скобки.

Читая эту книгу, в примерах кода вы видите некоторое форматирование — но оно делается всего лишь для удобства восприятия. Любые пробелы, табуляции и разрывы строк компилятор просто пропускает, поэтому зрительно располагать куски кода вы можете так, как вам удобно и как вам угодно.

Как уже было сказано, в операторе условия используется как раз составное действие.

```
if (a < 0)
{
    printf("Треугольник не складывается");
    exit(1);
}
```

Если значение переменной `a` меньше ноля, то требуется выполнить два действия. Чтобы представить два действия как одно — они объединяются в составное действие. Обратите внимание, кстати, что ключевые слова (`if`, `printf`, `exit`) написаны с маленькой (строчной) буквы.[1]

Безответные функции

Прямо сейчас не стану рассказывать о функции `printf()`. (Да, это тоже функция. Имя функции отличается от ключевого слова или другого имени наличием скобок.) Обратим своё внимание вот на эту строчку:

```
exit(1);
```

И это тоже функция. Её задача — вернуть управление в операционную систему (выйти из программы; наблюдается подобие слову `END` в Бейсике). В операционную систему функция `exit()` передаёт значение (это называется кодом возврата). Как правило, используют код возврата 0[2], если программа выполнилась успешно, и 1 (или другое число, отличное от ноля) — если в ходе выполнения возникли ошибки. Однако функция `exit()` не возвращает значение в вышестоящую (вызвавшую её) функцию — только в операционную систему. И это вполне законно. Не во всех случаях обязательно возвращать значение в вызвавшую или вышестоящую функцию.

---

1 В Си заглавные и строчные знаки являются разными знаками. Поэтому `if` — ключевое слово, а `If`, `IF`, `iF` — произвольные имена. (прим. перев.)

2 Обычно это понимают как "при выполнении программы возникло 0 ошибок", если код возврата 0. Но прочие коды возврата могут означать всё что угодно. (прим. перев.)

Возврат значений

В случае с функциями, возвращающими значения вышестоящим \ вызывающим функциям, мы можем воспользоваться значением, которое содержит переменная. Допустим, значение было записано в переменную `x`:

```
x = sqrt(a);
```

Это значение возвращается вышестоящей \ вызвавшей функции так:

```
return x;
```

При условии того, что функция `katet_tr()` уже написана, я могу обращаться к ней из тела другой функции, например так:

```
isk_katet = katet_tr(gipot, izv_katet);
```

И сработает это следующим образом:

1. Значение переменной `gipot` будет присвоено значению переменной `g` (которая используется внутри функции `katet_tr()`).

2. Значение переменной `izv_katet` будет присвоено значению переменной `b` (также использующейся внутри функции `katet_tr()`).

3. Функция `katet_tr()` вычисляет значение и записывает его в переменную `x`.

4. Значение переменной `x` возвращается вызвавшую в функцию, в которой это значение присваивается переменной `isk_katet`.

Функция "main()"

Если все вышеприведённые куски кода сравнить со среднестатистической программой на Бейсике, то можно заметить, что собственно программы мы ещё ни разу не видели. Так как в Си повсюду функции, вы скорее всего не удивитесь, узнав, что в каждой программе на Си должна быть функция, которая называется `main()`. Она вызывает все последующие функции, используемые в программе.

Теперь напишем нашу программу:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int katet_tr(g, b)
int g, b;
{
    int a, x;
    a = g*g - b*b;
    if (a < 0)
    {
         printf("Треугольник не складывается");
```

```
        exit(1);
    }
    x = sqrt(a);
    return x;
}

int main()
{
    int isk_katet, gipot, izv_katet;
    gipot = 13;
    izv_katet = 5;
    isk_katet = katet_tr(gipot, izv_katet);
    printf("Длина составляет %6d единиц(ы)", isk_katet);
    return 0;
}
```

(Так как это первый пример кода законченной программы в книге, вы можете пожелать попробовать его в действии. Поэтому в самом начале кода я дописал несколько необходимых строчек, об использовании которых сказано в книге значительно позже (см. главу 6, раздел "**#include**").

Дело в том, что в программе использованы библиотечные функции — `printf()`, `exit()` и `sqrt()`. Для того чтобы ими успешно пользоваться, необходимо в коде программы иметь их описание, которое содержится в служебных файлах данных о библиотеках. Функции `printf()` необходимо содержимое файла `stdio.h`, функции `exit()` — содержимое файла `stdlib.h`, функции `sqrt()` — содержимое файла `math.h`.

Кроме того, в сообщениях использованы русские знаки. Программа будет запускаться в консоли; в "Windows" консоль — командная строка — по умолчанию имеет кодировку "cp866", а среда разработки, по умолчанию — "cp1251". Поэтому русскоязычные сообщения в консоли винды выведутся кракозябрами. Чтобы этого избежать, можно вместо функции `printf()` использовать функцию `wprintf()`, которая работает с расширенной кодировкой знаков ввода — в наши дни это Юникод. Подробнее об использовании функций ввода \ вывода с приставкой "w", таких как `wprintf()`, читайте в Сети; в этой книге работа с ними не рассматривается.

Другой способ заставить консоль винды отображать русские знаки ввода — отправить в неё указание смены кодировки. В начале программы (после объявления переменных) напишите: `system("chcp 1251 > nul");`. Функции `system()` необходимо содержимое файла `stdlib.h`.

Внимание: на точечный шрифт, установленный в консоли по умолчанию, данный способ не подействует. Необходимо зайти в "Свойства" консоли и выбрать другой шрифт для отображения в ней.

Этим способом можно пользоваться всегда, когда в программе необходим вывод русских букв. В дальнейших примерах кода я не буду про него напоминать, несмотря на присутствие русскоязычных сообщений и русских знаков. Впрочем, вы всегда можете набирать любые сообщения транслитом. — прим. перев.)

В теле функции `main()` даются значения переменным `gipot` и `izv_katet`, после чего делается вызов функции `katet_tr()`; отработав, та возвращает значение, после чего это значение сразу присваивается переменной `isk_katet`. Затем это значение выводится на экран. (Повторю, что функцию `printf()` мы разберём в будущем.) Если мы не задумывали возвращать в среду исполнения программы (т.е. операционную систему) какой-либо код возврата, то писать `return` в конце тела функции `main()` необязательно[1].

Обратите внимание, что никаких аргументов (входных данных) в функцию `main()` мы не подали, скобки за её именем пусты. Но ставить эти скобки необходимо (чтобы имя воспринималось как имя функции — прим. перев.).

Перед телом функции `main()` следует определение функции `katet_tr()`. Нерассмотренными остались ещё три функции:

```
printf()
exit()
sqrt()
```

Первые две всегда имеются в библиотеке, и построитель создаст ссылки на них. В случае с `sqrt()` (функция для вычисления квадратного корня) есть вероятность, что такую функцию придётся писать самостоятельно. Как это сделать — разберём позднее.

Виды данных

Неизвестным осталось только одно слово в коде: `int`. Оно необходимо, чтобы указать вид данных "целое число" для объявляемой переменной или ряда переменных[2].

В Бейсике вид данных для переменной указывается через её имя. Как правило, вида данных там всего два. Если в конце имени переменной поставлен знак $, то переменная будет содержать строковое значение; если же его нет, то переменная будет содержать число (если точнее — число с плавающей точкой). В некоторых исполнениях Бейсика есть возможность приписывать к имени переменной знак % , тем самым указывая для неё вид данных "целое число".

В Си вид данных для переменной должен быть определён при её создании. Собственно имя может быть произвольным, до него компилятору дела нет. Подробный перечень видов данных рассмотрим потом, а сейчас остановимся на двух:

```
int
char
```

Первый, как уже сказано, означает "целое число". Второй, что предсказуемо, предназначен для одиночных знаков ввода (букв, цифр, пробелов, прочих). Например:

1 Строчка со словом `return` дописывается многими компиляторами в конце тела функции `main()`. Но никаких других функций это не касается. Кроме того, это делается не всеми компиляторами, поэтому чтобы избежать предупреждения при компиляции, лучше её писать. (прим. перев.)

2 И не только. Вид данных функции определяет вид данных значения, которое она возвращает. Поэтому в приведённых примерах кода у функций тоже подписан вид данных. (прим. перев.)

```
int a, dva, nomer;
char c, razdelitel;
```

Это пример объявления переменных, где переменные `a`, `dva` и `nomer` будут содержать целые числа, а `c` и `razdelitel` — знаки ввода.

Объём переменной вида "целое число" в оперативной памяти зависит от используемой среды и компилятора, но в большинстве случаев он равен 32 битам[1], то есть храниться могут значения от -2147483648 до 2147483647.

Для вида данных "знак ввода" объём составляет 1 байт. Переменные `c` и `razdelitel` хранят по одному знаку. Способа представления не одного-единственного знака, а строки мы с вами пока не узнали.

Область действия переменных

Как только вы напишете имя переменной в Бейсике, ему будет назначена область памяти, и дальнейшие упоминания этого имени будут означать обращение к этой области памяти. Казалось бы всё верно, однако такой подход означает, что если такое же имя переменной будет использовано в подпрограмме, то эти две переменные будут ссылаться на одну и ту же область памяти, перезаписывая значения в ней, что приведёт к ошибкам.

В Си это неудобство довольно просто решено. Имя и значение переменной, объявленной внутри функции, действительны только в пределах этой функции. Говорят, что область действия переменной ограничивается функцией, в которой она находится. В вышеприведённом примере программы переменные `a` и `x` (и их имена) не распространяются за пределы функции `katet_tr()`. Переменная `gipot` действует лишь в пределах функции `main()`, поскольку она объявлена именно там. Если бы у нас в `main()` и в `katet_tr()` было по одной переменной `y`, то эти две переменные были бы друг от друга полностью независимы.

Поэтому нам и нужны средства передачи данных между функциями (скобки для ввода аргументов \ входных данных, и слово `return`). Как только происходит выход из функции, теряются значения переменных, которые внутри неё действуют.

Можно также заметить, что переменные, которые передаются на обработку в функцию `katet_tr()`, отделены от переменных, которые используются внутри неё:

```
int katet_tr(g, b)
int g, b;
{
    int a, x;
```

Вместо этого вы, возможно, ожидали бы увидеть следующее:

```
int katet_tr(g, b)
{
    int g, b, a, x;
```

Дело в том, что вид данных аргументов в определении функции требуется указывать сразу после их упоминания, до тела функции. Если этого не сделать — Си по умолчанию назначает им вид данных "целое число". Если затем в теле функции встретится определение вида данных для этих переменных, то это будет воспринято как переопределение вида данных, что является нежелательным.

1 В соответствии с сегодняшним днём (2017 г. от Р.Х.). В исходной книге значится 16 бит. (прим. перев.)

Глобальные переменные

Конечно, могут быть случаи, когда предпочтительно, чтобы все функции в программе имели доступ к одной и той же переменной. Такие доступные для всех функций переменные называются глобальными. Чтобы создать глобальную переменную, её необходимо объявить до начала любой функции, включая функцию `main()`.

Пример кода с такой переменной:

```
int a;
int funktsiya(b, c)
int b, c;
{
    ...
    a = a + c;
    ...
}

int main()
{
    int b;
    ...
    a = a + b;
    ...
}
```

Переменная `a` доступна как из `main()`, так и из функции, которую я назвал `funktsiya()`. Поэтому значения переменных `b` и `c` прибавляются к значению из одной и той же области памяти.

(Важно: функция `funktsiya()` только получает значение `a`, но не записывает значение в саму глобальную переменную, а работает с её дублем (так функции работают со всеми поданными на вход переменными, о чём вы прочитаете далее).

Другими словами, описанный приём позволяет заранее объявить переменную с каким-либо именем для всех функций. — прим. перев.)

Умывание рук

Я в этой главе не был до конца точным и не рассказывал всё (причём намеренно). Например, много чего ещё следует пояснить об области действия переменных. В свою защиту скажу, что прямо сейчас моя задача — дать вам представление о коде на Си, без необходимости погружаться в тонкости и подробности. В последующих главах по всему увиденному мы пройдёмся ещё раз, более основательно.

Я например не рассказал, что эту запись

```
x = sqrt(a);
return x;
```

можно заменить вот такой

```
return sqrt(a);
```

, а также что строчки

```
isk_katet = katet_tr(gipot, izv_katet);
printf("Длина составляет %6d единиц(ы)", isk_katet);
```

тоже можно записать короче:

```
printf("Длина составляет %6d единиц(ы)", katet_tr(gipot,
izv_katet));
```

Целесообразно ли применять подобные сокращения повсеместно — вопрос обсуждаемый. Из-за обилия сокращений код может стать весьма трудночитаемым.

printf()

Название этой функции складывается из слов "print formatted", что означает "форматированный вывод". До сих пор я избегал её упоминания. Настало время немного её коснуться (и вновь отложить самые полные разъяснения на потом).

Первым аргументом для этой функции должна быть строка (последовательность знаков ввода, записанная в кавычках). Эта строка будет выведена на экран как есть, кроме случаев когда она содержит знак подстановки % . Например,

```
printf("Я что-то написал");
```

выведет на экран:

```
Я что-то написал
```

Но!

```
printf("Длина составляет %4d см", 1);
```

выдаст такое:

```
Длина составляет ---1 см
```

Минусами условно, только здесь в тексте книги, обозначены оставшиеся три незаполненные ячейки, предназначенные для вывода числа. То есть запись `%4d` означает: "подставить на это место значение, указанное вторым аргументом (в нашем случае 1); воспринимать его как десятичное число (на это указывает буква `d` от слова «decimal»), отвести ему 4 разряда для записи".

Таким же образом

```
printf("Первое значение: %3d; второе значение: %5d", x, y);
```

обеспечит следующий вывод:

```
Первое значение: ---; второе значение: -----
```

Минусами всё так же условно показано, что значению из переменной `x` отведено 3 разряда для записи, а значению из `y` — пять.

Пояснения (комментарии)

Как написать в коде комментарий, я ещё не показывал. Комментарии в Си отнюдь не запрещены, писать их можно где угодно. Всё что требуется — это заключить свой комментарий между последовательностями /* и */ .

Допустим, мне потребовалось оставить примечание, что функцию `sqrt()` я ещё не написал. Вот так это бы выглядело:

```
x = sqrt(a); /* функция ещё не написана !!!!11азазазпыщпыщ */
```

Оставлять разъяснения к своему коду — это хорошая привычка. Но в этой книге комментариев в примерах кода почти не будет, поскольку рассматриваемые примеры и так подробно обсуждаются в тексте.

Стоит понимать, что комментарии должны рассказывать суть, а не указывать на очевидное; бесполезных комментариев кода в этом мире слишком много. У одного опытного программиста я как-то увидел следующее:

```
i = 0; /* присваиваем i значение 0 */
```

Не вру.

(В защиту опытного программиста стоит сказать, что такие комментарии тоже полезны — в тех случаях, когда голова устала и отказывается понимать любой код, а понимает только слова. — прим. перев.)

ЗАДАНИЯ

Настало время немного поупражняться. Эти задания несложные. Если вы захотите воспользоваться функцией для вычисления квадратного корня, то попробуйте исходить из того, что в Си есть такая функция[1].

1. Напишите функцию, вычисляющую площадь треугольника со сторонами `a`, `b` и `c`. Воспользуйтесь формулой

$$A=\sqrt{(s(s-a)(s-b)(s-c))}$$

, где s = 0,5(a + b + c) . Программа должна проверять, что построение треугольника с заданными величинами сторон возможно; в противном случае пусть она выдаёт сообщение "Треугольник не складывается".

2. Что делает эта функция?

```
double preobr(cm)
{
    int cm;
    double in;
    in = cm*2.54;
    return in;
}
```

3. Напишите функцию `obr_preobr()`, действие которой обратно действию вышенаписанной.

4. Напишите программу разложения целого числа на простые множители.

5. Приблизительная скорость света — 300.000 километров в секунду. Астрономическую единицу примем за 150 миллионов километров. Напишите функцию `zaderzhka(e)`, вычисляющую время, за которое радиосигнал достигнет спутника, находящегося на удалении `e` астрономических единиц от передатчика.

---

1 Но для её использования требуется подключить файл данных о библиотечных функциях соответствующего раздела библиотеки, под названием `math.h`. Как уже упоминалось, по поводу включения в код содержимого внешних файлов см. главу 6, раздел "**#include**". (прим. перев.)

ГЛАВА 3

Циклы и операторы условия

Все современные языки программирования предоставляют возможность использовать операторы условия и циклы, чтобы выполнять те или иные действия — один или желаемое количество раз — в случае, когда выполняется требуемое условие. В Бейсике для этого используются ключевые слова `IF...THEN...ELSE` и `FOR...NEXT`.

В Си таких средств больше, и возможности их значительно шире.

if

Прежде всего разберёмся с `if`, поскольку мы его уже упоминали. В общем виде условие с этим оператором записывается так:

```
if (условие)
    действие, если условие выполняется
else
    действие, если условие не выполняется
```

"Действие", разумеется, может быть как простейшим, так и составным. Часть `else` является необязательной. Поговорим о сравнительных обозначениях.

То, что в Бейсике выглядит как

```
IF a = 7 THEN ...
```

, в Си записывается следующим образом:

```
if (a == 7) ...
```

Очень важно использовать именно сдвоенный знак равенства. В противном случае программа будет работать неправильно, причём с точки зрения компилятора код будет выглядеть безошибочным. Си не может самостоятельно определить, принимать ли ему одиночный знак равенства за присвоение значения или за условие "если одно равно другому".

Иначе говоря,

```
a = 7;
```

означает "присвоить `a` значение 7". А такая запись

```
a == 7
```

заставляет Си проверить, равно ли значение переменной `a` семи. В Бейсике же существует неопределённость — одинарный знак равенства имеет два значения. Си этого не допускает, вследствие чего компилятору становится проще работать, а также появляются дополнительные полезные возможности, которые мы скоро увидим.

Условные выражения

Проясню теперь некоторые допущенные неточности.

Си вычисляет значение условного выражения (такого как `x > 3` или `a == 7`), получая либо 0 (условие не выполняется — "ложь"), либо 1 (условие выполняется — "истина"). Выражение

```
x > 3
```

примет вид "истина" (1), если значение `x` равно 5, и примет вид "ложь" (0), если значение `x` равно 3.

Но! Условие считается не выполняющимся только в том случае, если значение условного выражения — "ложь" (0). При любых других значениях действие, написанное после условия, будет выполняться. Примеры:

```
if (x > 0)
    printf("x — положительное число");
```

Записанное в скобках условие очевидно. А вот другой случай:

```
if (x)
    printf("x не равна 0");
```

Запись `if (x)` означает "если значение `x` — любое, кроме 0".

(Заметьте, что в этих примерах нет фигурных скобок. В них нет необходимости, поскольку за условием следует простейшее действие.)

Значение условного выражения можно изменить на противоположное, приписав к нему спереди восклицательный знак. Например,

```
if (x == 0) printf("x равна 0");
```

сработает так же, как и

```
if (!x) printf("x равна 0");
```

Кроме того, восклицательный знак, с той же целью, можно приписывать и к сравнительным обозначениям. Вот полный список сравнительных обозначений:

`==` равно

`!=` не равно

`>` больше

`<` меньше

`>=` больше либо равно

`<=` меньше либо равно

Обратите внимание, что в обозначении "не равно" один из знаков "равно" меняется на восклицательный знак. Также обратите внимание на запись "больше либо равно" и "меньше либо равно". Можно легко озадачить компилятор, написав эти обозначения неправильно.

Логические связки

Условные выражения можно соединять при помощи логических связок `&&` ("и") и `||` ("или"). Выглядеть это может так:

```
if (a > 10 && a < 20)
    printf("значение a соответствует пределам");
```

Значение всего выражения можно обратить:

```
if (!(a > 10 && a < 20))
    printf("значение a вне указанных пределов");
```

Чтобы это сделать, я воспользовался дополнительными скобками. Порядок обработки обозначений `&&` и `||` у разных компиляторов может отличаться, поэтому в случаях, когда обе этих связки используются в одном выражении — целесообразно также использовать скобки, чтобы сделать порядок действий однозначным. Например,

```
if ((x >= 10 && x <= 50) || y == 0)
```

, что означает "если `x` лежит в пределах от 10 до 50, либо если `y` равна 0". Тогда как в случае записи

```
if (x >= 10 && (x <= 50 || y == 0))
```

в первую очередь происходит обработка связки "или", и значение `x` должно быть больше либо равно 10 в любом случае.

Циклы

В Си предусмотрено несколько видов циклов, и все они используют тот же подход к записи условия, что и `if`.

while

Самый простой из циклов Си. Общий вид его записи таков:

```
while (условие)
    действие, если условие выполняется
```

Действие (простейшее или составное) выполняется в случае, если значение выражения-условия — любое, кроме значения "ложь" (0).

Рассмотрим простой пример. Напишем программу, возводящую в третью степень (в куб) числа от 1 до 20. Вот она:

```
#include <stdio.h>


int main()
{
    int chislo, kub_chisla;
    chislo = 1;
    while (chislo < 21)
    {
         kub_chisla = chislo*chislo*chislo;
         printf("%2d %5d\n", chislo, kub_chisla);
         chislo = chislo + 1;
    }
    return 0;
}
```

Разберём написанное по порядку. Объявляются переменные `chislo` и `kub_chisla`, с

видом данных "целое число". После присвоения переменной chislo значения 1 начинается цикл, который выполняется, т.к. chislo меньше, чем 21. (Если бы значение chislo составляло 21, то цикл бы и не начал выполняться. Вспомните теперь цикл FOR-NEXT из Бейсика, который всегда выполняется хотя бы один раз, независимо от начальных условий.)

Вычисляется значение kub_chisla, значения chislo и kub_chisla выводятся на экран, и конечным действием chislo увеличивается на единицу. После этого условие из заголовка цикла вновь проверяется, и до тех пор пока значение переменной chislo не составит 21, эти действия будут повторяться.

Неплохо получилось, согласны? Сделаем ещё лучше.

Быстрое приращение

В коде на большинстве языков часто требуется использовать положительное и отрицательное приращение значений переменных (увеличение или уменьшение значения на единицу). Нам это только что встретилось:

```
chislo = chislo + 1;
```

Если не обращать внимания на точку с запятой, то в Бейсике такое действие выглядит так же. Но Си предлагает для этого действия более краткую запись:

```
chislo++;
```

Читать это можно как "увеличить значение переменной chislo на 1".

Значимость такой записи не только в том, что a++ пишется быстрее, чем a = a + 1. Для компилятора приращение и сложение существенно отличаются; записи a = a + 1 и a = b + c подразумевают схожее количество необходимых действий. Во втором случае необходимо найти ячейку памяти b и взять оттуда значение, найти и взять значение c, сложить эти два значения, найти ячейку a и поместить полученное значение в неё. Выражение a = a + 1 обрабатывается таким же образом. А вот a++ — совсем другое дело: приращение — это всего одно действие в машинном коде, и это значит, что обращение к переменной a будет лишь одно.

Такой подход позволяет компиляторам Си производить машинный код меньшего объёма, вследствие чего итоговая программа выполняется быстрее.

Это не всё. Выполнять приращение переменной можно в любом месте, в котором она упоминается. Поэтому нашу программу можно переписать так:

```
#include <stdio.h>


int main()
{
    int chislo, kub_chisla;
    chislo = 0;
    while (chislo++ < 20)
    {
         kub_chisla = chislo*chislo*chislo;
         printf("%2d %5d\n", chislo, kub_chisla);
```

```
    }
    return 0;
}
```

Теперь `chislo` увеличивается на 1 непосредственно после проверки условия в заголовке цикла. Следовательно, я поменял её первоначальное значение на 0, иначе на первом проходе цикла она бы имела значение 2. Чтобы остановить цикл вовремя, также пришлось изменить верхний порог в условии на 20. Приращение можно выполнять и непосредственно перед проверкой условия, вот так:

```
++chislo
```

В этом случае верхний порог в условии необходимо снова поменять на 21.

Можно произвести ещё одно упрощение нашей программы:

```
#include <stdio.h>


int main()
{
    int chislo;
    chislo = 0;
    while (chislo++ < 20)
        printf("%2d %5d\n", chislo, chislo*chislo*chislo);
    return 0;
}
```

Одним из аргументов функции `printf()` стало вычисление искомого значения. Составное действие превратилось в простейшее. Поэтому фигурные скобки отпали за ненадобностью.

Отрицательное приращение ничем не отличается от положительного, кроме знака:

```
a--
```

Функция "sqr()"

В прошлой главе я упоминал функцию извлечения кадратного корня, сказав, что её в библиотеке вашего компилятора может и не оказаться[1]. Так или иначе — давайте напишем её самостоятельно.

Будем последовательно вычислять более точное приближение. Если исходное число — `a`, и приближение его квадратного корня — `x`, то чтобы вычислить более точное приближение, требуется формула:

```
(x² + a)/(2x)
```

Исходное значение приближения неважно — указанная формула позволит получить более точное. Повторив вычисление по этой формуле несколько раз, можно добиться значения, близкого к искомому ответу.

Вот первая попытка создать функцию `sqr()` на этой основе:

1 В библиотеке она есть, называется `sqrt()`, можно пользоваться. (прим. перев.)

```
int sqr(a)
int a;
{
    int priblizh;
    priblizh = 1;
    while (a != priblizh*priblizh)
         priblizh = (priblizh*priblizh + a)/(2*priblizh);
    return priblizh;
}
```

Без всяких ухищрений. Исходное значение приближения принимаем за 1. Если значение приближения, возведённое в квадрат, не равняется `a` (число, квадратный корень которого мы ищем) — выполняется вычисление по формуле, итог которого становится новым значением приближения. Как только значения `a` и `priblizh` в квадрате совпадут — цикл прекратит выполнение, и значение `priblizh` будет возвращено в вызвавшую функцию.

Однако помните, что эти переменные имеют вид данных "целое число". На всякий случай я придерживаюсь этого вида данных, поскольку некоторые компиляторы Си не поддерживают другие виды данных для чисел. Если серьёзно браться за вычисление квадратного корня — то очевидно, требуется иметь дело не только с целыми числами. В Си на такой случай предусмотрен вид данных `float` ("floating point value" — число с плавающей точкой)[1]. Но пока обойдёмся без него.

Неплохо бы проверить на нескольких числах, как сработает эта функция. Сначала возьмём 16. Исходное значение `priblizh` равно 1, а значит следующее будет равно (1*1 + 16)/(2*1) = 17/2 = 8. Конечно, должно бы получиться 8.5 , но дробная часть отбрасывается из-за вида данных "целое число". Следующий проход выдаст (8*8 + 16)/(2*8) = 80/16 = 5. Следующий: 41/10 = 4. Условие в заголовке цикла перестанет выполняться, произойдёт выход из цикла и возврат значения 4 в вызвавшую функцию.

Однако если значение `a` составит 21, то получается последовательность 11, 6, 4, 4, 4, ... и так далее. Вновь и вновь вычисляется значение 4 , поскольку это — ближайшее предыдущее целое число от квадратного корня двадцати одного[2]. Цикл будет всегда выполняться, поскольку 4*4 никогда не даст 21.

На первый взгляд вывод очевиден: причина происходящего в том, что мы используем только целые числа. Однако даже если бы мы использовали вид данных "число с плавающей точкой", это бы ничего не решило. Для многих чисел значение их квадратного корня можно записать лишь приближённо, и никакой уровень точности не поможет завершить его запись. Мы приходим к выводу, что ожидать правильной работы условия `a == priblizh*priblizh` не следует!

Нам необходимо определить допустимую погрешность для вычисляемого значения. Пер-

---

1 В Си два вида данных для чисел с плавающей точкой — `float` и `double`. Второй отводит числам вдвое больший объём памяти, и вследствие этого имеет бóльшую точность. Поэтому сегодня вид данных `float` почти никто не использует, вместо него повсеместно используется `double`. (прим. перев.)

2 Если вычисление дало нецелое число, то перевод его в целое делается не округлением до ближайшего целого, а отбрасыванием дробной части, какой бы она ни была. Из значения 2.99 будет получено число 2. (прим. перев.)

вый способ это сделать — проверять, лежит ли значение `a` в пределах от `priblizh` в квадрате до `priblizh + 1` в квадрате.

```
while (!(a > priblizh*priblizh && a < (priblizh +
1)*(priblizh + 1)))
```

Но как-то это громоздко. Другой путь — сравнивать текущее значение приближения с предыдущим. Если разница между ними окажется меньше требуемой нам погрешности — произойдёт выход из цикла. Внесём соответствующие изменения:

```
int sqr(a)
int a;
{
    int pred_priblizh, sled_priblizh;
    pred_priblizh = 1; sled_priblizh = 0;
    while (pred_priblizh - sled_priblizh >= 1)
    {
        sled_priblizh = (pred_priblizh*pred_priblizh + a) /
(2*pred_priblizh);
        pred_priblizh = sled_priblizh;
    }
    return sled_priblizh;
}
```

Всё верно?

Нет! Последнее действие в цикле делает неизбежным то, что он выполнится всего один раз! После чего условие не будет выполнено, т.к. уменьшаемое и вычитаемое значения равны.

Возможно, так лучше?

```
int sqr(a)
int a;
{
    int pred_priblizh, sled_priblizh, beskonech;
    pred_priblizh = 1; beskonech = 1;
    while (beskonech)
    {
        sled_priblizh = (pred_priblizh*pred_priblizh + a) /
(2*pred_priblizh);
        if(pred_priblizh - sled_priblizh < 1)
            return sled_priblizh;
        pred_priblizh = sled_priblizh;
```

```
    }
    return 0;
}
```

Здесь условие написано так, что оно всегда будет выполняться. Это происходит потому, что значение переменной `beskonech` равно 1 ("истина"). Можно было написать и прямо, но менее очевидно:

```
while (1)
```

Выход из цикла при необходимости обеспечит оператор условия `if`.

Задача 1

Программа всё ещё неотлажена. По задумке, значение `pred_priblizh` должно быть всегда больше чем у `sled_priblizh`. Однако при первом проходе цикла значение `pred_priblizh` равно 1, а в условии `if` получается выражение 1 - 11 , итог которого явно меньше единицы. Ошибочка!

Обойти это можно, используя подобие функции `ABS` из Бейсика, которая отбросит знак минус у данного числа:

```
if (abs(pred_priblizh - sled_priblizh) < 1)
    return sled_priblizh;
```

Напишите в Си свою функцию `abs()`.

Можно подумать, что цикл `while` применим для любых целей, и нет необходимости в других видах циклов. С какой-то стороны это так. Но есть случаи, когда иное построение цикла более выгодно. Помимо `while`, в Си есть ещё два вида циклов.

do ... while

Общий вид записи:

```
do
    действие
while (условие);
```

Довольно очевидно, что это просто перевёрнутый цикл `while`, в котором проверка условия происходит после выполнения действия. Таким образом, по крайней мере один раз действие будет обязательно выполнено.

Вот как можно использовать этот цикл в нашей функции `sqr()`:

```
int sqr(a)
int a;
{
    int pred_priblizh, sled_priblizh;
    sled_priblizh = 1;
    do
    {
```

```
            pred_priblizh = sled_priblizh;
            sled_priblizh = (pred_priblizh*pred_priblizh + a) /
(2*pred_priblizh);
        }
        while (abs(pred_priblizh - sled_priblizh) >= 1);
        return sled_priblizh;
    }
```

Исчезло неудобство из предыдущего случая, когда в условии происходило вычитание равных значений. Теперь `pred_priblizh` и `sled_priblizh` уравниваются уже после проверки условия.

С другой стороны, до начала цикла приходится давать переменной `sled_priblizh` какое-то значение приближения (в нашем случае 1). Некоторые компиляторы, кстати, не будут ругаться, если вы этого не сделаете. Но об этом надо заботиться, поскольку в противном случае значением переменной станет содержимое области памяти, которая ей отведена (проще говоря, мусор). И дальшейшее поведение программы станет непредсказуемым.

Задача 2

Перепишите программу возведения в куб, используя цикл `do...while`.

for

А вот и старый добрый цикл `for`, не сильно отличающийся от своего собрата в Бейсике. Записывается он так:

```
for (a = 1; a < 50; a++)
{
    ...
}
```

Первое выражение в условии (здесь `a = 1`) — это действие, выполняемое перед запуском цикла. (Важно не путать с первым действием цикла: оно выполняется один раз, перед входом в сам цикл.)

Второе (здесь `a < 50`) — это собственно условие, при котором цикл должен выполняться. И третье (`a++`) — это завершающее действие цикла: оно выполняется при каждом проходе самым последним.

Вышенаписанный пример составлен так, чтобы он был предельно похож на запись в Бейсике:

```
FOR A = 1 TO 49 STEP 1
```

Но в Бейсике запись `STEP 1` необязательна. А в Си третье (завершающее) действие в условии цикла `for` писать необходимо.

Так как это завершающее действие может быть любым, то шаг приращения `a` легко изменить:

```
for (a = 1; a <= 50; a = a + 5)
```

Это сопоставимо с изменением шага `STEP` в Бейсике.

```
FOR A = 1 TO 50 STEP 5
```

Кстати, записать `a = a + 5` можно короче:

```
a += 5;
```

Преимущество этой записи то же, что и у краткой записи приращения — `a` упоминается только один раз. Такая запись допустима для любых знаков арифметических действий. Например:

`y *= 51`	увеличить `y` в 51 раз
`a -= 2`	уменьшить значение `a` на 2
`r /= 3`	уменьшить значение `r` в 3 раза

А это пример цикла `for`, где все части условия произвольные:

```
for (a = 5; i != 60; p *= 3)
```

Завершающее действие, по сути, используется вместо последней строчки в теле цикла.

Задача 3

Скорее всего вы обратили внимание, что когда мне требовалось возвести число в степень — я всегда умножал его само на себя. Причина этого в том, что в Си нет оператора, возводящего значение в степень. В Бейсике с этой целью используется знак ^ , но в Си он имеет другое значение.

Ваша задача — написать функцию `stepen(chislo, n)`, которая бы возводила "число" в степень `n`. Используйте в теле функции цикл `for`.

Ответы

Задача 1

Вот одно из решений:

```
int abs(znachen)
int znachen;
{
    if (znachen < 0)
         znachen = -znachen;
    return znachen;
}
```

Также можно написать

```
znachen *= -1;
```

вместо

```
znachen = -znachen;
```

, однако зачастую лучшим решением будет наиболее читаемый код.

Задача 2

```
#include <stdio.h>
int main()
{
    int chislo;
    chislo = 1;
    do
        printf("%2d %5d\n", chislo, chislo*chislo*chislo);
    while (chislo++ < 20);
    return 0;
}
```

Всё довольно понятно и без разъяснений. Возможно, вы заметите, что приращать переменную `chislo` можно было бы и внутри вызова функции `printf()`. Чисто теоретически да, однако порядок разрешения приращений внутри вызова функции у разных компиляторов отличается. Поэтому в целях надёжности и переносимости кода лучше так не делать.

Задача 3

```
int stepen(chislo, n)
int chislo, n;
{
    int schetchik, itog;
    itog = 1;
    for (schetchik = 1; schetchik <= n; schetchik++ )
        itog *= chislo;
    return itog;
}
```

Конечно, такая функция не подготовлена для вычисления отрицательной степени, но в случае `n = 0` она сработает верно. Переменная `schetchik`, будучи равной 1, окажется больше `n`, цикл не выполнится, и в вызвавшую функцию будет возвращено значение переменной `itog` (1).

ЗАДАНИЯ

1. Напишите программу, которая выдаст таблицу умножения для чисел от 1 до 12

а) с использованием цикла `for`

б) цикла `while`

в) цикла `do...while`.

2. Напишите функцию `kub_koren(a)`, способную извлечь корень третьей степени из числа.

Для последовательного уточнения приближения корня используйте формулу:

$$(2x^3 + a)/3x^2$$

, где `x` — значение приближения.

ГЛАВА 4

Арифметические и логические вычисления

Возможно, разговор об арифметике, когда мы уже вовсю применяем её направо и налево, выглядит запоздалым. Но до сих пор нами использовались знаки арифметических действий, совпадающие с оными в Бейсике (за исключением отдельно рассмотренных обозначений приращения). То есть сейчас на вооружении мы имеем следующее:

`+` сложение

`-` вычитание

`*` умножение

`/` деление

`++` положительное приращение на 1

`--` отрицательное приращение на 1

Остался один знак, предназначенный для взятия остатка от деления нацело: % . Пример:

`8 % 3 = 2` остаток от деления восьми на три — два

Чётные и нечётные числа

Одно из применений взятия остатка — определение, является ли число чётным или нечётным. Напишем функцию `chetn()`, которая должна вернуть единицу, если поданное значение чётное, и ноль — если нечётное.

```
int chetn(a)
int a;
{
    if (a % 2)
         return 0;
    else
         return 1;
}
```

Если `a` содержит нечётное число, то от деления на 2 получится остаток, и вычисление условия (a % 2) не даст 0, т.е. условие выполнится. Функция вернёт значение 0. Если вычисление условия даст 0 ("ложь"), то функция вернёт значение 1.

Теперь можно вызвать `chetn()` из какой-нибудь другой функции:

```
if (chetn(chislo))
    printf("%4d - чётное число", chislo);
```

Как видите, читается всё довольно просто, и к тому же задача вызванной функции видна со стороны. Это хороший образец того, к чему следует стремиться для понятности кода.

Логические преобразования

Людям, знакомым с машинным кодом, должны быть известны логические действия AND, OR, NOT и XOR (соответственно "И", "ИЛИ", "НЕ", "исключающее ИЛИ")[1]. Вероятно, все остальные с ними знакомы меньше. Причиной тому высокоуровневые языки программирования, которые или не вдохновляют на использование этих логических преобразований, или вовсе не дают ими пользоваться.

Вряд ли это к лучшему, поскольку порой это лишает программиста нескольких отличных возможностей. Си даёт вам эти преобразования в руки. Я расскажу о них более-менее подробно. Если вы с этим уже знакомы, то сумеете кратко пробежаться по ниженаписанному, но обращайте внимание на правила записи данных действий в Си.

Большинство примеров кода будет посвящено действиям над знаками ввода, поэтому переменные будут иметь вид данных "знак ввода" (char). Объём переменной этого вида данных в памяти — 8 бит, или один байт.

Кодировка "ASCII"

Скорее всего, вы знакомы с кодами-номерами знаков ввода в таблице кодировки. В Бейсике есть функция ASC, возвращающая код-номер знака. Возможно, вы даже помните какие-то из этих номеров: "A" (забугорная) — 65, "1" — 49, и другие.

Но если размышлять о том, как кодировка "ASCII" выглядит в двоичном коде, то вот вам представление, за что отвечают некоторые биты.

7 (восьмой)	В кодировании знаков ввода не задействован
6 (седьмой)	0 — цифра; 1 — буква
5 (шестой)	0 — заглавная; 1 — цифра \ строчная
4 (пятый)	0 — a-o (лат.); 1 — цифра \ p-z (лат.)

К примеру, если седьмой бит (с номером 6 — круто, да? — прим. перев.) равен 1 (то есть кодируемый знак — буква), то шестой бит определяет, заглавная она или строчная. Для понятности я умалчиваю о знаках наподобие * ? ! , а также об условных сочетаниях наподобие \n.

"И" ("AND")

Логическое действие "И" (AND) поразрядно (по битам) сравнивает два байта и в свою очередь выдаёт 8 бит, которые получаются так. Если в данном разряде \ бите у обоих сравниваемых байтов значится "1", то и в итоговый байт записывается 1 в соответствующий разряд \ бит. Во всех других случаях в данный разряд \ бит записывается 0.

Допустим, имеется два байта, а и б, с такими значениями:

```
а       01101001
б       10111010
```

Тогда сравнение "И" даёт:

```
    a & b = 00101000
```

1 Всё это устоявшийся дословный перевод. Возможно, вам больше понравятся более осмысленные (пусть и более длинные) обозначения, например "ОБА1", "ОДИН1", "ОБРТ" и "ОДИН1НЕОБА" (соответственно порядку в тексте), и вы будете использовать их для себя. Тем не менее, я вынужден пользоваться устоявшимся переводом для избежания путаницы. (прим. перев.)

То есть если и у `а`, и у `б` в данном разряде значится 1, то в итоговый байт в этом разряде также записывается 1.

В примере вы видите знак & , который в Си используется как знак логического сравнения "И". Может, вы вспомните логическую связку `&&` , которая также означает "и", но используется в условных выражениях.

Теперь о том, как это использовать в действиях над знаками в кодировке "ASCII". Представим, что на ввод поступают знаки, причём как заглавные так и строчные — и нам требуется, чтобы все они становились заглавными. Пригодилась бы функция, которая бы занималась этим преобразованием. Напишем её:

```
char vzaglavn(a)
char a;
{
    char maska;
    maska = 223;
    return (a & maska);
}
```

И работает это крайне просто. Мы взяли 223, потому что в двоичном виде это число выглядит так:

```
11011111
```

Просто для наглядности представим обрабатываемые 8 бит в виде последовательности букв:

```
абвгдеёж
```

Если при помощи логического сравнения "И" скрестить число 11011111 и эту последовательность, то получится следующее:

```
аб0гдеёж
```

Шестому биту (с номером 5. Ну удобно же! (нет) — прим. перев.) принудительно даётся значение 0, а все остальные биты остаются прежними. Чтобы переводить буквы в заглавные, нам именно это и нужно. Конечно, мы должны быть уверены, что на ввод поступают именно буквы.

В случаях, подобных нашему, слово "маскирование" применяется часто. Всё, кроме шестого бита, защищено от изменений — находится под "маской". Заметьте, что несмотря на вид данных "знак ввода" Си без споров присваивает переменной `maska` значение 223. Без нашего ведома во время выполнения программы происходит преобразование вида данных.

Могу заметить, что переменная `maska` в вышенаписанном примере является лишней. Так тоже сойдёт:

```
return (a & 223);
```

Задача 1

Напишите функцию `odinak_nach(a, b)`, которая бы возвращала 1 в случае, если у байтов `a` и `b` восьмой бит (номер 7) одинаковый, и 0 — если нет.

"ИЛИ" ("OR")

Это действие отличается от сравнения "И" тем, что для записи в итоговый бит единицы хотя бы один из рассматриваемых битов должен быть равен 1, а не оба сразу. Соответственно, ноль записывается только в случае, если оба рассматриваемых бита равны 0.

```
а       01101010
б       00100011
a | b = 01101011
```

Понимать слово "или" предлагается как "если `а` ИЛИ `б` равны 1, то записать единицу в итоговый бит".[1]

В Си преобразование "ИЛИ" обозначается одной вертикальной чертой | .

Воспользовавшись им, мы можем написать функцию, которая будет преобразовывать поступающие на ввод буквы уже в строчные.

```
char vstrochn (a)
char a;
{
    char maska;
    maska = 32;
    return (a | maska);
}
```

В двоичном и восьмибитном виде число 32 выглядит так:

```
00100000
```

Преобразование "ИЛИ" между этим значением и условной последовательностью `абвгдеёж` даст:

```
аб1гдеёж
```

То есть шестому биту даётся значение 1, что означает строчную букву. Надо сказать, что более совершенные версии функций наподобие созданных нами `vzaglavn()` и `vstrochn()` вероятнее всего есть в библиотеке вашего компилятора Си.

Задача 2

Напишите функцию `odinak_kontsi(a)`, которая бы возвращала 1, если у байта `а` начальный и конечный бит имеют одинаковое значение, и 0 — если нет.

Исключающее "ИЛИ" ("XOR", или "exclusive OR")

Как можно догадаться, это исключительный случай преобразования "ИЛИ". В случае, если оба рассматриваемых бита равны 1, то "исключающее ИЛИ" записывает 0 в итоговый байт.

```
а       01101010
б       00100011
a ^ b = 01001001
```

1 "ОДИН1". Один один. Только один равен 1. Выбирайте, что проще запомнить. (прим. перев.)

Как видно, для обозначения этого преобразования в Си используется знак ^ .

Теперь у нас есть возможность написать функцию `smenaregistra()`, которая будет менять заглавные буквы на строчные, и наборот.

```
char smenaregistra(a)
char a;
{
    char maska;
    maska = 32;
    return (a ^ maska);
}
```

Как вы считаете, что будет в итоге следующего "скрещивания" с помощью преобразования "исключающее ИЛИ"?

```
абвгдеёж
00100000
аб?гдеёж
```

Вне зависимости от значений битов а, б, г, д, е, ё, ж — они останутся прежними. Потому что 1 ^ 0 = 1 и 0 ^ 0 = 0. А вот если бит `в` равен 0, то 0 ^ 1 = 1; если же `в` равен 1, то 1 ^ 1 = 0.

Таким образом, в бите `в` принудительно меняется значение, что даёт нам смену регистра буквы.

"НЕ" ("NOT")

Это преобразование меняет значение каждого бита на противоположное. Например:

```
a   01110001
~a  10001110
```

Знак тильда ~ служит в Си обозначением данного преобразования.

Что бы такого привести в пример. Представьте, что вы создаёте программу, изображающую городское автомобильное движение, и в ней присутствуют светофоры (и корованы — прим. перев.). И у вас объявлены переменные:

```
char krasn, zelen, svet;
krasn = 0; zelen = 255;     жёлтого света нет, только красный и зелёный
```

(Число 255 в двоичном виде выглядит как 11111111.)

Таким образом, вы можете написать что-то вроде

```
if (svet == krasn) ....
```

"Переключать" же свет светофора можно с помощью ну очень простой функции:

```
char perekluch(svet)
char svet;
{
```

```
    return (~svet);
}
```

Задача 3

Сегодня объём байта в 8 бит — устоявшаяся и утвердившаяся величина. Однако в своё время были компьютеры с байтом в 9 бит, в 6 бит — и их было немало. Кто сказал, что на смену восьми битам в байте никогда не придут, скажем, десять?

По крайней мере часть вышеприведённых примеров не заработает на таком десятибитном компьютере.

Иначе говоря, я предлагаю вам подумать, как следовало бы писать код, чтобы разница в количестве бит в байте не была препятствием. Сможете таким образом переписать обрывки кода нашей "программы про светофоры"?

Сдвиги

Если биты в байте сдвинуть на один разряд влево, а образовавшуюся пустоту заполнить нолём — это равносильно умножению числа на 2. Например:

```
              a =       00010011  (19)
сдвиг на бит влево      00100110  (38)
```

В Си такое действие есть. Вот его запись:

```
y = x << 1;
```

Это значит, что сдвиг значения `x` произойдёт на 1 бит. Можно выполнить и больший сдвиг:

```
a = b << 4;
```

Сдвиг на 4 бита влево равносилен умножению на 16.

Смысл этого действия в том, что оно выполняется гораздо быстрее обыкновенного умножения. Таким образом, вычисления можно значительно ускорить. Вот пример функции умножения числа на 10, использующей двоичный сдвиг влево:

```
int umnozh10(a)
int a;
{
    int b, c;
    b = a << 1;
    c = b << 2;
    return (b + c);
}
```

Переменная `b` получает значение `2*a`. А `c` получает значение `4*b`, то есть `8*a`. Таким образом, возвращаемое значение равно 10-кратному значению `a`.

Сдвигать биты вправо тоже возможно.

```
a = b >> 2;
```

Если требуется присвоить изменённое значение той же самой переменной, то можно воспользоваться краткой записью:

```
b >>= 2
```

И конечно, запись для сдвига влево выглядит так же.

В общем случае сдвиг вправо равносилен делению числа на 2, но далеко не всегда. Этот вопрос подробнее обсудим в дальнейшем.

Задача 4

Напишите функцию, которая бы возвращала 1 в случаях, если первые 3 бита в `a` совпадают с последними тремя битами в `b`, либо наоборот — если первые три бита в `b` такие же, как последние три в `a`.

```
a   ***..+++
```

должно быть совпадение хотя бы одной пары

```
b   +++..***
```

Ответы

Задача 1

```
int odinak_nach(a, b)
char a, b;
{
    char maska;
    maska = 128;
    return (a & maska == b & maska);
}
```

Способ без лишней переменной `maska`:

```
int odinak_nach(a, b)
char a, b;
{
    return (a & 128 == b & 128);
}
```

Задача 2

```
int odinak_kontsi(a)
char a;
{
    return (((a & 129) == 0) || ((a & 129) == 129));
}
```

Поясню. 129 в двоичном виде — 10000001 (а ноль — 00000000). Если представить значение `a` как `абвгдеёж`, то

```
a & 129 = a000000ж
```

Наша запись по сути гласит:

```
return (если сравнением "И" получено значение 00000000 либо
10000001)
```

, т.к. если `a` = `ж`, то искомые нами значения — 0 (00000000) и 129 (10000001).

Задача 3

Требуется изменить всего одну строчку:

```
krasn = 0; zelen = ~krasn;
```

В переменной `krasn` все биты будут установлены в ноль — сколько бы их ни было. А преобразование "НЕ" изменит их значение на противоположное.

Задача 4

Я напишу ещё и вторую, вспомогательную функцию `vspm(a, b)`, на которую переложу часть работы.

```
int vspm(a, b)
char a, b;
{
    return (((a & 224) >> 5) == (b & 7));
}

int sovpad(a, b)
char a, b;
{
    return (vspm(a, b) || vspm(b, a));
}
```

(224 = 11100000
7 = 00000111).

ГЛАВА 5

Строки, массивы, указатели

До сих пор мы обходили стороной вопросы массивов, а также обработки строковых значений. В большинстве разновидностей Бейсика строковые значения хранятся в массивах, однако обнаружить и понять это не так просто. В Си взаимоотношения массивов и строковых значений более прозрачны.

Строки и указатели

Строковые значения нам уже встречались — в качестве входных данных для функции `printf()`.

```
printf("Это строка");
```

Ничего неожиданного. Как и в Бейсике, строка заключена в кавычки. У меня должно получиться сделать так...

```
a = "вижу надпись";
```

А после этого так:

```
printf(a);
```

Действительно, всё как в Бейсике. И я правда могу так писать в Си (хотя так лучше не делать — причины вскоре будут ясны). Но в этом случае совершенно не виден подход, применяемый в Си для обработки строковых данных. Что же такого особенного надо знать?

Когда Си встречает действие наподобие

```
a = "вижу надпись";
```

, то в оперативной памяти под строковое значение выделяется последовательность байт:

```
38012   в
38013   и
38014   ж
38015   у
38016
38017   н
38018   а
38019   д
38020   п
38021   и
38022   с
38023   ь
38024   0     знак "нуль", применяемый в кодировке "ASCII" как разделитель
```

Этот пример можно рассматривать как рабочий[1]. Конечно, вместо 38012 может быть ячейка с любым другим номером; важно запомнить, что выделяется неразрывная последовательность ячеек.

Также вы видите знак "нуль", разделитель. Это не цифра 0 — имеющая в "ASCII" номер 48. Это самый первый знак в таблице, называемый "нуль", имеющий номер 0 и двоичное значение 00000000.

Теперь о неочевидном. В нашем примере переменная `a` получила значение 38012. Иначе говоря, она не содержит само строковое значение — а указывает на его начало.

Использование указаний на значения в памяти — ключ к пониманию программирования на Си, и к этому вопросу я ещё не раз буду возвращаться. Сейчас вам достаточно взять на заметку, что в программе на Си при передаче куда-либо строкового значения передаётся не само значение, а лишь указатель на него или его начало (в виде адреса ячейки памяти — прим. перев.). Даже в случае вывода на экран какого-нибудь сообщения без использования переменной

```
printf("некоторое сообщение");
```

на вход функции `printf()` подаётся только указатель на область оперативной памяти, содержащую "некоторое сообщение".

Массивы

Мы скоро убедимся, что использование указания на строковое значение вместо самого строкового значения даёт важные преимущества. Но сперва я бы хотел сказать о явном недостатке.

В Бейсике я могу написать:

```
a$ = "вижу надпись"
```

А затем, позже в коде -

```
a$ = a$ + "приписка"
```

В Си попытка подобного действия в 99 случаях из ста закончится полным провалом. На нашем примере: мы хотим записать "п" в ячейку 38024, "р" в 38025, "и" в 38026 и так далее. Если делать это хоть немного позже самого первого распределения памяти под первое строковое значение — не сомневайтесь, что эти ячейки (38025, 38026 и дальнейшие) уже окажутся заняты другими данными. Си без лишних вопросов эти данные перезапишет, и скорее всего это выльется в бесславный вылет программы.

Объявление массивов

Нужен другой подход: создать массив, который будет хранить строковое значение. Сделать из переменной массив можно во время описания её вида данных. Например:

```
char a[30];
```

Такой массив будет иметь объём 30 знаков ввода (30 байт). А этот

```
int m[150];
```

— соответственно, 150 ячеек для целых чисел.

Всё это напоминает действие `DIM` из Бейсика, но есть отличия. Для обращений к номерам

1 Конечно, в "ASCII" русских букв нет. В Юникоде же русские буквы занимают не по одному, а по два байта, если говорить о кодировке "UTF-8". (прим. перев.)

ячеек массива в Си используются квадратные скобки [ ] . Помимо того, номера ячеек массива начинаются с ноля; в наших примерах массив a содержит ячейки 0-29, массив m — 0-149. То есть при объявлении массива число в квадратных скобках задаёт количество ячеек; и в отличие от Бейсика, номер последней ячейки массива не равен этому числу, а всегда меньше его на 1.

Да. Но вообще-то мы всё это затеяли, чтобы записать в массив строковое значение. Предположим, что у нас на руках есть функция strk_vmsv(), которая занимается такой записью (устройство этой функции ещё разберём). Поэтому для записи строки "здесь что-то написано" в массив a я пишу:

```
strk_vmsv(a, "здесь что-то написано");
```

И здесь скрывается важный вывод. Если я сделаю то же самое в два действия

```
b = "здесь что-то написано";
strk_vmsv(a, b);
```

, то станет видно, что a и b имеют схожую природу. Они указывают на свои значения в памяти! Выходит, что имя массива служит скрытым указателем на первую ячейку этого массива в оперативной памяти.

Так что такое указатели? Это разновидность переменных. От обычных переменных они отличаются тем, что хранят адреса ячеек памяти, а не обращаются к значениям в ячейках (но также могут и обращаться к значениям). Обычное обращение к имени указателя даёт адрес, который в нём хранится. Поэтому с именем указателя (то есть, по сути, с хранимым адресом) возможны арифметические действия.

(Ещё одно отличие переменной от указателя начинается при работе функций, использующих эту переменную или указатель. При чтении данных из переменной функция создаёт временный дубль этих данных, и работает с этим дублем; в случае же с указателем дублей не создаётся, а работа производится со значением в исходной ячейке памяти.

Имя массива — это почти чистый указатель. Отличие лишь в том, что оно указывает на первую ячейку данного массива (т.е. массив указывает сам на себя), а чистый указатель — это внешняя переменная, способная указывать на любой адрес. Функции не создают для массивов дублей. — прим. перев.)

Функции для обработки строковых значений

В качестве необходимых наглядных примеров напишем функции, подобные функциям LEN, LEFT$, RIGHT$ и MID$ из Бейсика[1].

Первой будет функция, которая при подаче в неё имени массива, содержащего строковое значение, вернёт длину этого значения в байтах (что является подобием LEN).

```
int dlina(s)
char s[];
{
    int a;
```

1 Функция LEFT$ в Бейсике усекает строковое значение до указанного количества знаков от начала (то бишь с левой стороны строки). Функция RIGHT$, соответственно, делает то же самое, но от конца строки справа налево. MID$ читает слева направо указанное количество знаков с указанного положения в строке. (прим. перев.)

```
    a = 0;
    while (s[a++])
        ;
    return (a - 1);
}
```

Как обычно, поясню. Обратите внимание, что при написании вида данных массива `s` мы не обязаны указывать, сколько в массиве ячеек[1]. И это хорошо, ведь при пользовании этой функцией мы не будем знать, насколько велики подаваемые ей на вход строки.

Во многих других языках программирования нам бы пришлось заранее задать массиву `s` объём, заведомо превышающий любое значение, которое предположительно будет подано на вход. А это неоправданное расходование памяти. Всё что нам нужно сделать в Си — это оставить квадратные скобки пустыми. Кстати, можете ли вы догадаться, каким образом для массива `s` в этом случае будет распределено место в памяти, если его объём неизвестен?

В теле функции производится поиск знака-разделителя, имеющего значение 0; все проходы подсчитываются. Значения счётчика и номера очередной ячейки массива совпадают, поэтому для них использована одна и та же переменная. Напоминаю, что запись

```
while (s[a++])
```

означает «до тех пор, пока выражение `s[a++]` принимает значение "истина"», и полностью равносильна записи

```
while (s[a++] != 0)
```

С какой основной целью мы ввели цикл? Чтобы считать проходы. Но эта задача уже выполняется в заголовке цикла, поэтому его тело оставлено пустым. Обратите внимание на закрывающую точку с запятой.

Возможно, не совсем ясно почему в вызвавшую функцию возвращается значение `a - 1`. Когда искомый знак-разделитель будет найден, выполнение цикла будет завершено; однако сразу после проверки условия значение `a` увеличится на единицу. Поэтому мы уменьшаем его обратно.

Отрезок строки слева

По некоторым причинам, о которых будет сказано позже, не лучшим решением будет воссоздать функцию `LEFT$` из Бейсика один в один:

```
b$ = LEFT$(a$, 6)
```

Более предпочтительно сделать исходную строку и массив, куда будет записано получаемое строковое значение, аргументами функции. То есть в Си выглядеть это может так:

```
sleva(a, b, 6);
```

Здесь `a` является именем массива, а значит и скрытым указателем на исходную строку. После выполнения функции усечённая строка будет записана в массив `b`. То есть `b` станет указывать на выделенные из исходной строки 6 первых знаков.

Вот пример решения этой задачи.

---

1 Это касается только массивов "знаков ввода". При других видах данных размерность массива обязательно требуется указывать. (прим. перев.)

```
int sleva(ish_stroka, usech, n)
char ish_stroka[], usech[];
int n;
{
    int scht;
    for (scht = 0; scht < n; scht++)
         usech[scht] = ish_stroka[scht];
    usech[scht] = 0;
    return 0;
}
```

Ничего незнакомого тут нет. Из исходной строки в "усечённую" дублируется `n` байт, а после цикла добавляется знак "нуль", необходимый в кодировке "ASCII". Заметьте, что по сути здесь не обязательно действие `return`, поскольку мы не задумывали никакого возврата значения из функции.[1]

Задача 1

Функция вышла несовершенной. Если `n` превышает количество знаков исходной строки, то в `usech` после окончания собственно строки начнут записываться данные из последующих ячеек оперативной памяти. Это нам вряд ли нужно. Однако особого вреда это не принесёт, поскольку в этом случае вместе с исходной строкой будет считан и её завершающий "нуль". Поэтому при выводе полученной строки через `printf()` ошибок не случится.

Тем не менее, попробуйте переписать функцию так, чтобы в случае если `n` превышает длину исходной строки, функция возвращала бы значение "ложь", иначе — значение "истина".

После этого исправления можно будет создать сообщение об ошибке:

```
if (!sleva(a, b, 7))
    printf("Заданная длина превышает длину исходной строки");
```

Отрезок строки справа

Логично предполагать, что функция для усекания исходной строки от её конца должна быть похожей.

```
int sprava(ish_stroka, usech, n)
char ish_stroka[], usech[];
int n;
{
    int scht, scht2 = 0;
    for (scht = dlina(ish_stroka) - n; scht <= dlina(ish_
stroka); scht++)
         usech[scht2++] = ish_stroka[scht];
    usech[scht2] = 0;
    return 0;
}
```

Функция `dlina()` использована для нахождения точки, с которой следует начать чтение слева направо. Собственно, другие пояснения вряд ли нужны.

---

1 Но без него компилятор будет писать предупреждение. (прим. перев.)

Задача 2

Напишите функцию `izvlech()`, которая так же извлекает часть исходной строки, но умеет делать это с определённого места. Таким образом, этой функции требуются два аргумента-числа. Первое задаёт начальное положение считывания, второе — количество считываемых знаков.

```
strk_vmsv(ish_stroka, "собака");
izvlech(ish_stroka, usech, 4, 2);
printf(usech);
```

В этом примере на экран должно быть выведено

```
ак
```

Конечно, всё это соответствует функции Бейсика `MID$`. Чьё название всегда казалось мне весьма неточным, поскольку оно намекает на середину строки, хотя функция может считывать строку из любого места.

Кстати, если бы мы написали `izvlech()` самой первой, то могли бы использовать её в `sleva()` и `sprava()`, что существенно упростило бы их создание.

Нужно больше указателей

Мало кто из программистов, предпочитающих Си, написал бы все эти функции подобно вышеприведённым примерам. Я уже упоминал, что за именем массива скрывается указатель; но также упоминал, что указатели в Си бывают и в явном виде. Чтобы создать явный указатель, необходимо при объявлении вида данных переменной приписать к ней спереди звёздочку * .

```
char *a;
```

В этом примере была объявлена переменная `a`, которая стала явным указателем на значение вида "знак ввода" (1 байт).

При выполнении действий со значением в ячейке памяти знак звёздочки используется в сочетании с именем переменной, как показано ниже. (Знак звёздочки * означает "взять значение по адресу". При этом, очевидно, адрес у указателя должен уже обязательно иметься. Запись `*a` означает "взять значение по адресу, хранящемуся в указателе `a`". А вот обращение к имени `a` теперь даст значение адреса,который хранится в указателе. — прим. перев.)

```
*a = 0;
```

Этот пример означает "записать 0 в ячейку по адресу, хранимому указателем `a`". (В соответствии с объёмом вида данных указателя.)

Перепишем с учётом вышенаписанного функцию `dlina()`.

```
int dlina(s)
char *s;
{
    char *nachalo;
    nachalo = s;
    while (*s++)
        ;
    return (s - nachalo - 1);
}
```

Порядок действий остался таким же. Для перемещения по знакам строки мы приращаем значение адреса. А в указателе `nachalo` мы сохранили его исходное значение.

Снова "sleva()"

Перепишем функцию `sleva()` с применением указателей:

```
int sleva(ish_stroka, usech, n)
char *ish_stroka, *usech;
int n;
{
    while (n--)
         *usech++ = *ish_stroka++;
    *usech = 0;
    return 0;
}
```

Недурно, согласитесь. Цикл `while` показался мне более уместным. Значение `n` последовательно уменьшается до ноля, таким образом цикл выполняется `n` раз. Внутри цикла значение по адресу `ish_stroka` записывается в ячейку по адресу `usech`. После чего значения адресов приращаются. В конце, как и в предыдущем образце функции, в конечный байт усечённой строки записывается знак "нуль".

Ещё раз "sprava()"

Таким же образом переработаем `sprava()`:

```
int sprava(ish_stroka, usech, n)
char *ish_stroka, *usech;
int n;
{
    ish_stroka = ish_stroka + dlina(ish_stroka) - n;
    while (*ish_stroka)
         *usech++ = *ish_stroka++;
    *usech = 0;
    return 0;
}
```

Я снова изменил вид цикла, и добавил запись завершающего "нуля". Изучите внимательно этот пример, чтобы вам было понятно всё до последнего знака.

Объединение строк в массиве

В начале этой главы мы немного говорили о функции `strk_vmsv()`, записывающей строковое значение в массив. Помимо неё нужна ещё одна, которая бы объединяла две строки: чтобы вторая дозаписывалась в массив к первой.

Напишем сначала `strk_vmsv()`.

```
int strk_vmsv(massiv, ish_stroka)
char *massiv, *ish_stroka;
{
    while (*ish_stroka)
         *massiv++ = *ish_stroka++;
    *massiv = 0;
    return 0;
}
```

Если вы внимательно изучили предыдущие примеры с указателями, то всё это уже должно быть вам хорошо знакомо.

Составим и вторую функцию:

```
int strk_edn(osn_stroka, dobav)
char *osn_stroka, *dobav;
{
    osn_stroka = osn_stroka + dlina(osn_stroka);
    while (*dobav)
         *osn_stroka++ = *dobav++;
    *osn_stroka = 0;
    return 0;
}
```

Адрес указателя `osn_stroka` увеличивается так, чтобы он указывал на конец основной строки. Затем в массив дублируется добавочная строка. Последнее можно выполнять с помощью только что написанной функции `strk_vmsv()`:

```
int strk_edn(osn_stroka, dobav)
char *osn_stroka, *dobav;
{
    osn_stroka = osn_stroka + dlina(osn_stroka);
    strk_vmsv(osn_stroka, dobav);
    return 0;
}
```

Конечно, отдельная задача для программиста — принять меры, чтобы в массиве было достаточно места для необходимого увеличения строки.

В данной главе было довольно много арифметики внутри функций. Хочу сообщить вам, что при работе с обыкновенными переменными функция создаёт их временные дубли, никак не затрагивая исходные значения; поэтому после выхода из функции исходные значения поданных в неё переменных сохранятся.

Пришло время разоблачений. Многие из созданных нами в этой главе функций являются

велосипедами. В библиотеке Си присутствуют следующие функции, выполняющие рассмотренные нами задачи:

`strlen()` — считает длину строкового значения

`strcpy()` — записывает строковое значение в массив

`strcat()` — объединяет строковые значения.

Надеюсь, это признание не повергло вас в уныние. Зато мы неплохо поупражнялись.

Ещё немного разоблачений. Присвоение переменной строкового значения, показанное в начале главы, может не поддерживаться некоторыми компиляторами. Безопаснее и надёжнее записывать строки в массивы. Но на том этапе я ещё не успел рассказать о функции, записывающей строку в массив.

В главе 6 вы познакомитесь с ещё одним способом присвоения значений переменным.

Ответы

Задача 1

Возможное решение:

```
int sleva(ish_stroka, usech, n)
char ish_stroka[], usech[];
int n;
{
    int scht, neprevish;
    neprevish = 1;
    for (scht = 0; scht < n; scht++ )
    {
         usech[scht] = ish_stroka[scht];
         if (!usech[scht])
              neprevish = 0;
    }
    usech[scht] = 0;
    return neprevish;
}
```

Будем исходить из предположения, что `n` не превышает длину исходной строки, поэтому ставим значение `neprevish` равным 1. Если же во время записи усечённой строки встретится знак "нуль", то `neprevish` получает значение 0. Помните, что здесь мы пользуемся массивами. На время появления задачи 1 в главе до использования указателей мы ещё не дошли.

Задача 2

На правах сдвоенного автора книги поступлю нечестно и раньше времени использую указатели.

```
int izvlech(ish_stroka, usech, otkuda, skolko)
char *ish_stroka, *usech;
int otkuda, skolko;
{
    ish_stroka = ish_stroka + otkuda - 1;
    while (skolko--)
        *usech++ = *ish_stroka++;
    *usech = 0;
    return 0;
}
```

Всё должно быть понятно. Адрес `ish_stroka` устанавливается в нужное положение в строке (заметьте, необходима поправка `-1`). Затем переменная `skolko` используется в качестве счётчика необходимого количества читаемых знаков.

Ранее я упоминал, что функцию `izvlech()` можно использовать в `sleva()` и `sprava()`:

```
int sleva(ish_stroka, usech, n)
char *ish_stroka, *usech;
int n;
{
    izvlech(ish_stroka, usech, 1, n);
    return 0;
}

int sprava(ish_stroka, usech, n)
char *ish_stroka, *usech;
int n;
{
    izvlech(ish_stroka, usech, dlina (ish_stroka) - n + 1, n);
    return 0;
}
```

Также звучал вопрос: "как Си распределяет место для массивов, передаваемых в функцию, если не указывать их объём?". Ответ: никак. Благодаря тому, что имя массива является указателем, функция получает доступ к ячейкам этого массива, и просто читает из них строковое значение с завершающим знаком "нуль".[1]

1 Повторю, что это касается только случаев, когда массив содержит строковое значение, и причина — именно в наличии знака "нуль", который позволяет определить, где заканчивается массив. В остальных случаях функции необходимо сообщить объём массива. (прим. перев.)

В решении задачи 2 поправка `-1` существует только потому, что я стараюсь повторить функции обработки строк из Бейсика. В них первый знак строки после записи в массив получает номер ячейки 1, вместо номера 0. И это странно, поскольку в любых других массивах Бейсик понимает обращения к ячейке 0. Вы можете переработать функцию так, чтобы избавиться от этой поправки.

ЗАДАНИЯ

1. Напишите функцию `dennedeli()`, которая принимает указатель `*data`, указывающий на строковое значение с датой в виде "март 18 1937". Возвращать функция должна указатель `*den`, указывающий на строку с названием соответствующего дня недели.

2. Напишите вечный календарь. Он должен принимать в качестве входных данных месяц и год, и выдавать распределение чисел по дням недели в привычном календарном виде:

```
Август  1986
ПН  ВТ   СР   ЧТ   ПТ   СБ   ВС
                  1    2    3
4   5    6    7    8    9    10
11  12   13   14   15   16   17
18  19   20   21   22   23   24
25  26   27   28   29   30   31
```

ГЛАВА 6

Число с плавающей точкой и другие виды данных

До сих пор мы пользовались только видами данных "целое число" и "знак ввода". Существует несколько других видов данных, однако не каждый компилятор поддерживает все из них. Рассмотрим, что же это за виды.

float (число с плавающей точкой)

Это слово знакомо каждому, кто писал на Бейсике. Этот числовой вид данных (объём в памяти 4 байта — прим. перев.) предназначен для записи дробных чисел.

В виде числа с точкой можно записать как огромные, так и крошечные величины. Область возможных значений и точность их представления зависят от используемого компилятора. Чаще всего существенны 6 первых цифр в значении (включая целую часть)[1], а наибольшее возможное значение — 10^38.

double

Это то же число с плавающей точкой, но его объём в памяти вдвое больше (8 байт; точность — первые 15 цифр, начиная от значимой.

Данный вид предпочтителен к использованию вместо `float`. Сегодня компиляторы, встретив любое число с точкой, назначают ему вид `double`. — прим. перев.).

Разновидности "целого числа"

Целое число (`int`) имеет ещё три подвида — малой величины (`short`), большой величины (`long`) и беззнаковое (`unsigned`).

То бишь уточняющие слова `short` и `long` управляют количеством ячеек памяти, которые выделяются для переменной. К примеру, компилятор, назначающий по умолчанию (без уточняющих слов) для "целого числа" 16 бит, при слове `short` назначит 8 бит, при слове `long` — 32.

Однако не существует никаких жёстких правил, гласящих, что всё всегда будет так. Разработчики компиляторов вольны изменять это поведение на своё усмотрение — для возможности написать наиболее подходящий компилятор под целевой компьютер, что выгодно нам с вами. И если на каком-то компиляторе слово `short` не оказывает никакого воздействия — переменная воспринимается как обычное "целое число", без уточнений.

(При этом вообще исключать слово `short` — плохой выбор, который может привести к непереносимости кода. Проще и безопаснее по-другому это слово воспринимать. К тому же перенос значения из "малой величины" в обычное "целое число" не приводит ни к потере, ни к искажению данных, поскольку объём "целого числа" в памяти — больше либо равен "малой величине".)

Теперь по поводу "беззнакового" подвида. Если объём "целого числа" в памяти равен 16 битам, то допустимые хранимые значения составляют от -32768 до 32767 (всего 65536 значений). Но если исключить возможность наличия отрицательных значений, то значения могли бы составлять от 0 до 65535. Именно это и позволяет сделать слово `unsigned`. Значение пере-

---

1 Если уточнить — то первые 6 цифр, начиная от значимой цифры (отличной от ноля). Например, если мы имеем значение 0.0000456... , то отсчитывать следует от цифры 4. (прим. перев.)

менной будет только положительным, и это позволит использовать вдвое большее по модулю предельно возможное число.

Если вы используете эти уточняющие слова при объявлении вида данных "целое число", то слово `int` можно не писать. Например:

```
unsigned polozhitelnoe;
short maloe;
long nemaloe;
```

Сдвиг битов и знак числа

Я ранее дал понять, что сдвиг битов изменяет значение по-разному, в зависимости от случая. Например, в случае деления обычного "целого числа" на 2 (сдвига битов вправо) наличие или отсутствие знака неважно. Во время сдвига вправо значение старшего бита записывается в "освободившийся" бит, и знак сохраняется[1]. Если же сдвигать вправо "беззнаковое" значение, то освободившиеся биты всегда будут заполняться нолями. Для понимания это несложно, происходящее вполне отражает наличие или отсутствие знака у значения.

Но что будет, если выполнить сдвиг вправо для вида данных "знак ввода"? Многие компиляторы считают, что этот вид данных имеет знак (положительное или отрицательное значение), тогда как другие воспринимают его как беззнаковый. В первом случае сдвиг битов вправо работает точно так же как с "целым числом", во втором — как с "беззнаковым" значением.

Это ещё не вся неразбериха. Если со "знаком ввода" выполнять арифметические действия, то он временно преобразуется в "целое число". И если данный компилятор воспринимает "знаки ввода" как значения со знаком (положительности \ отрицательности), то самый старший значимый бит "знака ввода" будет размножен до упора влево. Например, вместо исходного значения 00001010 вы получите 11111010.

В таких компиляторах отдельным видом данных выделен "беззнаковый знак ввода"[2], `unsigned char`. В главе 14 будут примеры, где объявить именно этот вид данных будет важно. В остальном в книге подразумевается обычный `char`, без уточнений. Но если в вашем компиляторе присутствует такой вид, как `unsigned char` — вам нужно пользоваться именно им.

Переменные в регистрах

И ещё одно уточняющее слово для "целого числа" и "знака ввода".

```
register int a;
register char b;
```

Собственно виды данных `a` и `b` никак не затрагиваются и остаются прежними — обыкновенное "целое" и обыкновенный "знак ввода". Слово `register` указывает компилятору, что значения `a` и `b` следует по возможности сохранить в регистрах центрального процессора. Если эти значения очень часто используются, то обращение к этим значениям будет происходить несколько быстрее, чем обращение за ними же в основную оперативную память; в свою очередь, это несколько ускорит выполнение программы.

1 Такой случай, когда при сдвиге вправо бит, отвечающий за знак, сохраняется, называют арифметическим сдвигом. В противном случае сдвиг называют логическим. В "ассемблере" арифметический и логический сдвиг — это два разных действия; в Си же то, какой именно сдвиг будет выполнен, зависит от наличия знака у обрабатываемого значения. Некоторые компиляторы выполняют только логический сдвиг вправо. (прим. перев.)

2 В этом месте не осилил. (прим. перев.)

Это одна из способностей Си по выжиманию всей возможной скорости из "железа". Но не для всякого "железа" этот приём возможен. Программисту следует чётко представлять себе возможности целевого процессора, в частности — количество регистров общего назначения, в которые и могут быть записаны данные создаваемой программы. Если данные в процессоре поместить некуда — они будут вынужденно сохранены в основную оперативную память, из чего следует, что никаких преимуществ в скорости работы не получится.

Объявление структур

Ряд переменных можно объединить в структуру (именованный ряд переменных). При этом переменные могут иметь разные виды данных.

К примеру предположим, что требуется создать базу данных для товаров магазина. Пусть каждая запись содержит код товара, описание, цену, и количество единиц этого товара на складе.

Напишем следующее:

```
struct zapis
{
    int kod_tvr;
    char opisanie [30];
    double tsena;
    int sklad_kolvo;
};
```

Таким образом мы создали именованный ряд значений, называемый структурой. Обратите внимание на точку с запятой после закрывающей фигурной скобки — это не ошибка. Всего лишь в двух случаях в Си эта точка с запятой должна ставиться, и объявление структур — один из этих случаев.

Мы определили, как должна выглядеть одна запись. Теперь создадим нашу базу данных:

```
struct zapis bazadannih [200];
```

Она, таким образом, представляет собой массив из 200 ячеек. Каждая из ячеек представляет собой структуру `zapis`.

И теперь структура `zapis` стала ещё одним видом данных, можно сказать составным. Это значит, что я могу создать указатель на структуры вида `zapis`:

```
struct zapis *u;
```

При помощи этого указателя я получаю возможность указывать на любое место в созданной "базе данных" (массив `bazadannih`). Рассмотрим это позже, в главе 11.

Пользовательские имена видов данных

Видам данных можно задавать любые желаемые имена, и обозначать их этими именами (при этом исходные обозначения видов данных, например `int`, `char` и др. продолжат работать). Предположим, что в программе вам потребовались даты. Месяцы будут представлять собой числа от 1 до 12 — очевидно, нужен вид данных "целое число". Однако согласитесь — было бы чуть проще иметь вид данных "целое число", но с названием "месяц". И это легко устроить.

```
typedef int mesyats;
```

После этого обозначения `int` и `mesyats` становятся равнозначными. И можно объявлять переменные:

```
mesyats data1, data2;
```

Так как структуры тоже становятся своеобразными отдельными видами данных, то дадим нашей структуре `zapis` для удобства новое обозначение:

```
typedef struct zapis vid_zapis;
```

Массивы из этих структур затем создаются так:

```
vid_zapis bazadannih[5000], prosto_massiv[2000];
```

Как видите, всё это существует исключительно для удобства и обеспечения читаемости — на уровне, невообразимом для Бейсика. Понятность написанного — это сбережённое время.

Числовые значения (постоянные, константы)

В присвоении числовых значений долго обсуждать нечего. Наиболее частые присвоения выглядят один в один как в Бейсике:

```
int x;
x = 1;
```

в случае с целым числом, и

```
double pi;
pi = 3.142;
```

для дробного числа.

Для больших или малых значений также может быть использована запись в виде экспоненты.

```
double skolko_navorovali;
skolko_navorovali = 1.3e9;
```

В данном примере переменная получила значение 1 300 000 000.

Значения вида "знак ввода"

Можно было бы подумать, что один знак ввода записывается в памяти как строка длиной в 1 знак. Это не так, поскольку для записи такой строки требуется объём не один, а два байта — второй нужен для знака-разделителя "нуль". Для записи значения в 1 знак ввода требуется давать значение только одному байту. Это делается так:

```
char a;
a = 'A';
```

Таким образом, одинарные кавычки используются для записи одного отдельного знака ввода. В двойных же кавычках должно записываться строковое значение.

Работа с условными сочетаниями

Есть знаки, которые выполняют какую-то задачу, но чаще всего не отвечают ни за какой привычно видимый знак ввода. Этих условных знаков на клавиатуре нет. А как же тогда использовать их в коде?

Для этого существует два подхода. Первый — действовать подобно функции `CHR$` в Бейсике (принимает код знака, возвращает сам знак — прим. перев.).

```
char dindin;
dindin = 7;
```

Переменной "диньдинь" присвоен код знака 7. Соответствующий знак в "ASCII" при выводе, как правило, заставляет компьютер издать звук наподобие гудка. (Подробнее о выводе — в главе 12.)

Второй подход. Для ввода многих из этих служебных знаков в Си предусмотрены сочетания, которые начинаются с косой черты \ . После неё следует или одна буква, или 3-х (либо менее) значное восьмиричное число — код знака. Это сочетание Си преобразует в один знак. Вот набор основных условных сочетаний:

`\n`	новая строка
`\t`	горизонтальная табуляция
`\b`	от слова "backspace", затирка предыдущего от указателя знака
`\r`	возврат каретки (возврат указателя в начало строки)
`\\`	используется для ввода косой черты \
`\'`	используется для ввода одиночной кавычки
`\xxx`	ввод восьмиричного кода знака, где `xxx` — цифры
`\0`	"нуль"

Например, напишем следующее:

```
printf("Разбито \n на три \n строчки");
```

На экран будет выведено:

```
Разбито
 на три
 строчки
```

Если далее выполнится ещё одна функция `printf()`, то её вывод появится сразу после слова "строчки", впритык. Это произойдёт потому, что после слова "строчки" в нашем примере нет разделителя строк `\n`. Ещё пример:

```
printf("Отогнать кота от принтера!\7\7\7");
```

Сообщение будет выведено на экран, и трижды будет подан звук. Подчёркиваю, что 7 в данном примере — это восьмиричное число.

То, что знак "нуль" также представлен среди условных сочетаний, позволяет вам отыскивать конец строкового значения:

```
while(*a != '\0')
{
    ...
}
```

Т.к. двоичные значения знака ввода "нуль" и ноля (целого числа) совпадают, и одновременно означают "ложь", я скорее предпочёл бы записать этот пример короче:

```
while(*a)
{
    ...
}
```

Но может показаться, что это слишком трудно читать. В любом случае вам следует быть знакомыми с такой записью — не исключено, что она встретится вам в чужом коде.

Другие условные обозначения в коде...

Одна из особенностей Си в том, что в его компиляторах содержится предобработчик кода (препроцессор). Ещё до этапа компиляции исходный код может быть (и будет) через него пропущен.

Все указания для предобработчика кода обозначаются знаком решётки # . Рассмотрим одно из таких указаний.

#define

Оно заставляет предобработчик повсеместно заменить в коде одно заданное слово другим заданным вхождением. (Под "словом" здесь понимается любое сочетание знаков в коде.) Пример:

```
#define  NUL  '\0'
```

(Здесь следует крайне внимательно относиться к пробелу между заменяемым и заменителем. В заменяемом никаких пробелов быть не должно, иначе самый первый из них будет воспринят как разделитель заменяемого и заменителя.

Всё это обусловлено тем, что предобработчик является программой текстовой обработки, и потому для него, в отличие от компилятора, наличие \ отсутствие пробельных знаков имеет значение. — прим. перев.)

Здесь предобработчику задано заменить каждое вхождение `NUL` вхождением, содержащим условное сочетание для знака "нуль". Скажем, в коде программы написано

```
while(*a != NUL)
{
    ...
}
```

Благодаря предобработчику перед компиляцией этот код будет приведён в рабочий вид. С такими словесными обозначениями код более читаем.

Заметьте, что заменяемое обозначение записано заглавными буквами; это вовсе не обя-

зательно, просто так принято. Благодаря этому обозначения для замены указанием `#define` выделяются, их сразу можно увидеть.

Указание `#define` может быть написано в любом месте в коде; но как правило, его вводят в самом начале. Разумеется, оно обработает только тот код, который следует после него.

...и их применение

Весьма часто в коде используются массивы, объём которых требуется менять от случая к случаю. Это, например, может быть дисковый буфер (он же кэш), объём которого часто отличается даже у носителей одного вида, но разной вместимости.

В общем случае для смены объёма массива придётся руками заменять каждое вхождение значения этого объёма в коде. Причём если это значение (допустим, 512) совпадает с другим значением (тоже 512), которое относится уже к чему-то другому — то во время этой замены следует быть ещё и предельно внимательным, дабы внести изменения только туда куда нужно.

Предобработчик позволяет действовать предусмотрительнее:

```
#define BUFER 512
```

Это позволит объявить массив в коде так:

```
char disk_bufer[BUFER];
```

И где-нибудь ещё в дальнейшем коде напишем

```
for (p = 0; p < BUFER; p++)
{
    ...
}
```

, в общем, думаю, мысль понятна. Долгая и нудная ручная замена по всему коду превращается в правку одной \ нескольких строчек с указанием `#define` в самом начале. Чего ещё желать?

#include

Если указаний `#define` много, то удобно было бы собирать их в отдельном файле, содержимое которого бы включалось в код перед компиляцией. Для включения в код содержимого другого файла служит указание предобработчика `#include`.

Допустим, у вас имеется файл `config.dat`, в котором заданы величины, касающиеся вашего компьютера (объём оперативной памяти, дискового кэша, разрешение экрана и прочее). Для добавления его содержимого в целевом файле исходного кода (в самом начале) требуется написать:

```
#include <config.dat>
```

Иногда требуются заключать название файла в двойные кавычки, а не в угловые скобки:

```
#include "config.dat"
```

Строго говоря, угловые скобки и кавычки используются в разных случаях — в зависимости

от расположения целевого файла[1]. Но всё это может меняться по воле разработчиков компилятора. Ознакомьтесь с руководством того компилятора, который используете.

Многие ваши программы будут начинаться так:

```
#include <stdio.h>
```

В файле `stdio.h` содержатся некоторые наиболее часто используемые замены `#define`[2] наподобие

```
#define  NULL '\0'
#define  EOF  -1
```

Это позволяет использовать их в коде сразу, без необходимости писать соответствующие правила `#define` самому.

Восьмиричные и 16тиричные значения

Бывают случаи, когда цифровое значение не обозначает ни число, ни знак ввода. Пример — двоичные "маски" из главы 4. Очевидно, что именно двоичное значение этих "масок" является целевым, искомым для нас. При работе с этими значениями предпочтительно, чтобы преобразование в двоичное значение выполнялось проще — как с точки зрения человека, так и алгоритма преобразования.

Язык Си позволяет записывать значения в восьмиричном и 16тиричном виде. В первом случае каждая цифра определяет по три бита, во втором — по четыре.

Соответствия 8- и 16-ричных значений двоичным

8ричное	двоичное	16тиричное	двоичное
0	000	0	0000
1	001	1	0001
2	010	2	0010
3	011	3	0011
4	100	4	0100
5	101	5	0101
6	110	6	0110
7	111	7	0111
		8	1000
		9	1001
		A	1010
		B	1011
		C	1100
		D	1101
		E	1110
		F	1111

1 Как правило, угловые скобки <> ставятся для всех файлов, содержащих описание стандартных библиотечных функций. Двойные кавычки "" используются для, если позволите, пользовательских файлов, созданных программистом. (прим. перев.)

2 И не только, ещё этот файл описывает библиотечные функции ввода \ вывода. Подробнее о содержимом файла `stdio.h` можно почитать в Сети. (прим. перев.)

К примеру, значение 1000 0001 можно представить как

```
10  000  001
2   0    1     восьмиричный вид
```

, или как

```
1000    0001
8       1     16тиричный вид
```

В обоих случаях разбор двоичного значения производится справа налево.

Не к месту будет обсуждать, как именно производятся эти преобразования. Если вы хотите подробнее познакомиться с двоичным, восьмиричным и 16тиричным представлениями — про них написаны целые библиотеки книг. Часто эти книги повествуют о машинном коде того или иного процессора, поэтому можете выбрать ту, которая рассказывает о процессоре, имеющемся у вас.

В Си необходимо отдельно обозначать запись в восьмиричном и 16тиричном виде. Если первая цифра — ноль, то значение воспринимается как восьмиричное. Если за первым нолём следует `x` (икс — прим. перев.) — то как 16тиричное.

```
mask = 129;
mask = 0201;
mask = 0x81;
```

Эти три значения равнозначны. "Икс" в 16тиричной записи, равно как и буквы от "A" до "F", может быть как заглавным, так и строчным.

Объявление переменной сразу со значением

В Си значение переменной можно присваивать сразу при её объявлении. То есть можно писать

```
int t = 7;
```

вместо

```
int t;
t = 7;
```

Присвоение значения переменной или массиву во время её \ его объявления принято называть инициализацией.

```
int massiv [3] = {0, 7, 12};
```

То же самое, но последовательно:

```
int massiv[3];
massiv[0] = 0;
massiv[1] = 7;
massiv[2] = 12;
```

Понятно, что это только возможность — необходимости так делать нет. К тому же, ряд компиляторов эту возможность не поддерживает. Поэтому всегда помните о переносимости кода:

если вы хотите, чтобы ваш код работал на как можно более разных компьютерах и процессорах — оставляйте в нём только те приёмы, которые поддерживаются всеми компиляторами.

И кстати. Объявление массива сразу со значением — это второй и последний случай в Си, когда после закрывающей фигурной скобки следует точка с запятой.

Присвоение значений указателям

Когда мы в главе 5 рассматривали указатели, то мы присваивали им значения (адреса ячеек памяти) неявно.

```
int a[50], *u;

u = a;
```

В этом примере `u` — указатель. Адрес `a` записывается в адрес `u` (так как `a` — имя массива, и потому является разновидностью указателя, а значит обращение к `a` даёт её адрес). Таким образом, указатель `u` стал указывать на первую ячейку массива `a`.

Но у нас есть и прямая возможность получить адрес переменной. Для этого перед ней следует записать знак & .

```
int bdi, *u;

u = &bdi;
```

Теперь `u` указывает на ячейку(-и) переменной `bdi`. Теперь мы можем вывести адрес `bdi` — требуется только вывести значение `u`. Таким образом, знак & означает "извлечь адрес".

Получается, что если нам нужен адрес первой ячейки массива `a`, то следовало бы писать:

```
&a[0]
```

И конечно, это ничем не отличается от просто

```
a
```

, потому как это имя массива — оно указывает на первую ячейку этого массива. И потому запись

```
int a[50], *u;

u = a;
```

равнозначна записи

```
int a[50], *u;

u = &a[0];
```

И если второй пример не вдохновляет на более длинную запись, то, скажем, следующий

```
u = &a[27];
```

даёт представление о том, зачем она может понадобиться. Например — для поиска по ячейкам массива:

```
int a[50], *u, *v;

u = a;

v = &a[49];

while (*u++ != *v--)

    ...
```

Задача 1

Что происходит в вышеприведённом примере?

Вид данных для функции

Так как функция вычисляет какое-то значение — ожидаемо, что вид данных для этого значения требуется объявить. По умолчанию[1] значение, возвращаемое функцией, воспринимается как "целое число".

В большинстве случаев это именно то, что и требуется. Даже если функция возвращает "знак ввода" — ничего страшного не будет. Как мы знаем, "знак ввода" успешно преобразуется в "целое число" без нашего участия.

Но давайте представим себе функцию `isk_znak()`, которая ищет указанный знак в указанной строке. Она принимает два указателя — на строковое значение и на знак, который надо искать. Получить же мы хотим указатель на найденный знак в строке.

```
int isk_znak(s, z)
char *s, z;
{
    while(*s++ != z)
        ...
    return (s - 1);
}
```

Думаю, в примере всё ясно. Для получения желанного указателя вызовем эту функцию, например, так:

```
int main()
{
    char stroka [30], *isk_znak(), *u;
    ...
    strcpy (stroka, "с таможенной поддержкой водителя мыши в
текущем двигателе ботинка");
    ...
    u = isk_znak(stroka, 'ш');
    ...
```

Здесь важно обратить внимание на строчку с объявлением функции в качестве указателя. (Получается указатель на значение, которое вернёт функция. — прим. перев.) А именно — запомните скобки: они необходимы, поскольку они показывают, что следующее перед ними имя является именем функции, а не чем-либо другим.

---

1 То есть если для функции не указать вид данных и не обращать внимание на предупреждение компилятора. (прим. перев.)

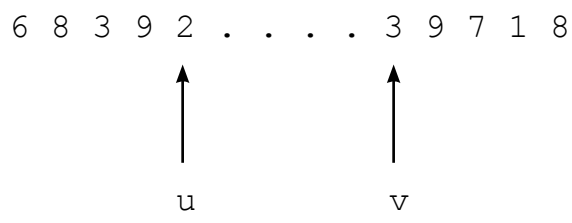
Ответы

Задача 1

Если значения по адресам "u" и "v" не равны, то адрес "u" приращается, адрес "v" приращается отрицательно. Цикл остановится, когда их значения окажутся равны; и после этого выполнится ещё по одному приращению. Например, если массив выглядит так

```
6 8 3 9 2 . . . . 3 9 7 1 8
```

, то остановка произойдёт на девятках, но ещё одно приращение даст ещё по одному сдвигу.



ГЛАВА 7

Ввод

Восполним ещё один пробел. Ранее мы совсем не обсуждали ввод данных в компьютер — и вывод — при помощи Си. Этим вопросам посвящена эта и следующая глава.

Простейший ввод

Чтобы начать не с пустыми руками — воспользуемся имеющейся в Си функцией `getchar()`, которая читает один знак ввода с клавиатуры и возвращает его. Ну то есть — чаще всего это чтение происходит с клавиатуры. В большинстве сред Си устройства для ввода и вывода легко можно поменять. По умолчанию назначены, соответственно, клавиатура и монитор — их и будем подразумевать.

Используя `getchar()`, создадим несколько собственных функций ввода.

Забить буфер до отказа

Самое бесхитростное использование `getchar()` — набрать последовательность знаков в некий буфер, для каких-то дальнейших действий. Создадим для этой задачи функцию `vvod_buf()`.

Сперва надо решить, каков будет объём буфера, и какой знак использовать в качестве разделителя. За объём примем ширину экрана используемого терминала в количестве знаков[1], за разделитель — возврат каретки в кодировке "ASCII"; таким образом, объём буфера будет равен одной строчке на экране.

Не будем забывать, что Си позволяет задать значения, которые вероятно будут изменены в будущем, указанием #define.

```
#define BUF 40
#define RAZDEL 13
```

Это позволит нам записать в функцию принятые буквенные обозначения, и забыть о нудной ручной правке значений в будущем.

Вот первый образец функции:

```
int vvod_buf(bufer)
char *bufer;
{
    char *ukazatel;
    for (ukazatel = bufer; ukazatel < bufer+BUF; ukazatel++)
    {
         *ukazatel = getchar();
         if (*ukazatel == RAZDEL)
              return 0;
    }
```

1 Ещё один привет из конца 80-х. Терминал представляет собой устройство, где объединены экран и клавиатура. Экран условно разбит на определённое количество столбцов и строчек, и один знак ввода занимает ровно одну ячейку. Командная строка ОС "Windows 7" по умолчанию имеет длину строки 80 ячеек (настраиваемо); в самых распространённых дистрибутивах ГНУ\Линукса терминалы, чей экран разбит на ячейки, мне не встречались. (прим. перев.)

```
    return 0;
}
```

В функцию подаётся имя массива, заранее заготовленного для буфера. В пределах функции это имя массива делается явным указателем. Кроме того создаётся ещё один указатель, с помощью которого в ячейки массива по одному записываются вводимые знаки. В случае когда записано ровно `BUF` знаков, либо если прочитан знак с кодом `RAZDEL` — выполняется возврат в вышестоящую функцию.

Важно заметить, что мы не создали условий, чтобы вывести получившееся содержимое буфера на экран:

```
printf(bufer);
```

Потому что в конце буфера мы не предусмотрели знак "нуль", а значит наш буфер не содержит строковое значение.

Вместо цикла `for` многие программисты на Си предпочли бы `while`, и выглядело бы это своеобразно.

```
int vvod_buf(bufer)
char *bufer;
{
    int schetchik;
    schetchik = 0;
    while ((*bufer++ = getchar()) != RAZDEL && schetchik <
BUF)
         schetchik++;
    *bufer= '\0';
    return 0;
}
```

Счётчик использован для подсчёта количества введённых знаков.

Цикл `while` и то, что у него написано в условии, разберём по порядку. Каждый раз, читая вводимый знак, мы проверяем, является ли он разделителем. Ожидаемо было бы видеть такую запись:

```
while (getchar() != RAZDEL)
```

Но в этом случае функция `getchar()` выполняется понапрасну — прочитанный введённый знак никуда не записывается. Хотелось бы, проверив на соответствие разделителю, заодно его и сохранить.

И эта задача выполнима. Мы уже встречали примеры "почти одновременных" действий — среди них приращение во время действия присваивания. Так вот, внутри сравнительного выражения можно выполнять присваивание. Исходя из этого, я пишу:

```
while (a = getchar() != RAZDEL)
```

С точки зрения записи — никаких ошибок, но сработает всё это не так, как нам надо. Первым делом будет выполнено сравнение — сравнение знака, который вернёт функция, и знака-разделителя. Это значит, что знак, полученный от функции, вновь будет потерян — ведь в первую очередь выполнится сравнение, которое выдаст значение "истина" или "ложь". Присваивать значение переменной `a` слишком поздно! Поэтому необходимо воспользоваться скобками.

```
while ((a = getchar()) != RAZDEL)
```

Уже похоже на то, что мы имеем в примере функции. Заменим `a` на `*bufer++` (взять значение по адресу `bufer`, затем прирастить адрес). Затем подключим второе условие (`schetchik < BUF`).

В теле цикла остаётся только приращать счётчик. Также перед выходом из функции я добавил запись "нуля" в конец буфера, чтобы его содержимое являлось строковым значением. Также можно было добавить

```
return schetchik;
```

Тогда функция будет возвращать количество введённых знаков в получившемся буфере.

Порядок выполнения действий

Здесь пора поговорить о том, в каком порядке выполняются действия, стоящие за теми или иными обозначениями. К примеру, только что нам встречалась запись

```
*bufer++
```

Прирастить адрес "буфера" и взять значение? Или сначала взять значение, а потом прирастить адрес? Правильный ответ — второй. Собственно, порядок действий соответствует порядку записи. Если написать

```
*++bufer
```

, то первым делом последует уже приращение.

Как быть, если требуется прирастить не адрес указателя, а значение, на которое он указывает? Вот так:

```
(*bufer)++
```

И как и в Бейсике, в случае если требуется принудительно задать порядок выполнения действий — используются скобки. Полезно правило: "если не уверен — поставь скобки".

Вот порядок срабатывания действий, которые нам встречались до сих пор:

```
( ) [ ]
* (указатель)   & (взять адрес)   - (отрицат. значение)   !   ~   ++   --
* (умножение)   /   %
+   - (вычесть)
>>   <<
<   >   <=   >=
==   !=
& (логическое "И")
^
|
&&
||
=   +=   -=   *=   /=   %=   >>=   <<=   &=   ^=   |=
```

Каждое действие в этом списке перебивает все действия, стоящие по списку ниже. Те действия, которые записаны в одной строке, равнозначны друг другу. При этом равнозначные действия не всегда имеют порядок выполнения слева направо. Я убеждён, что проще для подстраховки пользоваться скобками, чем держать в голове порядок их выполнения.

Задача 1

Измените функцию `vvod_buf()` так, чтобы она записывала в буфер не вводимые знаки, а те, код которых в "ASCII" больше на единицу чем у вводимых. То бишь если вы введёте `abcd`, то в буфер запишется `bcde`. Может показаться, что ценность этого упражнения никакая. Но уверен, что оно в силах проверить то, насколько вы усвоили порядок срабатывания обозначений.

Число из строки

Предположим, что всё, записанное в буфер созданной нами функцией `vvod_buf()`, следует обрабатывать как число. Напишем функцию `vtseloe()`, которая ищет в строке последовательность цифр, и первую найденную последовательность преобразует в "целое число". Такая функция уже имеется в библиотеке (и называется `atoi()` — прим. перев.), но в учебных целях сделаем её сами.

Предусмотрим случаи, когда: перед цифрой следуют пробелы; также перед ней может стоять знак положительности \ отрицательности; после цифр(ы) может быть любой знак. Таким образом, следующие примеры

```
137d
    137
+137?
```

должны быть восприняты как число сто тридцать семь. Во втором случае за числом ничего не следует, но надо учитывать, что в конце строки всегда будет знак "нуль".

По крайней мере всё это — конечная цель. Начнём с простого. Предположим, что искомое число в буфере всегда будет со знаком, и что переданный в функцию указатель указывает на его знак. Напишем функцию для извлечения числа из буфера, которая будет использоваться функцией `vtseloe()`.

```
int izv_chislo(u)
char *u;
{
    int itog, znak, vechno;
    itog = 0; znak = vechno = 1;
    if (*u++ = '-')
         znak = -1;
    while (vechno)
    {
         itog += (*u++ - '0');
         if ( !tsifra_li(*u) )
              return(itog * znak);
         itog *= 10;
    }
```

```
        return 0;
    }
```

Даже когда мы "начали с простого" — без объяснений не обойтись. Сначала взглянем на это:

```
    znak = vechno = 1;
```

Конечно, можно было бы пойти привычным путём -

```
    znak = 1;    vechno = 1;
```

Однако первый случай является ещё одним образцом присваивания, выполняемого сразу с другим действием (в данном случае с ещё одним присваиванием). Значением переменной `znak`, равным 1, я обозначил плюс, положительный знак числа.

```
    if (*u++ = '-')
        znak = -1;
```

Это — проверка на случай, если на деле знак у числа — минус. Тогда `znak` получает значение -1. Одновременно приращается адрес указателя, чтобы указатель установился на первую цифру.

Далее начинается безостановочный цикл. Чтобы объяснить, что в нём происходит, возьмём пример — число +31 . Содержимое буфера с ним будет выглядеть так:

Введённые знаки	+	3	1	\0	(вместо "нуля" может быть любой другой знак, кроме цифр)
Значения в десятичном виде	43	51	49	0	

```
      u
```

В цикле вычисляется значение переменной `itog`: к нему добавляется 51 - 48 (48 — код ноля-цифры 0 в "ASCII") = 3[1]. После чего адрес указателя приращается. Значение, на которое начинает указывать после этого указатель, проверяется — является ли оно цифрой, при помощи отдельной функции `tsifra_li()` (её создание будет рассмотрено далее); если да — эта функция возвращает значение 1 ("истина").

Следующий знак 1 является цифрой, поэтому `itog` умножается на 10 — получается 30. Возврат в начало цикла. 49 - 48 = 1, поэтому `itog` увеличивается на единицу — получаем 31.

К этому времени указатель указывает на "нуль", либо на другой знак не являющийся цифрой. Следовательно, в цикле выполняется умножение переменной `itog` на `znak`, и полученное возвращается в вышестоящую функцию. Так как `znak` имеет значение 1, то получается 31 — искомое положительное число. Если бы число в буфере было с минусом, `znak` бы имел значение -1, и умножение бы дало нам -31.

Чтобы всё это работало правильно, требуется два условия. Первое — наличие знака положительности или отрицательности числа, что уже упоминалось. Второе: по крайней мере первый знак после плюса или минуса должен быть цифрой, потому что к переменной `itog` хотя

---

1 Так как в таблице кодировки "ASCII" цифры 0-9 следуют подряд друг за другом, то и номера их ячеек — т.е. их коды — таковы, что если из кода 8 вычесть код 0, то получится десятичное значение 8; если из кода 6 вычесть код 2 — получится десятичное значение 4. Вычитание кодов этих цифр друг из друга равносильно вычитанию друг из друга самих чисел от 0 до 9. (прим. перев.)

бы один раз будет прибавлено значение, полученное вычитанием 48-ми из кода прочитанного знака.

Всё это означает, что перед вызовом функции `izv_chislo()` необходимо удостовериться, что эти условия выполнены. Для этого напишем функцию `podgot()`, которая принимает два указателя: указатель на исходный буфер, и массив знаков ввода под названием `chislo`.

Задача функции `podgot()` — сделать из буфера и записать в `chislo` строковое значение, которое уже можно подавать в `izv_chislo()`. Также `podgot()` должна возвращать значение 1 или 0 ("истина" или "ложь" соответственно), в зависимости от успеха выполнения.

```
int podgot(bufer, chislo)
char *bufer, *chislo;
{
    while ( !tsifra_li(*bufer) )
         if (!*bufer++)
              return 0;
    if (*(bufer - 1) == '-')
         strcpy(chislo, bufer - 1);
    else
    {
         strcpy(chislo, "+");
         strcpy(chislo, bufer);
    }
    return 1;
}
```

Цикл занимается поиском первой цифры. Если он её не найдёт, и наткнётся на значение 0 (то есть на завершающий "нуль") — возвращаем 0 в вышестоящую функцию. Если цифре(-ам) предшествовал знак минус — то просто записываем строковое значение по адресу массива, начав читать его с адреса знака "минус". В противном случае сначала записываем в массив знак плюс, а затем дописываем значение, прочитанное из буфера.

Что касается функции `tsifra_li()` — она проста:

```
int tsifra_li(a)
char a;
{
    if (a >= '0' && a <= '9')
         return 1;
    else
         return 0;
}
```

Здесь можно было бы написать более прямо:

```
if (a >= 48 && a <= 57)
```

Однако в этом случае не сразу догадаешься, что за этими значениями стоят коды знаков.

Взгляни вы на этот код спустя некоторое время — вы бы тоже не сразу поняли. По этой же самой причине ранее я писал

```
while (vechno)
{
    itog += (*u++ - '0');
```

Очень полезно принимать меры против вероятной головной боли в будущем.

И наконец, создадим же главную функцию — `vtseloe()`.

```
int vtseloe(bufer)
char *bufer;
{
    char chislo[BUF + 1];
    if ( !podgot(bufer, chislo) )
    {
        printf("Не является целым числом");
        exit(0);
    }
    return izv_chislo(chislo);
}
```

Массиву `chislo` задан объём `BUF + 1` по той причине, что в буфере могут быть только цифры. В этом случае требуется, чтобы в массиве была предусмотрена ещё одна ячейка для знака положительности или отрицательности числа.

Вместо оправданий

Теперь о том, о чём вы скорее всего подозревали. Всё что разыгралось выше — явно не является разумным программированием. Если хотите убедиться окончательно — знайте, что упомянутая функция `atoi()` из библиотеки Си, вместо которой мы создавали `vtseloe()` и всё остальное, состоит из примерно девяти строчек кода, и не вызывает никаких других функций. Прямо сейчас моя задача — не писать вместе с вами безупречный краткий код, а рассматривать различные примеры и приёмы, подходы и решения.

scanf()

Среди библиотечных функций ввода функция `getchar()` не единственная. Есть ещё функция `scanf()`, можно сказать обратная функции вывода `printf()`.

Пример её использования:

```
scanf("%4d %c %2d", &a, &bukva, &b);
```

(Для всех, кто, читая это впервые, ищет "а где же здесь место для ввода". Ввод осуществляется с клавиатуры, а функция ожидает этого ввода. — прим. перев.)

Конечно, перед этим используемые переменные должны быть объявлены:

```
int a, b;
char bukva;
```

Если при работе программы, когда она дойдёт до этой функции, мы введём

`3162 k 47` вводить следует именно так — всё подряд, с пробелами

, то переменным будут соответственно присвоены значения:

```
a = 3162
bukva = 'k'
b = 47
```

Главное, что следует усвоить — все аргументы (входные данные) для функции scanf() должны быть указателями. Я подал в функцию именно адреса ячеек переменных, а не их имена.

В примере я заранее определил для входных значений место на экране, соответствующее их длине. Но я мог написать и так:

```
scanf("%d %c %d", &a, &bukva, &b);
```

Тогда вводимые пробелы будут восприняты как разделители, и место на экране также будет успешно распределено.

Однако если при вводе пробелов не будет (т.е. если ввести `3162k47`), то в случае с заранее распределённым местом всё сработает правильно, а во втором случае — уже нет.

Мы бы могли сейчас сделать собственное подобие `scanf()`, но что-то мне подсказывает, что в этом нет необходимости. В любом случае я пока рассказал о ней немного, остальные подробности оставим на потом.

Ответы

Задача 1

```
int vvod_buf(bufer)
char *bufer;
{
    int schetchik;
    schetchik = 0;
    while((*bufer++ = getchar()) != RAZDEL && schetchik < BUF)
    {
         schetchik++;
         (*(bufer - 1))++;
    }
    *bufer = '\0';
    return 0;
}
```

Внутри цикла нам требуется прирастить значение предыдущего введённого знака. Именно в этом смысл записи `(bufer - 1)` — предыдущий адрес от адреса `bufer`.

ЗАДАНИЕ

Напишите программу-шифровальщик, работающую путём замены знаков. При запуске она должна запросить ввод "шифрованного алфавита". Например такого:

```
ЕХИБПШДВЙЩЭТГНЯРАЛУСЮАЬФЁЧКОЗЪЫЦЖ
```

После этого пусть программа свободно принимает ввод, и выдаёт его "зашифрованный" вид. Каждая буква обыкновенного алфавита должна быть подменена соответствующей буквой "шифрованного". Чтобы, используя пример, зашифровать что-нибудь в слово `БОЯРИНЪ`, вам придётся ввести[1]

```
ХРЖЛЩЯО
```

1 При использовании исходного алфавита из 33-х букв, вида `абвгдеёжзийклмнопрстуфхцчшщъыьэюя`. (прим. перев.)

ГЛАВА 8

Вывод

Вторая часть обещанного разговора.

Подробнее о "printf()"

Как мы знаем, этой функции необходимо указывать вид данных выводимого значения. И пока из обозначений, применяемых в функции для этих целей, нам известны только `d` (представить в десятичном виде) и `c` (обработать как знак ввода). Вот ещё несколько:

- `o` в восьмиричном виде
- `x` в 16тиричном виде
- `u` в десятичном беззнаковом виде
- `s` обработать как строковое значение
- `f` в десятичном виде с плавающей точкой

Пример:

```
printf("%s%5.3f", u, x);
```

Предположим, что `u` — это указатель на строку "Вычисление дало", а переменная `x` содержит значение 5.813. Тогда вывод будет выглядеть так:

```
Вычисление дало5.813
```

Обратите внимание на запись `5.3` в управляющей строке. Как мы знаем, число перед обозначением вида данных выводимого значения указывает, сколько разрядов отвести для записи значения. В данном случае записано два числа, 5 и 3, через точку-разделитель. Два числа нужны в случае со значением с плавающей точкой. Первое, как обычно, указывает сколько разрядов на экране отвести для записи значения, а второе — точность десятичного значения, в количестве знаков после запятой.

Длина строки, как видите, в управляющей строке не указана. Это и не требуется, так как окончание строки будет определено по знаку "нуль".

Откровенно говоря, любое выводимое функцией `printf()` значение в любом виде данных преобразуется в строковое перед самым выводом на экран. Поэтому не указанная длина значения к ошибкам в работе программы не приведёт. Всё это требуется для упорядочивания вывода на экране.

Выравнивание

Если мы зададим для выводимого значения больше места на экране, чем нужно, то пустующие разряды будут заполнены пробелами. По умолчанию пробелы появятся слева от значения (т.е. значение будет выровнено по правому краю отведённого пространства).

```
printf("%s%6.3f", u, x);
```
отводим пространства больше на один разряд

```
Вычисление дало 5.813
```
получаем вывод

Можно задать выравнивание значения по левому краю отведённого пространства, поставив знак минус перед указанием количества отводимых разрядов.

```
printf("%s%-6.3f", u, x);
```

```
Вычисление дало5.813
```

получаемый вывод

пробел

Вывод в ячейки памяти

Существует функция `sprintf()`, которая отличается от `printf()` только назначением вывода: она передаёт принятые данные в массив, в качестве строкового значения. Например, значение `b` равно 7, значение `a` — "3AD2" (в 16тиричном виде). Тогда в ходе этого действия

```
sprintf(m, "%3d%5x", b, a);
```

в массив `m` будет записано строковое значение:

```
7 3AD2
```

(Обратите внимание на пробел, возникший из-за излишка отведённого пространства для значения `a`).

Для чего это может быть нужно? Рассмотрим на простом примере. Предположим, что ваша программа вычисляет денежные суммы, притом с копейками. Переменные можно объявлять как "целое число", но при выводе хотелось бы видеть нечто вроде P38.73, а не 3873. Для этого можно написать функцию `print_dengi()`, которая принимает сумму выраженную целым числом в копейках, а выводит — в рублях и копейках.

```
int print_dengi(kop)
int kop;
{
    char vivod [8] = "         ";      /* здесь 8 пробелов */
    *vivod = 'P';
    sprintf(vivod + 1, "%5d", kop);
    *(vivod + 6) = *(vivod + 5);
    *(vivod + 5) = *(vivod + 4);
    *(vivod + 4) = '.';
    printf(vivod);
    return 0;
}
```

(Здесь `P` используется в качестве условного знака рубля. Использовать международный знак рубля ₽ получится только при условии работы в кодировке Юникод, что требует использования особых функций, работа с которыми в книге не рассматривается. — прим. перев.)

Прежде всего на первое место в строке записывается знак рубля. Затем (по адресу `vivod + 1`) подряд записывается сумма, выраженная в копейках. После чего значения из

двух самых младших разрядов сдвигаются на разряд вправо, а на освободившееся место записывается точка. Можно выводить.

Как видите, предварительно записанное в память строковое значение обеспечивает дополнительную гибкость при выводе.

Простейший вывод

Так же, как функция `getchar()` записывает куда-либо один введённый с клавиатуры знак, функция

```
putchar(a);
```

выведет знак ввода, который хранится в `a`.

Учитывая, что функции `printf()` и `sprintf()` предоставляют куда более широкие возможности — можно задаться вопросом, действительно ли она необходима.

Довольно распространённым примером применения будет функция, позволяющая набирать текст до тех пор, пока не будет введён знак, принятый за разделитель.

```
int pechatalka()
{
    char a;
    while((a = getchar()) != RAZD)
        putchar(a);
    return 0;
}
```

Вряд ли что-то здесь требуется разъяснять. Принятый знак просто бросается на вывод. Зачем всё это? Вспомните — я упоминал, что источник ввода и назначение вывода можно менять. (Как это сделать — я не говорил, да.) Если мы назначим источником ввода содержимое файла, то вышеприведённый пример выведет нам его на экран. Такую задачу решим в главе 12.

ГЛАВА 9

Дополнительные операторы условия

В главе 3 мы рассмотрели бóльшую часть операторов условия в Си. Осталось ещё несколько. Также, помимо них, познакомимся с несколькими новыми ключевыми словами, которые дают дополнительные возможности по управлению циклами.

Собственно об условиях

Сейчас вы увидите сокращённый и предельно скрытный способ записи привычного условия `if...else`. Вот он.

```
условие ? действие1 : действие2
```

Ещё раз — эта запись полностью равносильна обыкновенной:

```
if (условие)
    действие1
else
    действие2
```

Да, единственное применение этого — получение более короткого кода. Для примера перепишем функцию `abs()` из задачи 1 в третьей главе.

```
int abs(znachen)
int znachen;
{
    return (znachen < 0 ? - znachen : znachen);
}
```

Задача 1

Перепишите таким же образом функцию `chetn()` из главы 4.

Прерывание

Иногда требуется делать остановку работы цикла. Предположим, что у нас есть два буфера, для обращения к ним используются указатели `u` и `v`, и наша задача — переписать содержимое одного буфера в другой. Примем, что объём буферов — 100 знаков.

```
for (scht = 0; scht < 100; scht++)
    *v++ = *u++;
```

Но в случае, если будет прочитан знак "нуль", цикл требуется остановить, и не переписывать ничего из оставшегося. Для этого потребуется волшебное слово на букву "б":

```
for (scht = 0; scht < 100; scht++)
{
    if (!*u)
         break;
    *v++ = *u++;
}
```

Слово `break` (или по-русски "бряк" — прим. перев.) прерывает работу цикла, в котором оно находится, с последующим выходом из цикла.

Пропуск прохода цикла

Ещё можно принудительно начать следующий проход цикла, не завершив текущий. Другими словами — с какого-то места пропустить оставшуюся часть тела цикла. Изменим наш пример так, чтобы не записывать ноли в целевой массив:

```
for (scht = 0; scht < 100; scht++)
{
    if (!*u)
         break;
    if(*u++ == '0')
         continue;
    v++ = *(u-1);
}
```

Как вы поняли, за пропуск прохода отвечает слово `continue`. Чтобы не нарушить правильную работу цикла, мы приращаем адрес `u` прямо в проверке условия на пропуск прохода, и ниже из-за этого приращения вынужденно делаем поправку (`u-1`).

Во многих случаях в коде можно придумать некую замену словам `break` и `continue`, и обходиться без них. Часто это выливается в сложносоставленные условия, которые тяжело читать и разбирать. Но в случае с двумя вышеприведёнными примерами обойтись без этих ключевых слов нетрудно, и здесь вы встаёте перед выбором — как быть?

Ответ даст только опыт, который будет подсказывать, какой подход более приемлем в каждом случае. Старайтесь чувствовать удачность, правильность, естественность кода. Не забывайте про читаемость.

Переключатель

Порой требуется на основе проверки простого условия делать выбор из нескольких дальнейших действий. Оператор `if...else` обеспечивает выбор только из двух путей. Хотя с помощью вложенности можно добиться и большего...

```
if (условие1)
    действие1
else
    if (условие2)
         действие2
    else
         if (условие3)
              действие3
              и т.д.
```

Но это не самый опрятный способ решения такой задачи. Си предоставляет средство для более множественного ветвления и выбора по нему, и называется оно переключатель.

Для примера представим себе программу работы с базой данных, управляемую через такое меню.

```
1. Создать новую БД
2. Добавить запись
3. Удалить запись
4. Изменить запись
5. Поиск
Введите номер требуемого действия (1-5):
```

Каждое из перечисленных действий выполняется отдельной функцией. За меню также отвечает функция `menu()`; она же и возвращает введённый пользователем номер действия.

Вот пример работы переключателя (`switch`) в этом случае:

```
switch (menu())
{
    case 1 : sozdat();
         break;
    case 2 : dobavit();
         break;
    case 3 : udalit();
         break;
    case 4 : izmenit();
         break;
    case 5 : poisk();
         break;
    default: oshibka = 1;
         printf("Ошибка\n");
         break;
}
```

Пример нагляден. Словами `case` (досл. "случай" — прим. перев.) обозначаются собственно ветви, а после них записываются значения, которые служат их номерами или именами. Переключатель выбирает ту ветвь, чей номер \ имя соответствует введённому значению (в нашем случае значению, которое вернёт функция `menu()`).

Важно понимать, что тело переключателя будет выполняться от точки входа — и до конца сверху вниз, поэтому ключевые слова `break`, прерывающие выполнение, необходимы.

Также обратите внимание на слово `default`. Если на ввод поступило нечто, для чего нет ветви с соответствующим именем — то выполняется ветвь `default`.

Порядок записи всех этих ветвей не имеет никакого значения (поэтому в конце ветви `default` я дописал слово "бряк"; благодаря этому позже можно будет просто дописать новые ветви в конце тела переключателя).

С сомнением про "goto"

Великий и ужасный `goto` мы с вами ещё не вспоминали. В Бейсике его использования избежать трудно. Однако я не думаю, что вплоть до этой секунды вам не хватало этого оператора в Си. Несмотря на это — в Си он есть. Пример использования:

```
goto nazovite_bukvu;
```

Предположим, что где-то в окружающем коде есть строчка

```
nazovite_bukvu: a = getchar();
```

, и тогда `goto` перебросит программу на неё. Метка для перехода пишется по тем же

правилам, что и имя переменной, и в целевой строке метка отбивается от собственно строки двоеточием, как показано выше. Это более читаемо, чем

```
GOTO 1495
```

в Бейсике. Но если в коде таких переходов слишком много — отслеживать их тяжело.

В большинстве случаев можно обойтись без использования `goto`; за восемь лет программирования на Си я ни разу не воспользовался им. И да, я не призываю считать всех использующих `goto` еретиками, просто, как я уже говорил — я сторонник естественности и красоты кода. В большинстве случаев `goto` выглядит неуклюже.

Единственный случай применимости `goto`, который я могу себе представить — использовать его внутри многократно вложенных циклов, когда из всех них требуется разом выйти.

```
while(...)
{
    ...
    while(...)
    {
        ...
        while(...)
        {
            ...
            if (x < 0)
                goto vihod;
        }
    }
}
vihod: ...
```

Напоминаю, что слово `break` выводит программу только из того цикла \ той функции, в котором \ которой оно находится. Поэтому в каждом из этих циклов пришлось бы повторять строки для выхода:

```
if (x < 0)
    break;
```

Как грубо.

Ответы

Задача 1

```
int chetn(n)
int n;
{
    return (n%2 ? 0 : 1)
}
```

ГЛАВА 10

Рекурсия (самовызов функции)

Маленьких детей учат, что нельзя объяснять явление или предмет путём называния этого предмета или явления. Те, кого этому в своё время не научили, пишут в учебниках: "Сила упругости — это сила, действующая ..." и так далее.

Но в программировании допустимы подобные определения, частично ссылающиеся сами на себя. Речь идёт о функциях, которые в своём теле вызывают себя же. Для такого явления принято название "рекурсия".

Даже в Бейсике есть возможность писать такие функции, поэтому ожидаемо, что и в Си это позволяется. Очевидно, делать это следует с некоторой продуманностью. Нельзя писать нечто такое:

```
nechto(a)
int a;
{
    nechto(a);
}
```

Такая запись приведёт к безостановочному циклу — функция вызывает функцию, которая вызывает функцию, которая вызывает функцию, ... При каждом из этих вызовов требуется запомнить точку возвращения, поэтому с каждым вызовом расходуется всё больше памяти. Рано или поздно вся память будет забита.

Значит, в теле функции должен быть предусмотрен выход из неё, а также условие, при котором он выполняется.

Факториал числа

Введём рекурсию по ходу написания функции вычисления факториала числа. Факториал числа `n` (обозначим его как `n!`) представляет собой последовательное умножение естественных чисел:

```
n! = 1 x 2 x 3 x ... n
```

Само собой напрашивается использование цикла.

```
int faktorial(n)
int n;
{
    int itog;
    itog = 1;
    while (n--)
         itog *= n;
    return itog;
}
```

Как видно, использован обыкновенный приём отсчитывания от исходного значения до ноля.

Можем ли мы выразить факториал через факториал?

```
n! = n(n-1)!
```

Таким образом, например,

```
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
```

Вот и пришло время использовать рекурсию в нашем коде.

```
int faktorial(n)
int n;
{
    if (n==1)
        return 1;
    return n * faktorial(n -1);
}
```

Конечно, данный пример прост, и в рекурсии в данном случае нет необходимости. Но даже здесь видно, насколько она, при всей своей непривычности, может упрощать код. В более сложных случаях польза от рекурсии может быть весьма обширна.

Шаманство над строковыми значениями

В качестве следующего примера возьмём задачу по обращению порядка знаков в строковом значении (например, `сообщение` изменится на `еинещбоос`). Её можно было бы решить и привычными способами, но сегодня мы сознательно идём другим путём, дабы опробовать новый подход.

Для начала следует рассмотреть самый простой случай. Самое короткое возможное строковое значение, в котором придётся менять порядок знаков — состоит из двух знаков.

Поменять два знака местами очень просто, но я буду настаивать на использовании двух дополнительных функций. Первая пусть возвращает самый первый знак в строке, а вторая — указатель на оставшиеся знаки; назовём их "начало" и "остаток".

В нашем простейшем случае задача сводится к тому, чтобы после "остатка" приписать "начало". Мы не сможем воспользоваться библиотечной функцией `strcat()`, поскольку "начало" вернёт нам только один знак, а не строковое значение. Поэтому понадобится ещё одна вспомогательная функция, которая умеет приписывать к концу строкового значения один знак. Создадим её:

```
int pripisalka(u, a)
char *u, a;
{
    while (*u++)
        ;
```

```
    *(u-1) = a;
    *u = 0;
    return 0;
}
```

С помощью пустого цикла адрес указателя приращается до тех пор, пока не будет обнаружен завершающий "нуль". Из-за того что приращение выполняется после проверки условия, адрес в итоге указывает на одну ячейку дальше "нуля". Поэтому знак записывается по адресу `(u-1)`. А в следующую ячейку записывается новый "нуль".

По ходу дела напишем также "начало" и "остаток".

```
char nachalo(u)
char *u;
{
    return *u;
}


int ostatok(u)
char *u;
{
    return (u + 1);
}
```

Вряд ли здесь что-то надо пояснять.

Вооружившись всем этим, пишем основную функцию:

```
int obrat_znaki(u, v)
char *u, *v, *ostatok();
{
    if (strlen(u) == 2)
    {
        strcpy(v, ostatok (u) );
        pripisalka(v, nachalo(u) );
    }
    else
    {
        obrat_znaki( ostatok(u), v);
        pripisalka(v, nachalo(u) );
    }
```

```
        return 0;
    }
```

`u` и `v` — это соответственно указатели на исходное строковое значение и на целевое (в которое будет записываться итоговая строка).

Действия в случае, если строка состоит из двух знаков, очевидны. А вот в случае, если строка длиннее — мы вновь применяем рекурсию: функция вызывает сама себя, и в этом вторичном вызове обрабатывается уже более короткая строка (короче на 1 знак). И какова бы ни была длина строки — когда она, вследствие множества дальнейших вызовов функцией самой себя, сократится до двух знаков, то функция наконец поменяет эти два знака местами. После чего начнётся волна возвратов, при каждом из которых "приписалка" будет добавлять в конец строки по одному знаку, пошагово воссоздавая исходную строку в обратном порядке.

Понять всё это сразу не так просто, поэтому рассмотрим на примере. Пусть исходная строка — `абвг`. Её длина больше двух знаков, следовательно функция вызывает сама себя, чтобы обратить `бвг`, а потом в конце дописать `а`. Но `бвг` тоже длиннее двух знаков, поэтому функция ещё раз вызывает сама себя, обращает `вг` и получает `гв`. Происходит возврат на уровень выше, в ходе которого к `гв` приписывается `б`. Ещё один возврат на уровень выше, и к `гвб` приписывается `а`. Получаем строку в обратном порядке — `гвба`.

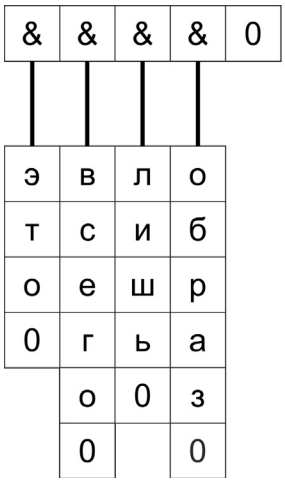
Только не пытайтесь таким же образом разбирать все свои будущие функции, построенные на рекурсии. Ведь чаще всего уровней вложенности (самовызовов функции) получается огромное количество. В таких случаях попытка пошагово представить себе всю картину происходящего скорее всего выльется в страх и ненависть.

Обратный порядок слов

Продолжим наши опыты. Поставим задачу перестановки слов в строке наоборот, без затрагивания порядка букв в словах.

Всё это классические примеры, но подобные задачи нередко бывают полезны при обработке текста. К тому же требуется поразмыслить над тем, как в этом новом случае лучше хранить обрабатываемые данные.

Ведь теперь нам нужно работать только с началом и окончанием каждого слова — сам порядок букв в слове должен остаться прежним. Поэтому представить всю входную запись как одно строковое значение — не лучшее решение, поскольку придётся искать разделители слов. Если же мы представим каждое слово как отдельное строковое значение, то текст можно выразить через массив указателей на эти строковые значения.



массив указателей
в виде строкового значения

Вот так выглядит объявление массива указателей, в данном случае на значения вида "знак ввода":

```
char *u[200];
```

Мы ещё встретим и более замысловатые объявления.

Заворачиваем и разворачиваем

Начнём воплощать задуманное. Напишем две функции, `vukazateli()` и `izukazatelei()`. Первая должна читать исходное строковое значение и делать из него "свёртку" в виде указателей, как показано выше, а вторая — выводить строку с использованием этих указателей.

```
int vukazateli(stroka, u, svertka)
char *stroka, *u[], *svertka;
{
    int a;
    a = 0;
    while (*stroka)
    {
        u[a++] = svertka;
        while ((*svertka++ = *stroka++) != ' ')
            ;
        *svertka++ = 0;
    }
    u[a] = 0;
    return 0;
}
```

Обратите внимание на объявленный массив указателей. Нет нужды прописывать его объём, потому как он (объём) будет взят из вызывающей функции. В область, куда указывает имя `svertka`, будет записан ряд разделённых "нулями" слов из исходной строки — на них будут указывать указатели из массива.

Наружний цикл `while` проверяет, является ли текущий знак в исходной строке завершающим "нулём". Если нет — указатель из текущей ячейки массива получает текущий адрес области `svertka`, и в эту область записывается следующее слово вместе с пробелом после него. После каждого слова записывается "нуль". Последним действием в конечный указатель в массиве мы записываем адрес 0, чтобы обозначить окончание.

Признаю, что использовать в качестве строкового разделителя указатель с адресом 0 — затея несколько спорная, ведь адрес 0 существует и может задействован. (Отрицательного адреса указателя быть не может, поскольку указатель имеет беззнаковый вид данных.) Следовательно, можно было бы ожидать, что в каком-то случае область `svertka` будет изначально иметь адрес 0, и первый указатель в массиве получит этот адрес, который затем будет прочитан как разделитель.

Как правило, в вызываемых функциях переменные получают более высокие адреса. Что касается нашего случая — в вызывающей функции область `svertka` может получить адрес 0 только в случае, если она объявлена самой первой. За этим легко проследить и не допустить этого.

Заметьте, что внутренний цикл может остановить только найденный пробел, поэтому в конце исходной строки пробел тоже должен быть. Но код легко можно изменить, чтобы распознавались также другие знаки. А можно просто дописать в конец строки пробел, перед началом основных действий.

После всего этого вторая функция покажется простой.

```
int izukazatelei(u)
char *u[];
{
    int a;
    a = 0;
    while (u[a++])
         printf(u[a-1]);
    return 0;
}
```

Просто выводим строковые значения с помощью каждого указателя в массиве, до тех пор пока не будет прочитан адрес 0.

Обратный порядок слов. Продолжение

Скорее всего вы уже поняли, что задача обратной перестановки слов во входной строке свелась к перестановке указателей в обратном порядке. Написанные нами функции отлично справятся, но в них требуется поменять вид данных. Там где был вид данных "знак ввода" — будет указатель на "знак ввода"; а где уже был указатель на "знак ввода" — будет... указатель на указатель на "знак ввода".

Изменим соответствующим образом функции "начало" и "остаток":

```
char nachalo(u)
char **u;
{
    return *u;
}

int ostatok(u)
char **u;
{
    return (u + 1);
}
```

Вы увидели, как обозначается указатель на указатель? Две звёздочки подряд — это указатель на указатель. (Я придаю значимости мгновению, нда.)

Функцию "приписалка" тоже надо изменить, и учтём, что теперь она будет приписывать в конец строкового значения не отдельный знак, а одно слово.

```
int pripisalka (u, a)
char **u, *a;
```

```
{
    while (*u++)
        ;
    *(u-1) = a;
    *u = 0;
    return 0;
}
```

Создаём главную функцию.

```
int obrat_slova(u, v)
char **u, **v, **ostatok();
{
    if (dlina_ukz(u) == 2)
    {
        strk_vmsv_ukz(v, ostatok(u));
        pripisalka(v, nachalo(u));
    }
    else
    {
        obrat_slova(ostatok(u), v);
        pripisalka(v, nachalo(u));
    }
    return 0;
}
```

В ней потребовалось заменить библиотечные функции `strlen()` и `strcpy()` похожими функциями, предназначенными для работы с массивами указателей на строковые значения.

Задача 1

Напишите функции `dlina_ukz()` и `strk_vmsv_ukz()`, которыми в коде выше заменены `strlen()` и `strcpy()`.

Ответы

Задача 1

```
int dlina_ukz(u)
char **u;
{
    int a;
    a = 0;
    while (*u++)
        a++;
    return a;
```

```
}
int strk_vmsv_ukz(massiv, ish_stroka)
char **massiv, **ish_stroka;
{
    do
        *massiv++ = *ish_stroka++;
    while (*ish_stroka);
    return 0;
}
```

Эти функции, как и прочие в конце этой главы, отличаются от заменяемых только бóльшим количеством звёздочек.

ЗАДАНИЕ

Загляните в главу 12. Отыщите там описание поиска делением пополам. Перепишите это описание таким образом, чтобы оно включало использование рекурсии.

Дополнение

Эта глава началась с объяснения рекурсии, и плавно перетекла в обсуждение работы с данными. Стоило бы ещё раз подытожить всё сказанное.

Обрабатывать любые данные можно опосредованно, через указатель на них. В свою очередь, эти данные также могут быть представлены указателями на другие данные, и эта многоуровневость может быть сколь угодно запутанной.

Также обратите внимание, как была создана итоговая функция перестановки слов. Сначала все более простые функции, из которых она состоит, мы обкатали на более простых случаях, а потом просто внесли в них необходимые небольшие изменения. Вероятно, опытный программист способен написать всё сразу в готовом виде. Но в тех случаях, когда работа с данными в том коде, который вы создаёте, слишком сложна для того чтобы её вообразить и постоянно держать в голове, такой подход может очень облегчить жизнь.

Наконец, о пройденном подходе к хранению данных. Массив, который мы с вами создали, по сути — двумерный. Легко представить его в виде таблицы: каждый указатель в массиве — это номер столбца, а ячейки где хранятся буквы слов — построчно следуют друг за другом. Но так как каждый указатель указывает на строковое значение (т.е., в свою очередь, массив знаков ввода) — длина строковых значений может меняться, и зависит от количества записанных в них знаков.

Теперь становится понятным, как получить более многомерные массивы: если двумерный массив — это массив указателей на простейшие массивы, то трёхмерный массив — это массив указателей на двумерные массивы. И так далее. В большинстве языков программирования многомерные массивы выполнены именно так. Язык Си же позволяет чётко увидеть и понять такое устройство.

ГЛАВА 11

Структуры

В шестой главе я завёл разговор о полезности структур, который затем мною же был оборван. Пора его возобновить и продолжить.

Вспомним из шестой главы структуру для записи о каком-либо товаре на складе магазина:

```
struct zapis
{
    int kod_tvr;
    char opisanie [30];
    double tsena;
    int sklad_kolvo;
};
```

Я также писал, что проще простого создать массив из объявленных структур:

```
struct zapis bazadannih [5000];
```

Возникают вопросы: как передать куда-либо содержимое структуры? Как изменить в структуре какое-либо значение?

Как передать? Сразу всю структуру — никак. Только путём передачи каждого значения по очереди. В большинстве случаев часто будет меняться только одно значение в структуре — например, количество товара на складе; однако в случае необходимости передача структуры возможна только указанным нудным способом. Кроме того, структуры не передаются между функциями — передаются только указатели на них, как и в случае с массивами.

Вместо развлечений

Обратиться к значению внутри структуры (назовём его полем) можно двумя способами. Первый записывается так:

```
имяСтруктуры.поле
```

или

```
имяМассиваСтруктур[номерЯчейки].поле
```

Например, обратимся к полю "цена" в третьей записи нашей базы данных:

```
bazadannih[3].tsena
```

Если мне надо удалить `n` единиц товара номер 147 из поля "количество на складе", я пишу:

```
bazadannih[147].sklad_kolvo -= n;
```

Не слишком кратко, но понятно.

Но как мы уже видели, предпочтительный подход в Си — ссылаться на ячейки массива не номерами ячеек, а указателями на них. К тому же между функциями можно передавать только указатели на структуры, и потому мы просто вынуждены прибегать к указателям. Поэтому — вот второй способ обращения к полю в структуре:

```
указательНаСтруктуру -> поле
```

Для нашего примера сначала требуется объявить указатель на структуры вида zapis:

```
struct zapis *uzps;
```

Если uzps указывает на ячейку 147 в "базе данных", то чтобы вычесть n единиц из поля "количество на складе", я пишу:

```
uzps -> sklad_kolvo -= n;
```

При этом предпочтительно, чтобы указатель получал адрес нужной ячейки вследствие выполнения какого-либо действия наподобие поиска, и не было необходимости указывать ему этот адрес вручную.

Также обратите внимание на обозначение обращения к полю -> .

Означенные два способа обращения к полям в структуре можно свободно сочетать. Предположим, нам требуется передать структуру, на которую указывает uzps, в промежуточную структуру vrem:

```
struct zapis bazadannih [500], vrem, *uzps;
...
vrem.kod_tvr = uzps -> kod_tvr;
vrem.opisanie = uzps -> opisanie;
vrem.tsena = uzps -> tsena;
vrem.sklad_kolvo = uzps -> sklad_kolvo;
...
```

Вряд ли здесь имело бы смысл создавать указатель для структуры vrem, просто чтобы был. Но это следовало бы делать в случаях, когда код должен быть предельно читаем. В более коротких обозначениях вероятность сделать опечатку ниже, а увидеть её — выше.

Приторговываем помаленьку

Настало время разработать несколько вещей, призванных помочь в учёте поступающего и убывающего товара.

Поставим задачу, чтобы присутствовало подобное управляющее меню:

```
Добавить запись
Удалить запись
Изменить количество на складе
Показать цену
...
```

Мы уже видели, как это можно сделать посредством переключателя и функции menu(), которая выводит на экран строчки меню и возвращает введённый пользователем выбор. Воспользуемся этим же самым способом:

```
#include <stdio.h>
#include <stdlib.h>
struct zapis
```

```
{
    int kod_tvr;
    char opisanie [30];
    double tsena;
    int sklad_kolvo;
} bazadannih[500];

int main()
{
    int nikogda = 0;
    while (!nikogda)
         switch( menu() )
         {
              case 1: dobavit();
                   break;
              case 2: udalit();
                   break;
              case 3: izm_kolvo();
                   break;
              case 4: pokaz_tsenu();
                   break;
              default: exit(0);
         }
    return 0;
}
```

Пока всё просто. Обратите внимание, что структура и массив структур объявлены короче, чем обычно — в одном и том же действии. Мне не пришлось отдельно писать объявление массива структур:

```
struct zapis bazadannih[500];
```

Так как массив `bazadannih` был объявлен за пределами функции `main()` — он доступен из любого места программы. Пока предположим, что все необходимые данные в нём уже есть; позже узнаем, как можно загрузить эти данные с запоминающего устройства.

Вот содержимое функции `menu()`:

```
int menu()
{
    int vvod = 0;
    printf("Введите номер действия:\n");
    printf("1) Добавить запись\n");
    printf("2) Удалить запись\n");
    printf("3) Изменить количество товара на складе\n");
    printf("4) Показать цену товара\n");
    printf("Для выхода введите любое другое значение\n");
    scanf("%1d", &vvod);
    return (vvod);
}
```

Всё что остаётся для добавления других действий — создать в ней выводы `printf()`, и прописать случаи и действия в переключателе.

Удаление записи

Первым делом возьмёмся за функцию `udalit()`; для её создания придётся решить ещё пару вопросов с массивом.

Как помечать удаляемые записи? Примем, что структуры из массива мы удалять не будем — просто из записей будут вычищены все данные, следовательно они станут пустыми. Для обозначения пустой ("удалённой") записи можно в поле `kod_tvr` писать значение 0. Таким образом, все непустые записи не должны иметь "код товара" 0. Также по коду 0 функция добавления записей сможет отыскивать пустые записи.

Получается, что изначально при создании массива все записи пусты, и должны иметь код 0; но мы уже решили, что наши записи уже заполнены данными. Предположим также, что в конечной записи массива (`bazadannih[499]`) значение поля "код товара" равно -1. Эта запись будет служить разделителем.

```
int udalit()
{
    struct zapis *uzps;
    int vvod;
        char vvod2;
        uzps = bazadannih;
        printf("Введите номер записи для удаления: ");
        scanf("%d", &vvod);
        while (uzps -> kod_tvr != vvod)
        {
```

```
            if (uzps -> kod_tvr < 0)
            {
                printf("Запись не найдена\n");
                return 0;
            }
            uzps++;
        }
        fflush(stdin);
        printf("Описание удаляемой записи: %s . Подтвердите
удаление (д/н): ", uzps -> opisanie);
        scanf("%1c", &vvod2);
        if (vvod2 == 'д')
            uzps -> kod_tvr = 0;
        return 0;
}
```

(Функция `fflush()` здесь очищает буфер потока ввода. Это необходимо, т.к. первая функция `scanf()` оставляет после себя в буфере знак переноса строки (который вводится нажатием `Enter`). Если буфер потока ввода не очистить, то перенос строки будет прочитан из него следующей функцией `scanf()`, вследствие чего пользователю даже не представится возможность сделать ввод самому. — прим. перев.)

Примечания. То, что функция ссылается на структуру `zapis` и создаёт на неё указатель, возможно потому, что структура сделана видимой для всех (объявлена до функции `main()`). Но сам указатель создаётся только в пределах функции `udalit()`.

Также помните, что функция `scanf()` на ввод требует указатели. Поэтому подаём ей адрес переменной (`&vvod`), а не имя переменной (`vvod`).

Приращение `uzps++` означает, что адрес указателя изменится так, чтобы указывать на следующую структуру в массиве.

Если функция найдёт запрошенный номер записи, то попросит подтверждения на удаление; любой ввод, кроме `д`, будет воспринят как отказ от удаления. Если клавиатура пользователя переключена в режим ввода заглавных букв, то происходящее может его смутить \ напугать \ взбесить \ рассмешить подъ лавкою \ привести в уныние.

Поэтому усовершенствуем конец функции:

```
scanf("%1c", &vvod2);
if (vvod2 == 'д' || vvod2 == 'Д')
    uzps -> kod_tvr = 0;
```

Также можно воспользоваться библиотечной функцией `tolower()`, которая (если необходимо) преобразует введённые знаки ввода в строчные:

```
if (tolower(vvod2) == 'д')
    uzps -> kod_tvr = 0;
```

(В этом случае к программе необходимо подключить содержимое файла данных о библиотеке `ctype.h`, т.к. именно он хранит описание функции `tolower()`. — прим. перев.)

Добавление записи

Функция добавления записи должна просто искать любую пустую запись, обозначенную кодом 0, и записывать в неё данные.

```
int dobavit()
{
    struct zapis *uzps;
    uzps = bazadannih;
    while (uzps -> kod_tvr)
    {
        if (uzps -> kod_tvr < 0)
        {
            printf("В базе данных не осталось места\n");
            return 0;
        }
        uzps++;
    }
    printf("Введите код товара: ");
    scanf("%d", &(uzps -> kod_tvr));
    printf("Введите описание товара без пробелов: ");
    scanf("%s", &(uzps -> opisanie));
    printf("Введите цену товара: ");
    scanf("%lf", &(uzps -> tsena));
    uzps -> sklad_kolvo = 0;
    return 0;
}
```

Здесь ничего непонятного быть не должно. Обратите внимание, что ввода количества товара здесь не предусмотрено — для этого потребуется воспользоваться соответствующей функцией.

Задача 1

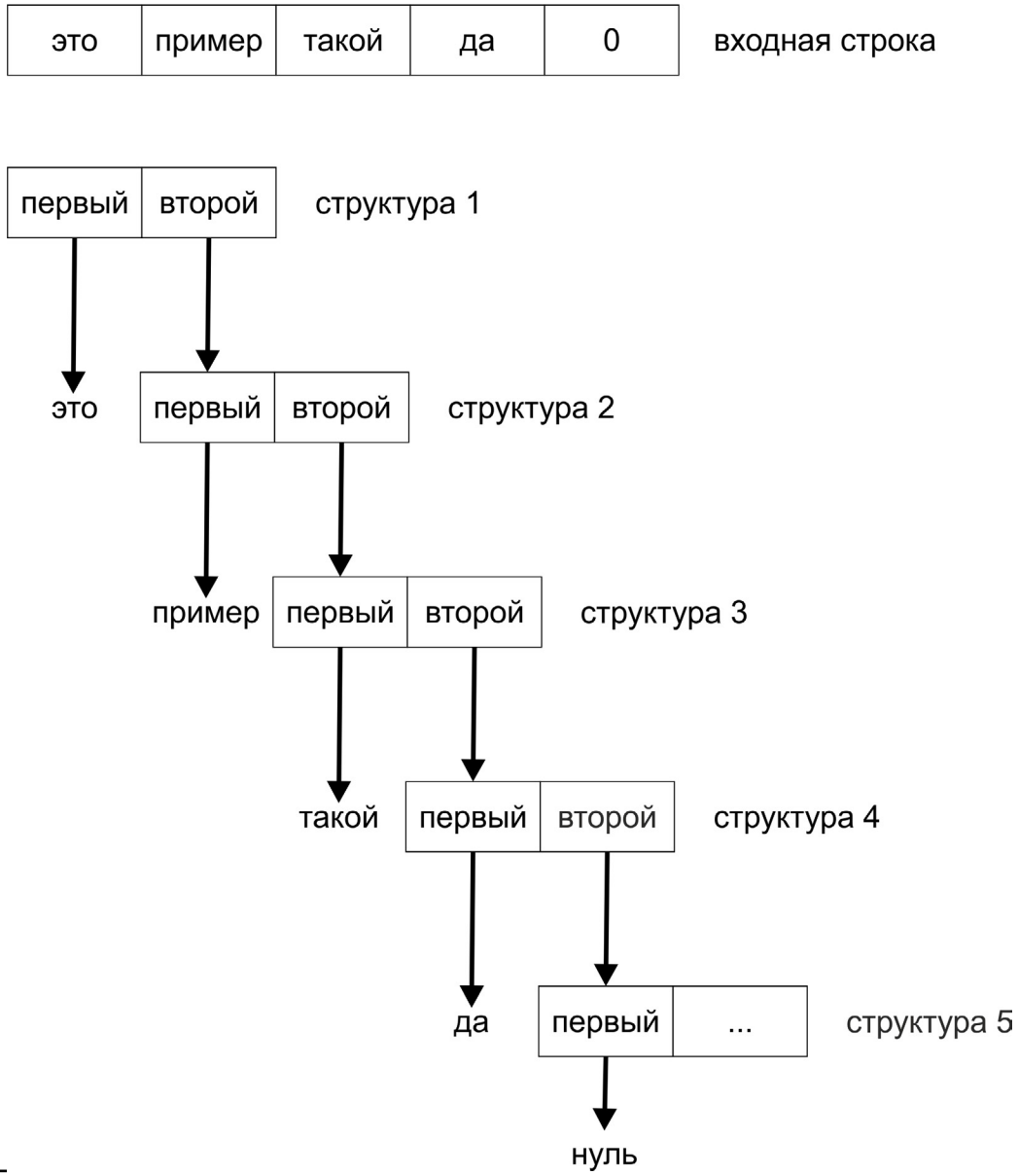
Напишите функции `izm_kolvo()` и `pokaz_tsenu()`.

Рекурсия (самовызов) структур

Вспомните, как мы в главе 10 хранили слова предложения, в котором требовалось переставить слова в обратном порядке. Использовался массив указателей на строковые значения. Затем этот массив функциями `nachalo()` и `ostatok()` условно делился на две части.

Вместо массива указателей на слова (строковые значения) мы могли бы поместить эти слова в ряд структур. Представим, что мы снова находимся в главе 10. Пусть у нас, здесь и далее, уже есть все необходимые функции, определяющие адреса каждого слова в исходной строке и прочее — просто чтобы сейчас про них не думать.[1]

Создадим структуру, которая состоит из двух указателей: первый указывает на первое слово в строке, второй указывает на эту же структуру (точнее — на ещё одну такую же). Тогда, при записи слов из входной строки в такую структуру, мы получим следующее:



1 В этой главе авторы внезапно стали очень ленивыми, то тут то там говоря "представим, что у нас уже всё готово". Но в этом месте исходной книги они вообще ничего не написали про то, какую задачу должна выполнять создаваемая структура, и как в неё должна производиться запись. Поэтому думайте не о том, как будут по-новому выглядеть действия из главы 10, а о том как выглядит и работает вызывающая сама себя (рекурсивная) структура. Это в данном случае — главное, о чём "хотел сказать автор", а остальное — шелуха для примера. (прим. перев.)

Как видите, это ещё один пример применения рекурсии.

Как должна выглядеть структура, о которой говорится в этом примере, в коде?

```
struct nabor_slov
{
    char *perv;                         /* первый указатель */
    struct nabor_slov *na_strukt;       /* второй указатель */
};
```

Важно заметить, что в теле структуры нельзя обращаться к самой этой же \ такой же структуре — только к указателю на неё, как в данном случае.

Ответы

Задача 1

```
int izm_kolvo()
{
    struct zapis *uzps;
    int vvod, velichina;
    uzps = bazadannih;
    printf("Введите код целевого товара: ");
    scanf("%d", &vvod);
    while (uzps -> kod_tvr != vvod)
    {
         if(uzps -> kod_tvr < 0)
         {
              printf("Товар не найден\n");
              return 0;
         }
         uzps++;
    }
    printf("Описание выбранного товара: %s. Введите величину
изменения количества: ", uzps -> opisanie);
    scanf("%d", &velichina);
    uzps -> sklad_kolvo += velichina;
    return 0;
}
```

Эта функция очень похожа на функцию удаления записи. Я только не стал запрашивать у

пользователя подтверждения на внесение изменений, поскольку всегда можно ввести ноль, чтобы ничего не изменить.

```
int pokaz_tsenu()
{
    struct zapis *uzps;
    int vvod;
    uzps = bazadannih;
    printf("Введите код целевого товара: ");
    scanf("%d", &vvod);
    while (uzps -> kod_tvr != vvod)
    {
        if(uzps -> kod_tvr < 0)
        {
            printf("Товар не найден\n");
            return 0;
        }
        uzps++;
    }
    printf("Описание товара: %s.\nЦена: "%lf\n", uzps ->
opisanie, uzps -> tsena);
    return 0;
}
```

У всех созданных функций много общего, не так ли? В таких случаях будьте уверены, что задачу можно было решить проще и изящнее. В наших примерах повторяется кусок кода, выполняющий поиск товара по его номеру. Для этой задачи можно создать функцию, которая принимает на ввод код товара, ищет его и возвращает указатель на запись, к которой он относится. В случае неудачи она пусть возвращает ноль.

```
int poisk_tvr(vvod)
int vvod;
{
    struct zapis *uzps;
    uzps = bazadannih;
    while (uzps -> kod_tvr != vvod)
    {
        if (uzps -> kod_tvr < 0)
            return 0;
        uzps++;
    }
```

```
    return uzps;
}
```

Теперь можно переписать функцию показа цены:

```
int pokaz_tsenu()
{
    struct zapis *uzps;
    uzps = 0;
    int kod;
    printf("Введите код целевого товара: ");
    scanf("%d", &kod);
    if(uzps = poisk_tvr(kod))
        printf("Описание товара: %s.\nЦена: %lf\n", uzps ->
opisanie, uzps -> tsena);
    else
        printf("Товар не найден\n");
    return 0;
}
```

ЗАДАНИЯ

1. Новые записи в нашей базе данных попадают туда, где есть свободное место. Напишите функцию, которая бы занималась перестановкой записей по возрастанию кода товара.

2. Напишите функцию, которая выведет на экран все записи, имеющиеся в базе данных.

3. Измените функцию поиска товара так, чтобы она также принимала на ввод описание товара и могла искать товар по нему.

4. Напишите функцию, которая бы показывала записи о товаре, которого осталось меньше 20 единиц.

ГЛАВА 12

Действия с файлами

То и дело в ранних главах я писал, что те или иные возможности Си зависят от исполнения среды разработки. К счастью, в большинстве случаев разница заключается в наличии \ отсутствии каких-то возможностей, а не в способе их записи и разнице обозначений.

Но когда дело доходит до обработки файлов — разница между исполнениями становится ощутимее. И здесь причина не в прихотях разработчиков сред, а в отличиях, с которыми разные операционные системы работают с файлами.

Лучше всего на Си работается в окружении ОС "Unix", для которой он создавался, и "Unix"-подобных систем. В остальных случаях отличия данного исполнения языка от исходного вида могут быть неизбежны. Чтобы извлечь из этой главы наибольшую пользу, вам следует точно выяснить все особенности вашего исполнения, предназначенного для вашей операционной системы; полагаем, что вы уже знаете достаточно, чтобы изучение этого вопроса не составило для вас трудностей.

Небольшой отказ от ответственности. Примеры кода в этой главе были опробованы на нескольких распространённых видах компьютеров и ОС[1]. Но я не могу обещать, что эти примеры заработают без всяких изменений во всех исполнениях Си (и на всякий случай добавлю, что пусть это касается большинства примеров кода в книге). Следовательно, вам стоит внимательнее относиться к личному опробованию описываемых примеров.

Перенаправление ввода \ вывода

Большинство операционных систем сейчас общаются не напрямую с подключенными устройствами ввода \ вывода, а посредством программных передатчиков, которые называются потоками ввода \ вывода. Каждый поток связан с определённым подключенным устройством. Программа общается только с потоком. В случае, если требуется поменять источник ввода или цель вывода данных, такой подход позволяет не переписывать значительную часть программы, а только поменять поток ввода \ вывода.

Весь ввод \ вывод в Си основан на этом подходе. Например, функция `getchar()`, строго говоря, забирает вводимый знак не с клавиатуры напрямую, а с потока[2], который по умолчанию связан с клавиатурой. Этот поток ввода обозначается как `stdin` ("standard input"). Соответственно, поток вывода носит обозначение `stdout`, и по умолчанию целью вывода назначен экран.[3]

В общем виде указание в терминале ОС на ввод из файла и вывод в файл выглядит так:

```
programma <a:file1.txt >b:file2.txt
```

Это — указание запустить программу `programma`. При этом указывается, что ввод она должна читать из файла 1 на диске `a`, а записывать вывод — в файл 2 на диске `b`.

Стрелочка влево < служит знаком перенаправления потока ввода, а стрелочка вправо > , соответственно, перенаправляет поток вывода. (Справедливо как для ОС "Windows", так и для "Unix"-подобных систем. — прим. перев.)

1 Написано авторами. (прим. перев.)

2 Если точнее — с буфера потока. Это важно, т.к. если буфер не пуст, то функция сразу читает то, что в нём содержится, без всякого ввода со стороны пользователя. (прим. перев.)

3 Эти имена — `stdin` и `stdout` — являются заранее определёнными именами стандартных потоков ввода \ вывода, и могут быть использованы при вызове функций, работающих с потоками. (прим. перев.)

Поток ошибок

Всего двух потоков — ввода и вывода — недостаточно, поскольку в вывод будут также направляться все сообщения программы об ошибках, если таковые будут. Итоговый вывод может выглядеть как мешанина. Поэтому предусмотрен третий поток — `stderr`, поток вывода для сообщений об ошибках.

Как правило, по умолчанию он направлен на экран, и часто его нельзя перенаправить. В некоторых исполнениях Си в поток ошибок можно вводить данные; в этом случае ввод всегда производится с клавиатуры, и неважно, куда направлен поток `stdin`.

Ввод и вывод применительно к файлам

Конечно, использование только этих потоков накладывает на нас определённые ограничения. Си предоставляет дополнительные возможности для действий ввода \ вывода с файлами. Кстати, в виде файлов теперь часто бывают представлены даже вошедшие в моду окна пользовательского интерфейса операционных систем.

В библиотеке предусмотрен ряд функций, выполняющих необходимое взаимодействие с операционной системой, а также подготовительные действия вроде создания буфера и указателя на него, чтобы избавить от всего этого программиста. Вот эти функции:

`fopen()`	открывает поток ввода \ вывода для указанного файла
`getc()`	читает один знак ввода с указанного потока
`putc()`	выводит один знак ввода в указанный поток
`fscanf()`	`scanf()`, читающая ввод с указанного потока
`fprintf()`	`printf()`, выполняющая вывод в указанный поток
`fclose()`	очищает буфер указанного потока и закрывает файл

В некоторых средах вместо функций `getc()` и `putc()` присутствуют функции `fgetc()` и `fputc()`. В этом случае все названия функций ввода \ вывода для файлов начинаются на букву "f". Рассмотрим каждую из них внимательнее.

Функция "fopen()"

Её можно использовать так:

```
птк = fopen(имяФ, режим);
```

`имяФ` — это указатель на строку, хранящую имя целевого файла, или сама строка (строковое значение).

`режим` — тоже указатель на строку либо сама строка, состоящая только из одной буквы:

`r` если файл требуется читать

`w` если требуется создать файл, либо в файл требуется начисто записывать данные (в этом случае, если будет обнаружен файл с этим же именем, он будет предварительно удалён)

`a` если в файл требуется записывать данные, добавляя к уже существующему содержимому

С помощью функции `fopen()` в показанном примере создаётся указатель на поток (обозначенный как `птк`, и далее как `ptk`); подготовленный поток содержит все необходимые ссыл-

ки на файл. Программисту о них думать не нужно — требуется лишь обращаться по имени указателя.

При работе в "Unix" этот указатель будет указывать на структуру, которой задан пользовательский вид данных под названием `FILE`. Очевидно, здесь необходимо, чтобы используемый компилятор поддерживал указание `typedef` — иначе возникнут неожиданности.

В других же ОСях по адресу указателя может оказаться "целое число", которое будет использовано в качестве файлового дескриптора (по сути, номера файла — прим. перев.). Вам следует обратиться к руководству по вашему компилятору \ вашей среде разработки, чтобы понять, какой вид данных для указателя `ptk` и для функции `fopen()` следует назначать в вашем случае.

Создание файла

Теперь все необходимые ссылки на файл создаются через имя потока. Следующий пример создаст файл[1] и запишет в него целые числа от 1 до 100, а рядом запишет квадраты этих чисел.

```
#include <stdio.h>


int main()
{
    FILE *ptk;
    int a;
    ptk = fopen("kvadrati.txt", "w");
    for (a = 1; a < 101; a++)
         fprintf(ptk, "%d %d\n", a, a*a);
    fclose(ptk);
    return 0;
}
```

Как видите, в моём случае `ptk` указывает на структуру вида `FILE` (как в Юниксе) — вам, возможно, потребуется изменить эту строчку для вашего случая[2].

Функция `fopen()` подготавливает файл для записи данных, и записывает адрес потока ввода в него в указатель `ptk`. Затем функция `fprintf()` записывает в файл числа. Она работает так же как и `printf()`, с тем лишь отличием, что ей требуется указывать дополнительный самый первый аргумент — целевое имя потока. Вот кусочек вывода, который попадёт в файл:

---

1 Если используемая вами среда разработки использует т.н. проекты, то файл скорее всего будет создан в расположении текущего проекта.

В строке с именем файла, которая подаётся в `fopen()`, можно также написать полный путь к файлу. Используемый в "Windows" знак косой черты \ (обратный слеш) в пути должен быть двойной — т.к. одиночный воспринимается как ввод условного сочетания (такого как `\n`). Под ГНУ/Линуксом эта косая черта смотрит в другую сторону (простой слеш), поэтому она вводится одиночной; под "Windows" также можно пользоваться ей, а не обратной. (прим. перев.)

2 Под "Windows" работает без изменений. (прим. перев.)

```
1 1
2 4
3 9
4 16
5 25
...
```

Между соседними значениями — ровно один пробел, поскольку именно так указано в управляющей строке. Я обращаю на это внимание потому, что при чтении из файла требуется в точности знать, как именно в нём записаны и разделены данные. Поэтому полезно представлять себе, как вывод будет выглядеть — или проверять это.

Следующий шаг

Теперь надо вывести данные из файла. Конечно, ничего особенного в этом нет — любая операционная система обеспечивает вывод файла на экран \ на принтер \ куда-либо ещё. Написание программы для этой задачи будет полезным упражнением для понимания работы с файлами.

Так как порядок записи в наш файл нам точно известен, и мы даже знаем количество записей в нём — напишем некую противоположность программы создания файла:

```
#include <stdio.h>
int main()
{
    FILE *ptk;
    int a, znach, znach_kv;
    ptk = fopen("kvadrati.txt", "r");
    for (a = 1; a < 101; a++)
    {
         fscanf(ptk, "%d %d", &znach, &znach_kv);
         printf("%d %d\n", znach, znach_kv);
    }
    fclose(ptk);
    return 0;
}
```

Пара замечаний. Функция `fscanf()`, как и `scanf()`, требует, чтобы на ввод ей подавались указатели. Что до использования `fclose()` в конце — в данном случае необходимостью данный шаг не является[1], поскольку не требуется очищать буфер; однако имя потока освобождается, что даёт возможность дальнейшей работы с ним.

1 Закрывать потоки после использования желательно вручную, т.к. во-первых, число одновременно открытых потоков ограничено, а во-вторых, после выполнения программы потоки будут закрыты только в случае неаварийного завершения. (прим. перев.)

Эта программа написана ради одного-единственного определённого файла. Вероятно, стоит переделать её для более широкого применения.

```
#include <stdio.h>


int main()
{
    FILE *ptk_vv, *ptk_vivod;
    char otkuda[20], kuda[20], b;
    printf("Файл для чтения: ");
    scanf("%s", otkuda);
    printf("Путь для записи файла: ");
    scanf("%s", kuda);
    ptk_vv = fopen(otkuda, "r");
    ptk_vivod = fopen(kuda, "w");
    while ((b = getc(ptk_vv)) != EOF)
        putc(b, ptk_vivod);
    fclose(ptk_vv);
    fclose(ptk_vivod);
    return 0;
}
```

Получилась программа, похожая на программы копирования файлов большинства операционных систем. Сначала она спрашивает, откуда считывать файл, и куда записать читаемые данные. Затем, до тех пор пока не встретится конец исходного файла, его содержимое побайтово записывается в получаемый файл. (Признак конца файла обозначается как `EOF`, это обозначение будет заменено обработчиком `#define` — в большинстве случаев на значение -1. Обозначение `EOF` входит в список обработчика `#define` для замены по умолчанию.)

Правда, мы будем вынуждены вводить путь для выводного файла даже в тех случаях, когда хотим вывести исходный файл только на экран. Тогда нам вообще не нужен выводной файл. Заменим

```
putc(b, ptk_vivod);
```

на

```
putc(b, stdout);
```

или вообще на

```
putchar(b);
```

Ошибки доступа к файлам

В вышеприведённом коде не предусмотрены случаи ошибок чтения \ записи. Причиной для них могут послужить как неисправности запоминающего устройства, так и более баналь-

ные вещи вроде нехватки свободного места. Также ошибка возникнет, если мы попытаемся читать несуществующий файл.

Однако в случае если возникновение ошибок не предусмотрено в коде, программа невзирая ни на что продолжит работу. В лучшем случае её работа не принесёт никакой пользы, в худшем — будут перезаписаны данные на диске.

Оставлять всё в таком виде — плохо, тем более что функция `fopen()` сообщает, удалось ли ей открыть поток чтения \ записи для файла. В случае неудачи она запишет в указатель-имя потока адрес 0.

`putc()` и `fclose()` тоже умеют сообщать об ошибках. Первая в этом случае возвращает значение `EOF`, а вторая — код ошибки, отличный от 0.

Дополненная программа будет выглядеть так:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *ptk_vv, *ptk_vivod;
    char otkuda [20], kuda[20], b;
    printf("Файл для чтения: ");
    scanf("%s", otkuda);
    printf("Путь для записи файла: ");
    scanf("%s", kuda);
    if ((ptk_vv = fopen(otkuda, "r")) == 0)
    {
        printf("Файл не найден: %s", otkuda);
        exit(0);
    }
    if ((ptk_vivod = fopen(kuda, "w")) == 0)
    {
        printf("Не могу открыть файл для записи: %s", kuda);
        exit(0);
    }
    while ((b = getc(ptk_vv)) != EOF)
        if (putc(b, ptk_vivod) == EOF)
        {
            printf("Ошибка записи на диск");
            exit(0);
        }
```

```
    fclose(ptk_vv);
    fclose(ptk_vivod);
    return 0;
}
```

Произвольное чтение \ запись файла

Наши примеры выстроены так, словно в файле записаны сплошь полезные данные. Мы не использовали пока способность жёсткого диска переключаться при необходимости между дорожками и пропускать байты.

Си и здесь предлагает библиотечную функцию, выполняющую бóльшую часть необходимой работы. Она позволяет представить файл как массив знаков ввода, и соответственно даёт возможность перемещаться по ячейкам этого массива. Вот эта функция и её запись:

```
fseek(птк, номБайта, откуда);
```

Использованы следующие обозначения:

`птк` — как и прежде — это имя потока ввода \ вывода

`номБайта` — порядковый номер байта, на который должна быть установлена головка диска перед началом чтения \ записи. Откуда отсчитывается этот байт — зависит от следующей настройки:

`откуда` — и эта настройка может принимать следующие значения:

- `0` — отсчитывать целевой байт от начала файла
- `1` — отсчитывать целевой байт от текущего положения в файле
- `2` — отсчитывать целевой байт от конца файла

Пример:

```
fseek(ptk, 200, 0);
```

Головка жёсткого диска будет перемещена так, что следующая функция `getc()` прочитает 200-й байт из файла. Следующий после этого вызов функции, например такой

```
fseek(ptk, 50, 1);
```

установит головку на 250-й байт. Также возможно и такое использование:

```
fseek(ptk, 0, 2);
```

В этом случае головка будет установлена на байт, содержащий метку конца файла. Можно осуществить дозапись в этот файл.

Вид данных "номера байта" в разных средах разный. Часто это "целое число большой величины" (`long int`). В других случаях — просто "целое число". Понятно, что вид данных для него должен умещать большие числа, на случай работы с объёмными файлами.

Перед тем как переходить дальше, следует сказать вот что. Никогда не используйте свою программу, выполняющую чтение \ запись, не отладив её работу на черновом запоминающем устройстве[1], и никогда не отлаживайте её на устройстве, хранящем полезные данные. Убедитесь несколько раз, что ваша программа с функцией `fseek()` записывает данные именно так, как должна.

1 Или в виртуальной машине. (прим. перев.)

Таблица квадратов чисел

Очередной пример будет сам по себе банален, однако позволит показать всё то важное, что я хочу, о произвольном чтении \ записи файлов. У нас на руках имеется файл `kvadrati.txt`, со списком целых чисел и их квадратов. Мы, предположительно, можем отыскать в нём квадрат нужного значения, как в таблице. Файл начинается так:

```
1 1
2 4
3 9
```

Таким образом, каждая "запись" (одна строчка) состоит из 4-х байт (число, пробел, квадрат числа, разрыв строки)[1]. Кстати, в некоторых средах разрыв строки состоит из двух знаков — собственно перевода строки и возврата каретки в начало строки (и в этом случае длина "записи" — 5 байт).[2]

Примем длину "записи" 4 байта. Тогда, чтобы найти в файле значение квадрата числа `a`, отсчитывая от первого байта, следует перейти к байту, порядковый номер которого при отсчёте составит `4*(a-1)`. Правда, уже четвёртая "запись" в файле всё сломает.

```
4 16
```

Ведь на ней длина "записи" становится больше. Справиться с этим возможно, однако простого решения не будет. Поэтому всегда следует помнить: если возможно — делайте длину всех "записей" всегда одинаковой.

Задача 1

Перепишите программу создания файла с квадратами чисел так, чтобы каждая строчка в файле имела длину 10 байт (при условии, что знак новой строки занимает 1 байт).

Пробуем ещё раз

Надеюсь, что задача 1 выполнена. Подразумевая, что это так, и что в файле `kvadrati.txt` длина всех строк одинакова, можно приступать к написанию программы поиска квадрата нужного числа.

```
#include <stdio.h>


int main()
{
    FILE *ptk_vv;
    int a, nmr_baita, znach, znach_kv;
    ptk_vv = fopen("kvadrati.txt", "r");
    printf("Исходное число, не возведённое в квадрат: ");
    scanf("%d", &a);
    nmr_baita = 10*(a - 1);
```

1 В "Unix"-подобных ОС. (прим. перев.)

2 В "Windows". (прим. перев.)

```
    fseek(ptk_vv, nmr_baita, 0);
    fscanf(ptk_vv, "%d %d", &znach, &znach_kv);
    printf("%d в квадрате равно %d", znach, znach_kv);
    fclose(ptk_vv);
    return 0;
}
```

Функция `fseek()` устанавливает головку жёсткого диска на нужное место в файле, затем `fscanf()` читает всю запись. Нам, конечно, требуется только второе значение — собственно квадрат числа, но я решил выводить оба, чтобы можно было убедиться, что найдена верная запись.

И наоборот

Теперь видно, что данный пример неполезен, поскольку сам подход к решению задачи неверен. Поясню, почему.

Во-первых, значения полей одной записи взаимосвязаны — одно является степенью другого. Значение соседнего поля можно просто высчитать, зная эту связь — читать соседнее поле не обязательно.

Легко найдутся примеры, где всё иначе — скажем, структура для учёта товара на складе, из главы 11. Связи между значениями в полях нет вообще. Я выбрал пример с квадратами чисел просто потому, что меньше пришлось стучать по клавиатуре, чтобы набрать его.

Вы скажете: "А почему бы не записать весь файл в массив, и не работать с ячейками массива? К тому же чтение будет быстрее." Стоящее замечание. Такой приём не сработает только в случае, когда объём файла превышает доступный объём оперативной памяти.

Прошу вас придержать недоверчивость и принять эти два условия: пусть наш условный файл больше доступного объёма памяти, а также нет никакой вычислимой связи между полями. Можем ли мы в этих условиях выполнить обратную задачу — вывести квадратный корень заданного числа?

Можно просматривать по порядку все записи и сравнивать, совпадёт ли число в поле с числом, введённым в программе. Это долгий последовательный поиск[1]. Более предпочтительно использование поиска делением пополам. Если вы незнакомы с этим приёмом — вот его описание:

> Взять срединную запись в файле. Если это искомая запись — поиск завершён. Если номер поля больше, чем искомый — значит, искомое поле находится в первой половине файла. Если меньше — то наоборот. В этой выбранной половине всё повторяется заново, и так до тех пор, пока искомое не будет найдено.

Исчерпывающий поиск среди примерно миллиона записей, использующий этот приём, требует не более 24-х проходов. Вот пример применения такого поиска для решения нашей задачи.

---

1 В основном для его обозначения применяется заморское словечко - "линейный поиск". "Линейный" - т.е. однонаправленный, последовательный. В качестве промежуточного решения могу предложить ещё одно слово - "ленейный", от слова "лень". Если лень придумывать умный и быстрый способ поиска - пользуйся тупым и медленным. (прим. перев.)

```
#include <stdio.h>

int main()
{
    FILE *ptk_vv;
    int ver_grn, nizh_grn, srd_zps, pole_kvd, vvod, nmr_baita,
koren;
    ver_grn = 100; nizh_grn = 1; pole_kvd = 0;
    srd_zps = 50;
    ptk_vv = fopen("kvadrati.txt", "r");
    printf("Число для извлечения кв. корня: "); scanf("%d",
&vvod);
    while (vvod != pole_kvd)
    {
        nmr_baita = 10*(srd_zps - 1);
        fseek(ptk_vv, nmr_baita, 0);
        fscanf(ptk_vv, "%d %d", &koren, &pole_kvd);
        if (pole_kvd > vvod)
            ver_grn = srd_zps - 1;
        else
            nizh_grn = srd_zps + 1;
        srd_zps = (ver_grn + nizh_grn)/2;
    }
    printf("Квадратный корень равен %d\n", koren);
    fclose(ptk_vv);
    return 0;
}
```

Он непосредственно отражает порядок поиска делением пополам. Переменные `ver_grn` и `nizh_grn` определяют границы для поиска; `srd_zps` получает значение, среднее между ними. Читается запись, соответствующая местоположению `srd_zps`, и её поле, содержащее квадрат числа (переменная `pole_kvd`), сравнивается с искомым квадратом числа (переменная `vvod`). Если значение, прочитанное из файла, больше введённого — вторая половина файла отбрасывается (`ver_grn = srd_zps - 1`). В противном случае похожим образом отбрасывается первая половина. Когда оба значения оказываются равны — цикл прерывается, и на экран выводится значение первого поля текущей записи.

Чтобы оставить только необходимые для обсуждения строчки кода, я не включил сюда проверки вывода `fopen()` и `fscanf()` на ошибки чтения \ записи. Когда я запустил программу в самый первый раз — она повисла. Я подумал, что цикл начал выполняться безостановоч-

но — и добавил в него вывод `printf()`, чтобы убедиться в этом. Но никакого вывода на экран не появилось. Только потом я понял, что на диске не было файла `kvadrati.txt`. Если бы проверки ошибок ввода \ вывода были предусмотрены — всё бы решилось гораздо быстрее. Надеюсь, вас глубоко тронул этот рассказ.

Задача 2

Есть одно условие, при котором цикл из рассмотренного примера действительно будет выполняться вечно. Что это за условие, и как от него застраховаться?

Задача 3

Наша программа непригодна для прикладного использования. Ей требуются усовершенствования.

Напишите функцию `isk_vfaile()`, которая принимает множество аргументов: имя целевого файла, количество записей в нём, объём одной записи в байтах, порядковый номер (внутри записи) первого байта поля для номера записи, длина этого поля в байтах, и искомый номер записи. Функция должна отыскивать запись с нужным номером, читая в записях соответствующее поле, и возвращать этот номер. Например:

```
a = isk_vfaile("file", 2000, 25, 7, 3, 123);
```

Здесь показан поиск записи с номером 123 в файле `file`, состоящем из 2000 записей по 25 байт. Поиск осуществляется в поле длиной 3 байта, начинающемся в 7-м байте каждой записи.

Ответы

Задача 1

Всё, что требуется — изменить настройки вывода функции `fprintf()`.

```
fprintf(ptk, "%3d %5d\n", a, a*a);
```

Таким образом число `a` займёт 3 байта, после него последует пробел, затем квадрат `a` займёт 5 байт, и ещё 1 байт[1] — это перенос строки.

Задача 2

Собственно, речь идёт о случае, когда введённое в программу значение (переменная `vvod`) не находится ни в одном поле `pole_kvd`. Очевидно, если переменные — "границы поиска" в конце концов получают одно и то же значение, это свидетельствует, что поиск не завершился успехом. Меры по противодействию этому таковы:

```
while (vvod != pole_kvd)
{
    nmr_baita = 10*(srd_zps - 1);
    fseek(ptk_vv, nmr_baita, 0);
    fscanf(ptk_vv, "%d %d", &koren, &pole_kvd);
    if (nizh_grn == ver_grn && vvod != pole_kvd)
```

1 В "Unix"-подобных ОС. (прим. перев.)

```
    {
        printf("Запись в таблице квадратов не найдена");
        exit(0);
    }
    if (pole_kvd > vvod)
        ver_grn = srd_zps - 1;
    else
        nizh_grn = srd_zps + 1;
    srd_zps = (ver_grn + nizh_grn)/2;
}
```

Могу также сказать, что в данном случае использовать цикл `do...while` было бы чуть предпочтительней: не пришлось бы давать начальные значения переменным `srd_zps` и `pole_kvd`. Моё нежелание использовать его скорее личное (вокруг себя я таких людей, кстати, тоже замечал). Я не умею писать красивые циклы `do...while`.

Не вижу в этом ничего плохого, потому как каждый пишет так, как ему удобно, и пользуется тем, что хорошо знакомо — это спасает от многих ошибок. Вам я тоже советую придерживаться своих собственных излюбленных подходов.

Задача 3

```
int isk_vfaile(im_faila, kolvo_zps, dlina_zps, nach_bait,
dlina_polya, isk_nomer)
char *im_faila;
int kolvo_zps, dlina_zps, nach_bait, dlina_polya, isk_nomer;
{
    FILE *ptk_vv;
    int ver_grn, nizh_grn, srd_zps, nmr_baita, znach_vpole;
    ver_grn = kolvo_zps; nizh_grn = 1; znach_vpole = 0;
    srd_zps = (ver_grn + nizh_grn)/2;
    if ((ptk_vv = fopen(im_faila, "r") == 0)
    {
        printf("Файл не найден: %s", im_faila);
        return 0;
    }
    while (isk_nomer != znach_vpole)
    {
        nmr_baita = dlina_zps*(srd_zps - 1);
        fseek(ptk_vv, nmr_baita, 0);
        fscanf(ptk_vv, "%s", zapis);
        izvlech(zapis, usech, nach_bait, dlina_polya);
        znach_vpole = atoi(usech);
```

```
            if (nizh_grn == ver_grn && isk_nomer != znach_vpole)
            {
                 printf("Not found");
                 return 0;
            }
            if (znach_vpole > isk_nomer)
                 ver_grn = srd_zps - 1;
            else
                 nizh_grn = srd_zps + 1;
                 srd_zps = (ver_grn + nizh_grn)/2;
       }
       fclose(ptk_vv);
       return srd_zps;
  }
```

Эта функция довольно похожа на программу нахождения квадратного корня. Существенное отличие лишь в извлечении номера поля из обрабатываемой записи.

Для простоты подразумевается использование доступного отовсюду (объявленного до `main()`) массива, под названием `zapis`, в который записывается вся читаемая запись. Также заранее подготовлен и массив `usech`, в который функция `izvlech()` из главы 5 записывает номер записи из соответствующего поля.

В качестве номера записи также может быть использовано буквенное обозначение, поскольку в функции оно преобразуется в вид данных "целое число".

ЗАДАНИЯ

1. Поиск делением пополам срабатывает только в случае, если записи строго расположены в порядке по возрастанию \ убыванию. Напишите функцию, которая бы располагала в файле записи по возрастанию, на основании любого указанного поля, при условии что оно в каждой записи содержит число.

2. Напишите функции `cht_zapis()` и `zps_zapis()`, которые соответственно читают из файлов на диске и записывают в них структуры для учёта товара на складе из главы 11. Что касается функций, описанных в той главе — они могут быть изменены, чтобы указывать не на массив, а на файл. Совместно с доведёнными до ума функциями поиска в записях и упорядочивания записей всё это может быть использовано по прямому назначению.

3. Обработка файлов с применением произвольного чтения \ записи может длиться долго, особенно если объём записей в файле велик.

Ускорить дело поможет хранение полей с номерами записей в оперативной памяти в массиве, в порядке возрастания номеров. Тогда последовательным поиском придётся обрабатывать только эти поля в этом массиве, ища номер нужной записи, который позволит нам обратиться к нужной записи на диске. Такой массив называется индексом.

Напишите функцию, которая бы создавала для файла индекс, и записывала его на диск в качестве ещё одного файла — таким образом, пока основной файл не изменится, в индекс вносить изменения не потребуется. Если хотите добиться ещё большей скорости — то пусть у записей будет несколько полей с номерами-обозначениями, чтобы для одного файла было предусмотрено несколько индексных файлов, хранящих значения этих разных полей.

ГЛАВА 13

Отладка

Какие бы вы программы ни писали — следует всегда помнить две вещи.

1. Всё, что хотя бы в теории может пойти не так — пойдёт не так.

2. Всё остальное, скорее всего, тоже.

Кошки мяукают. Собаки лают. Политики лгут. Программисты ищут ошибку, из-за которой ничего не работает.

Если к отладке не подходить продуманно — она вытянет из вас все производительные силы. Особенно это касается компилируемых программ, потому как после любого изменения конпелять их надо заново.

Во многих исполнениях Си предусмотрены собственные средства отладки. Все они разные, поэтому рассматривать их здесь мы не будем. Вместо этого поговорим в общем о том, как лучше действовать в тех или иных случаях, и рассмотрим некоторые приёмы, способные упростить при отладке жизнь.

Ошибки написания

Они встречаются у многих, только начинающих писать на Си.

1. Вместо == пишут = .

Одиночный знак равно = означает присвоение значения. Двойной == означает проверку "равно ли одно другому". Чтобы присвоить переменной "икс" значение 3, мы пишем

```
x = 3;
```

А чтобы проверить, равно ли её значение трём — пишем, например,

```
if (x == 3)
```

и т.д.

В Бейсике для обеих этих задач используется одиночный знак равно = , так что привыкшим к Бейсику придётся отвыкать. Для привыкших к Паскалю всё ещё более грустно.

Обычно ожидают, что такого рода ошибки будут обнаружены компилятором. Но в Си всё пойдёт иначе. Допустим, мы написали

```
if (x = 3)
    ...
```

"Икс" получит значение 3. Значение этого выражения равно трём, и не равно 0, следовательно его значение — "истина". Поэтому код, записанный после условия `if`, будет выполняться. Что касается выражения

```
x == 3
```

, то его значение (0 или 1) также будет вычислено, однако на значение "икс" оно никак не повлияет.

2. Пропускают разделитель действий — точку с запятой ; .

Этим знаком должно заканчиваться каждое действие. Соответственно, если точка с запятой пропущена, то два действия сливаются в одно, что может привести к занятным последствиям.

Предположим, что мы хотели написать

```
a = a + c;
b = b + d;
```

, а написали

```
a = a + c
b = b + d;
```

И тогда компилятор прочитает следующее:

```
a = a + cb = b + d;
```

Получившееся слово `cb` будет воспринято как имя переменной. Если (что наиболее вероятно) переменная с таким именем отсутствует, то появится ошибка "undeclared variable" ("необъявленная переменная"). И гораздо хуже, если такая переменная есть.

Из этого следует вывод, что если компилятор сообщает о какой-то ошибке — она может быть следствием совсем другой совершённой вами ошибки.

3. Пропускают закрывающую фигурную скобку } .

Данная ошибка похожа на предыдущую, но относится к записи составных действий. Как правило, компилятор обнаруживает эту ошибку, потому как количество открывающих и закрывающих фигурных скобок не совпадает. Но в том случае, если в составном действии фигурная скобка пропущена, а далее в коде вставлена лишняя — компилятор ничего не обнаружит. А программа будет выполняться с ошибкой.

4. Пропускают двойные кавычки " " .

В них должно быть заключено любое строковое значение. В противном случае оно будет воспринято как одно или несколько имён переменных.

5. Пропускают закрывающее сочетание комментариев */ .

Комментарии обособляются сначала сочетанием /* , а в конце сочетанием */ . Если не поставить закрывающее сочетание — вся оставшаяся после начала комментария часть программы не будет прочитана.

6. Забывают объявлять переменные.

Все переменные должны быть правильно объявлены. В противном случае компилятор сообщит об ошибке и не сможет обращаться к этим переменным.

Однако не всё, что компилятор воспринимает как необъявленные переменные, обязано быть необъявленными переменными. Компилятор может считать переменную необъявленной вследствие каких-то других ошибок.

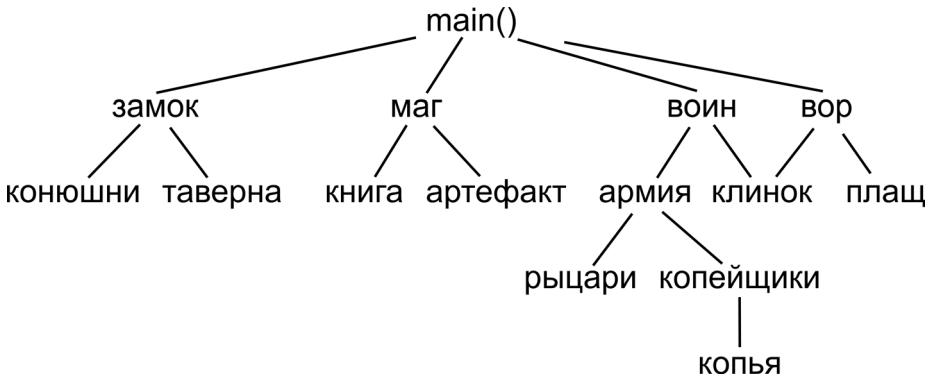
Ошибки выполнения

Выше приведены ошибки, касающиеся правил записи кода в Си и неправильного понимания обозначений. Более сложный для устранения вид ошибок проявляется только при запуске программы. Эти ошибки называются ошибками времени выполнения (т.к. происходят во время выполнения программы; англ. — "runtime error"), или просто — ошибками выполнения. Программа может иметь безупречный код с точки зрения правил записи языка, но это не значит, что она правильно заработает.

Очень трудно заранее предсказать ошибки выполнения и избежать их, однако если ваша программа чётко поделена на части, то их причины гораздо проще обнаружить и устранить.

Примером такого кода послужит программа, состоящая из нескольких небольших функ-

ций, которые могут также вызывать друг друга. Функция `main()` просто объединяет в своём теле их все. Такую программу можно представить в виде дерева, например так:



Дерево показывает порядок вызова функций.

Попытка начать отлаживать сразу всю программу (на уровне `main()`) может обернуться кошмаром. Ошибка может скрываться очень глубоко (например, на уровне функции "рыцари"). Причём последствия этой ошибки могут сильно разрастаться, если эта функция часто вызывается в теле программы.

Поэтому при отладке следует подниматься по выстроенному дереву снизу вверх, от веточек к корням. От проверки работы самых нижних в дереве функций — к проверке работы тех функций, которые их вызывают, и так далее.

Проверка работы функции

Из предыдущего раздела следует, что писать функции стоит по одной, и перед тем как каждую из них использовать — следует их проверять в отдельной проверочной \ черновой программе.

Например, мы написали функцию, вычисляющую объём коробки[1] по заданным сторонам А, Б и В.

```
int obj (a, b, v)
int a, b, v;
{
    return a*b*v;
}
```

Проверить её можно так:

```
#include <stdio.h>
int main()
{
    int g, d, e;
    g = 2; d = 3; e = 5;
    printf("Объём составляет %3d единиц", obj(g, d, e));
    return 0;
}
```

После компиляции и запуска эта программа выдаст:

1 Вместо дурацкого слова "параллелепипед". — прим. перев.

```
Объём составляет  30 единиц
```

Если необходимо убедиться в правильности вычислений — можно просто повбивать разные числа вручную, или же воспользоваться для этого библиотечными функциями (см. главу 8).

Если функция при проверке выдаёт неверное значение — следует искать ошибку в записи вычисления.

```
int obj (a, b, v)
int a, b, v;
{
    return a*b+v;
}
```

Конечно, данный пример предельно прост, но подход остаётся всегда тем же.

Проверочные включения в код

Очень часто конечный вывод программы не даёт никаких сведений о происходящем в её теле. Чтобы увидеть промежуточные значения вычислений или других действий, вводят временные проверочные участки кода.

Перед нами функция с ошибкой, задача которой — вычислить степень (`n`) целого числа `a`.

```
int stepen(a, n)
int a, n;
{
    while (n > 0)
         a *= n--;
    return a;
}
```

Для начала требуется черновая программа для проверки работы функции при её вызове.

```
#include <stdio.h>


int main()
{
    int a, n;
    a = 7; n = 4;
    printf("\nЧисло в заданной степени равно %6d", stepen(a,
n));
    return 0;
}
```

После компиляции и запуска получаем вывод:

```
Число в заданной степени равно 168
```

Что-то 168 непохоже на 2401 (чему равняется четвёртая степень семёрки). Путём долгого расследования, опроса свидетелей и дорогостоящей экспертизы приходим к выводу, что скорее всего ошибка находится в единственной строчке, производящей вычисления: `a *= n--;`. Когда я это писал — хотелось объединить отрицательное приращение `n` и последовательное увеличение значения `a`. Или что-то неправильно, или чего-то не хватает.

Можно отслеживать происходящее внутри цикла, выводя в каждом проходе значения `a` и `n`.

```
int stepen(a, n)
int a, n;
{
    while (n > 0)
    {
        printf("a = %3d n = %3d\t", a, n);
        a *= n--;
    }
    return a;
}
```

Вывод станет таким:

```
a =   7 n =   4   a =   28 n =   3   a =   84 n =   2   a = 168
n =   1
Число в заданной степени равно    168
```

Теперь наглядно видно, что ошибочно само производимое действие:

```
7*4*3*2*1 = 168
```

Перепишем в функции первые строчки следующим образом:

```
int stepen(a, n)
int a, n;
{
    n0 = n;
    while ...
    a *= n0; n--;
```

После чего обнаружим, что код не компилируется. Получаем сообщение наподобие такого:

```
"this variable was not in parameter list"
```

Всё потому, что новая переменная `n0` не объявлена. Исправим эту оплошность.

```
int stepen (a, n)
int a, n;
{
    int n0;
    n0 = n;
    while (n > 0)
    {
         printf("a = %3d n = %3d\t", a, n);
         a *= n0; n--;
    }
    return a;
}
```

Получаемый вывод:

```
a =   7 n =   4   a =  28 n =   3    a = 112 n =   2    a = 448
n =   1

Число в заданной степени равно   1792
```

Удивительно, но производится последовательное умножение на 4, а не на 7. Надо поменять

```
n0 = n;
```

на

```
n0 = a;
```

Теперь вывод приобрёл следующий вид:

```
a =   7 n =   4   a =  49 n =   3    a = 343 n =   2    a =
2401 n =   1

Число в заданной степени равно  16807
```

Как и требуется, число последовательно умножается само на себя, но происходит одно лишнее умножение. Причина — в одном лишнем проходе цикла. Поэтому заменим

```
while (n > 0)
```

на

```
while (n > 1)
```

Вследствие чего получим вывод:

```
a =   7 n =   4   a =  49 n =   3    a = 343 n =   2

Число в заданной степени равно   2401
```

Кажется, всё хорошо.

Упорнее, лучше, тщательнее

Есть бородатый анекдот про бродягу, который сидит перед грудой хвороста и по очереди зажигает спички. Те спички, которые не зажглись, он выбрасывает. Которые зажглись — задувает и кладёт обратно в коробок. Один прохожий спросил его, зачем он это делает. Ответ был таков: "Я проверяю, какие из них работают, а какие нет."

Программы подобны этим спичкам из анекдота, потому что если программа сработала один раз — это ещё не значит, что она заработает и дальше.

Проверим функцию вычисления степени числа с другими значениями. Возьмём следующие значения: 3^5, 10^3, 2^{10}, 6^1, 11^0. Проверим, правилен ли получаемый итог.

3^5	= 243	верно
10^3	= 1000	верно
2^{10}	= 1024	верно
6^1	= 6	верно
11^0	= 11	ошибка (верный ответ — 1)

Как видно, возведение в нолевую степень работает неправильно. Причина в том, как мы возвращаем значение из функции:

```
return a;
```

Требуются отдельные действия в случае, если степень (`n`) равна 0.

Задача 1

Измените функцию `stepen()` так, чтобы возведение числа в нолевую степень всегда давало 1.

Задача 2

Хороший способ работы с проверочными строчками кода — заготовить отдельную функцию, которая бы выводила текущее значение заданной переменной. Пусть у неё будет два аргумента — сообщение и собственно переменная. Предположим, что переменная "икс" имеет значение 99. Тогда такой вызов этой функции

```
otladka("x =", x);
```

должен дать такой вывод:

```
>> x = 99
```

Знаки >> тоже должны появиться в выводе, чтобы сообщение этой функции было выделено.

Напишите эту функцию.

Спящие ошибки

Как нам окончательно убедиться, что программа делает в точности то, что требуется? При ответе на этот вопрос не могу не пофилософствовать.

Представьте, что вы спросили астронома "где завтра будет восходить Солнце". Чтобы к его ответу невозможно было придраться, он скажет, что планета уже долгое время ходит кругами вокруг нашей звезды, и что все наши наблюдения позволяют предполагать, что завтра Солнце

взойдёт, во-первых, с восточной стороны, а во-вторых — там же, где оно всегда восходит в это время года; но затем он добавит, что окончательных подтверждений верности и тем более полноты наших знаний о природе нет, поэтому нельзя отрицать возможность того, что завтра произойдёт нечто, чего никто не ожидал.

По схожим причинам первые тыщапицотмильёнаф раз успешной работы программы с данными не означают, что в следующий раз программе не попадутся те самые данные, при обработке которых выявится давно затаившаяся ошибка. Порой такие ошибки не выявляются месяцами и даже годами после выпуска программы.

Пример. Джон Смит, сидя где-то в Англии со своим котелком на голове и попивая английский чай, пишет рассчётную программу для местного электроэнергетического управления. В управлении ему объяснили, что есть два вида расценок — А и Б. При способе А потребитель каждые 3 месяца платит взнос 15 долларов[1], после чего за единицу потреблённого электричества платит 4 цента. В случае расценки Б взнос отсутствует, но цена за единицу электричества — 7 центов. Исходя из этого, Джон Смит пишет функцию:

```
int scheta (edinits, a_b)
int edinits;
char a_b;
{
    if (a_b == 'A')
         return (15 + 4*edinits/100);
    else
         return (7*edinits/100);
}
```

А затем — основную программу:

```
#include <stdio.h>

int main()
{
    int edinits;
    char a_b;
    edinits = 1035
    a_b = 'B';
    printf("Девон Майлз должен %6d", scheta (edinits, a_b));
    return 0;
}
```

Надеюсь, вы понимаете, что все эти данные — 1035, способ оплаты Б, имя потребителя и др., — которые по-хорошему должны запрашиваться у пользователя программы, в нашем примере имеют такой вид для простоты.

1 Ох уж этот американский империализм. (прим. перев.)

Джон Смит проверяет эту программу на работоспособность. Всё прекрасно работает, и он, надев плащ и взяв в руки трость истинного денди, уходит от заказчика домой в сопровождении гвардейцев в красных кителях, приговаривая, что на такие крошечные программы не стоит тратить усилий лучших программистов Соединённого Королевства.

Программа безупречно работает много лет. Но в один ужасный день она вдруг выдаёт счёт на 0 долларов и 0 центов. Этого, конечно, никто не замечает — вокруг тысячи других счетов. К тому же вероятно, что заворачивание счёта в конверт делается машиной.

Получатель такого счёта, увидев его, скорее всего пожмёт плечами и выбросит, не предприняв никаких действий. Но к несчастью для него, Джон Смит написал для электроэнергетического управления ещё одну программу, которая записывает время отправки каждого счёта, и в течение 28 дней после этого ожидает подтверждения оплаты. В случае, если оно не приходит — программа печатает и отправляет потребителю последнее предупреждение.

Предположим, что потребитель и на этот раз ничего не предпринял. Та же программа, что выдаёт последнее предупреждение, по прошествии 60 дней неоплаты отправляет распоряжение об отключении неплательщика от электричества.

А что же произошло? То, что не было предусмотрено. Потребителем оказался престарелый местный житель, который воспользовался предложением туристической фирмы и зимой уехал на отдых длительностью 3 месяца. За всё это время он не потреблял электричества вообще. К тому же его рассчитывают по способу Б. Поэтому программа выдала ему нолевой счёт. Разумеется, это весьма нечастый случай.

Как только этот недочёт обнаружен, Джону Смиту не составит труда его исправить путём добавления следующих строк кода:

```
if (scheta (edinits, a_b) == 0)
    printf("Не отправлять счёт");
```

(Здесь на месте функции `printf()` можно представить любые действия, действительно предотвращающие отправку счёта.)

Я эту историю услышал на улице. Возможно, она произошла на этапе довольно раннего развития компьютеров, но не берусь судить. В любом случае это хороший пример того, как выглядят "спящие ошибки".

Ещё один совет. Не берите с потолка числа, с которыми вы проверяете работу программы. Если в программе присутствует условие — попробуйте задать число, соответствующее числу в условии, а также значения, очень близкие к нему. Например:

```
if (a < 30)
{
    те или иные действия
}
```

Проверьте такой кусок кода со значениями `a` 29.9999, 30, 30.0001. Возможно, вам больше подойдёт не условие "равно", а "больше \ меньше либо равно".

```
if (a <= 30)
{
    те или иные действия
}
```

Убедитесь, что вы проверили работу каждого участка программы, и что получаемые значения именно такие, какие должны быть.

Ответы

Задача 1

```
int stepen(a,n)
int a, n;
{
    if (n==0)
         return 1;
    else
    {
         int n0;
         n0 = a;
         while (n > 1)
         {
              a *= n0; n--;
         }
         return a;
    }
}
```

Задача 2

```
int otladka(soobsh, znach)
{
    char *soobsh;
    int znach;
    printf(">> %s %d", soobsh, znach);
    return 0;
}
```

Примечание. В задачах мы для простоты предположили, что переменные должны иметь вид данных "целое число". Обратитесь к заданию 2 ниже.

ЗАДАНИЯ

1. Впридачу к функции `otladka()` было бы неплохо получить функцию, которая бы удаляла в коде проверочные строчки после полного завершения отладки. Напишите такую функцию. Пусть она полностью читает программу, вычищает из неё соответствующие строчки, и записывает обратно в файл.

2. Измените ответ к задаче 2 так, чтобы не было нужды заранее знать вид данных переменной, чьё значение выводится на экран.

ГЛАВА 14

Арифметика с рациональными числами[1]

В большинстве языков программирования высокого уровня предлагается два основных вида данных для чисел — "целое" и "с плавающей точкой". Таким образом, дробные значения представлены только одним видом данных.

И здесь есть над чем задуматься. Вычисления значений с плавающей точкой выполняются весьма своеобразно — если только ваш компьютер не обладает сопроцессором для этих вычислений.[2] Чтобы получить представление об этом своеобразии, рассмотрим такой пример:

```
5.83 + 642.1
```

Подразумеваем, что работа производится в десятичной системе счисления. При виде данных "с плавающей точкой" эти числа представлены так:

```
1   583000
3   642100
```

числа `1` и `3` указывают положение плавающей точки от левого конца

Чтобы выполнить сложение, необходимо уравнять положение точек. То есть значение `1` увеличивается на 2, а соответствующее ему число сдвигается вправо на 2 бита.

```
3   005830
3   642100
```

Теперь можно поразрядно сложить два числа. Числа, указывающие положение точки, складывать не требуется.

```
3   647930
```

Иначе говоря, получилось 647.93 . И это верный ответ.

Однако для простого действия сложения необходимы несколько сдвигов битов и перемещений плавающей точки. Если бы число было записано в 24-х битах, дело бы могло дойти до 24 выполнений цикла "сдвиг-приращение положения точки-проверка". В повседневности же порядок выполняемых действий ещё более сложен, по некоторым техническим причинам.

Поиск решения

Как видно, высокой скорости вычислений с дробными числами ждать не приходится. Однако в "числе с плавающей точкой" могут храниться не только дробные числа, но и большие величины, к примеру такие:

```
100 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Признайтесь теперь, часто ли приходится работать с подобными числами?

Вероятно, пойдя на определённые ограничения — невозможность работы с особо крупными величинами — возможно добиться сравнительно быстрого выполнения вычислений с дробными значениями.

---

1 Целые числа, а также те, которые можно записать в виде обыкновенной дроби. (прим. перев.)

2 Приблизительно со времён процессоров архитектуры 486 эта штука добавляется в центральные процессоры. (прим. перев.)

Несколько путей есть. Можно было бы закрепить точку в строго определённом положении, для любых чисел, и уже с этим условием производить с ними вычисления. Такой подход на ранних компьютерах имел место, но сейчас его почти не встретишь.

Другой подход предложил Бертольд Хорн (см. журнал "Software — Practice and Experience", номер 2 за 1978 год). Подход заключается в том, чтобы представить число в виде обыкновенной дроби, в числителе и знаменателе которой — целые числа. Таким образом 3.2 можно представить как 32/10, или же 16/5 .

Понятие "рациональная арифметика" происходит от слова "ratio" (соотношение), т.к. число представляется в виде соотношения двух других чисел. В дальнейшем разговоре любое число А мы можем представить как соотношение двух других чисел `а/а'`.

Рассмотрим применение этого подхода на четырёх основных арифметических действиях с числами А и Б.

1) Сложение

```
А + Б   = а/а' + б/б' =
        = (аб' + а'б)/(а'б')
```

Полученное выражение также является соотношением, или обыкновенной дробью.

Число В представим в виде дроби в/в' . Тогда, если В = А + Б, то в = (аб' + а'б) , и в' = а'б' . Здесь намечается загвоздка — ведь для записи значения произведения (например `аб'`) потребуется вдвое больше битов, чем для записи одного числа — то есть для записи чисел `в` и `в'`. Не возникнет ли вопрос переполнения битов?[1]

2) Вычитание

```
А - Б   = а/а' - б/б' =
        = а/а' + ( - б)/б'
3/2 - 5/7 = 3/2 + (-5)/7   пример
```

Необходимо лишь слегка преобразовать `б` и воспользоваться функцией сложения. По сути ничего нового.

3) Умножение

```
А * Б   = а/а' * б/б' =
        = (а * б)/(а' * б')
3/2 * 5/7 = (3 * 5)/(2 * 7)     пример
```

Как и в случае сложения, итоговое выражение является обыкновенной дробью, и точно так же встаёт вопрос переполнения битов.

4) Деление

```
А / Б   = (а/а')/(б/б') =
        = (а * б')/(а' * б)
(3/2) / (5/7) = (3*7) / (2*5)   пример
```

---

1 Переполнение битов, арифметическое переполнение ("arithmetical overflow", "integer overflow") — случай, когда двоичное значение не умещается в отведённый для него объём битов. (прим. перев.)

На удивление всё просто, хотя обычно деление доставляет много головной боли — требует множества действий и не отличается скоростью.

Переполнение битов

Вопрос переполнения битов надо принять во внимание как можно раньше. Рассмотрим его на примере, когда для записи значений чисел мы отвели 6-разрядные поля; возьмём числа 3.1 и 1.8 .

```
    31        10
3.1 011111    001010


    18        10
1.8 010010    001010
```

Используем для этих двух чисел рассмотренный выше способ сложения.

```
31/10   + 18/10 =


31*10   + 18*10   (000111101010)     первую часть
          /                          делим на
          10*10   (000001100100)     вторую
```

Получаем значения произведений, содержащиеся в полях по 12 бит. Имеем выражение 490/100, что соответствует правильному итогу сложения. Но нам необходимо записать итог, поместив его в 6 бит.

Попробуем сдвигать биты влево, до тех пор, пока в одном из значений два первых бита не начнут отличаться (тем самым отбросим незначимые ноли с левого края). Затем возьмём первые 6 бит слева. Получится следующее:

```
011110  (30)
/
000110  (6)
```

Деление первого на второе даёт 5. Неудивительно, поскольку 6-ти бит очевидно недостаточно для записи нужных нам значений. (Этот вопрос будет решён позднее. — прим. перев.)

Но если мы ещё раз взглянем на происходящее, то увидим, что в нашем примере знаменатели чисел одинаковы; можно было просто сложить числители, не производя лишних действий. Поэтому стоило бы ввести проверку на одинаковость знаменателей. Кроме того, второе число 18/10 сократимо. Если мы запишем его как 9/5 и произведём сложение, то получим:

```
000011110101      (245)
/
000000110010      (50)
```

Выборка 6-ти значимых битов с левого края даст то же самое, что и в прошлый раз! Сами же значения сдвинулись вправо на один разряд, то есть стали в 2 раза меньше.

А вот не может ли значение выражения `аб' + а'б` превысить объём в 12 бит?

Не может. При объёме в 6 бит наибольшее хранимое значение равно 011111 , или 1F в 16тиричном виде[1]. Поэтому предельное значение составит (1F * 1F) + (1F * 1F) = 1922 = 011110000010 , что вполне укладывается в 12 бит. Затруднения бы возникли в случае, если бы все числа были отрицательными — но делители-знаменатели `а'` и `б'` всегда будут положительны. Знак будут иметь только `а` и `б`.

Таким образом мы убедились, что при вычислениях необходимых выражений переполнения битов не будет.

Приступая к воплощению

Направим теперь свои силы на исполнение задуманного. Каждую часть рационального числа (числитель и знаменатель) можно хранить как "целое число". Создадим произвольный вид данных `rtsnln`.

```
typedef struct
{
    int chislit;
    int znamenat;
} rtsnln;
```

Теперь мы можем создавать рациональные числа и указатели на них, например так:

```
rtsnln a, *u;
u = &a;
```

Охват возможных значений в них, конечно же, зависит от объёма "целого числа". При объёме в 16 бит можно представить рациональные числа от -32768 до 32767, при условии что знаменатель будет равен единице. Самое же близкое к нолю возможное рациональное число будет равно 0.00003052 (1 / 32767). Получаемая точность — 4 достоверных знака после запятой (без учёта незначимых нолей).

Для многих целей этого достаточно, но с 32 битами она возрастает до 9 достоверных знаков; охват значений становится чуть более 2-х миллиардов, как с плюсом так и с минусом. Это весьма неплохие показатели. Но вскоре мы увидим, что не всё так прекрасно, как хотелось бы.

Функции

Какие нам понадобятся функции? Помимо четырёх уже рассмотренных арифметических действий также потребуется средство приведения чисел к рациональному виду и наоборот. Проще всего будет, если исходные числа будут содержаться в строковых значениях.

Так как рациональное число будет содержаться в структуре, мы можем передавать в функцию \ из функции только указатели на ячейки структуры. Среди аргументов функции должен быть указатель на полученный итог вычислений. Поэтому в качестве возвращаемого функцией значения мы можем избрать что-нибудь другое — например, знак полученного итога (плюс или минус).

Таким образом у нас получается следующее (`per`, `vtor` и `itg` — указатели на структуры вида `rtsnln`):

---

1 Используемый вид данных подразумевает наличие или отсутствие у числа знака положительности \ отрицательности. Отсюда и не используемый для хранения собственно числа старший бит. (прим. перев.)

`slozhen_rts (per, vtor, itg)` работает как `itg = *per + *vtor`

`vichit_rts (per, vtor, itg)` работает как `itg = *per - *vtor`

`umnozh_rts (per, vtor, itg)` работает как `itg = *per * *vtor`

`delen_rts (per, vtor, itg)` работает как `itg = *per / *vtor`

При этом во всех случаях, если значение на которое указывает `itg` отрицательно — функции возвращают -1 ; если значение равно 0 — возвращают 0; и если оно положительно — возвращают 1. Функции преобразования данных будут такими:

`strk_vrch(strk, rch)` преобразует строковое значение в рациональное число

`rch_vstrk(rch, strk)` преобразует рациональное число в строковое значение

(`strk` и `rch` — указатели)

Только удвоенная точность

Прежде чем переходить к написанию кода, надо решить ещё один вопрос. Он связан с тем, что в ходе вычислений, как мы выяснили ранее, появляются двоичные значения удвоенной длины, требующие удвоенного объёма памяти для хранения (например `аб'`).

Решается этот вопрос несложно, но только при условии, что используемый компилятор поддерживает вид данных "целое число большой величины" (`long int`), и объём этого вида данных в два раза превышает объём обычного "целого числа"[1]. Было бы здорово сделать так, чтобы наш код работал даже при отсутствии этих условий.

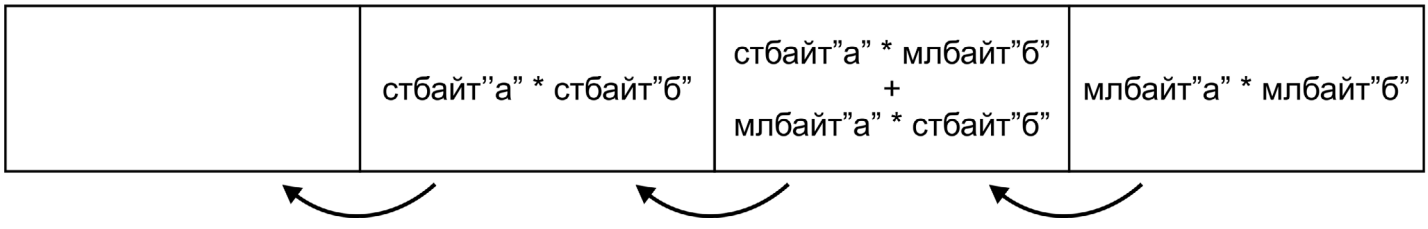
Для начала рассмотрим довольно частый случай, когда "целое число" имеет объём 16 бит, "знак ввода" — 8 бит, а "целое число большой величины" — либо не предусмотрено, либо также имеет объём 16 бит. Если мы перемножим два "целых числа", то объём для записи итогового значения будет всё так же ограничен 16-ю битами; вследствие этого могут быть записаны только младшие 16 бит полученного значения. Для нас это неприемлемо.

Тогда представим наши числа в двоичном виде, разделив их на байты:

число "а" `стбайт"а" млбайт"а"`

число "б" `стбайт"б" млбайт"б"`

Каждый байт мы можем выразить через число вида "знак ввода". 32-битное число, таким образом, мы можем представить четырьмя значениями-"знаками ввода", каждое из которых означает один байт. Вычислить произведение чисел А и Б мы можем, выполнив сложение в столбик следующих значений:



1 Да, бывает, что якобы "двойной" объём на самом деле не двойной, а такой же как у обычного "целого числа". О разности в исполнениях подвидов `short` и `long` говорилось в главе 6. (прим. перев.)

(байты расположены по старшинству слева направо, т.е. 1й байт — самый старший.

Стрелки означают, что в каждом байте остаются только младшие 8 бит от получившегося значения. Излишек же, как при любом вычислении в столбик, переносится в более старший разряд и увеличивает значение в нём)

Теперь для хранения промежуточных значений (вычисляемых для каждого байта) можно использовать "целые числа" (2 байта / 16 бит). А значения для записи в "знаки ввода" из них извлекать так: младшие 8 бит извлечь при помощи битовой маски, а после этого сдвинуть значение на 8 бит вправо, чтобы получить старший байт.

Напишем функцию `stolbik()`, которая перемножает вышеописанным способом числа А и Б, и записывает итог в массив "знаков ввода", поданный в функцию с помощью указателя.

```
int stolbik(a, b, znvv)
int a, b;
char *znvv;
{
    char mas1 [4], mas2 [4];
    int vspm, a_mlbait, a_stbait, b_mlbait, b_stbait, znak =
1;
    if (a<0)
    {
         znak *= -1;
         a *= -1;
    }
    if(b < 0)
    {
         znak *= -1;
         b *= -1;
    }
    a_mlbait = a; b_mlbait = b;
    a_stbait = a >> 8; b_stbait = b >> 8;
    vspm = a_mlbait * b_mlbait;
    mas1[0] = vspm;
    mas1[1] = vspm >> 8;
    mas1[2] = mas1[3] = 0;
    vspm = a_stbait * b_mlbait;
    mas2[0] = mas2[3] = 0;
    mas2[1] = vspm;
```

```
    mas2[2] = vspm >> 8;
    slozhit(mas1, mas2, znvv);
    vspm = a_mlbait * b_stbait;
    mas1[0] = mas1[3] = 0;
    mas1[1] = vspm;
    mas1[2] = vspm >> 8;
    slozhit(mas1, znvv, znvv);
    vspm = a_stbait * b_stbait;
    mas1[0] = mas1[1] = 0;
    mas1[2] = vspm;
    mas1[3] = vspm >> 8;
    slozhit(mas1, znvv, znvv)
    if (znak < 0)
        otrits(znvv);
    return 0;
}
```

В начале выполнения функции ожидается, что `а` и `б` положительны. Если это не так — у них меняется знак на противоположный; при этом в переменной `znak` также меняется знак, чтобы она показывала, будет ли итоговое значение отрицательным.

Затем остаётся лишь образовать в массивах части итогового числа, и сложить их в едином массиве. Для этого я использовал отдельную функцию `slozhit()`; кстати, она нам ещё пригодится в дальнейшем, в случае со сложением чисел.

В конце, если переменная `znak` показывает такую необходимость, требуется придать значению нужный знак. Но не только. Предельное положительное и предельное отрицательное значения используемого вида данных отличаются по модулю на единицу. Поэтому преобразование положительного значения в отрицательное выполняется отдельной функцией `otrits()`. Она обращает знак каждого байта в массиве, а также использует функцию `slozhit()`, чтобы прибавить к младшему байту 1.

```
int otrits(a)
char a[];
{
    int scht;
    char odin[4];
    odin[0] = 1; odin[1] = odin[2] = odin[3] = 0;
    for (scht = 0; scht < 4; scht++)
        a[scht] = -a[scht];
    slozhit(a, odin, a);
```

```
    return 0;
}
```

В функции `slozhit()` ничего незнакомого нет:

```
int slozhit(a, b, v)
char a[], b[], v[];
{
    unsigned vspm = 0;
    int scht;
    for (scht = 0; scht < 4; scht++)
    {
        vspm >>= 8;
        vspm = vspm + a[scht] + b[scht];
        v[scht] = vspm;
    }
    return 0;
}
```

Требуется ещё одна функция для работы с этими 4-байтовыми массивами; полученные 4-байтовые числитель и знаменатель она должна преобразовать в структуру вида `rtsnln`. Назовём указатель на массив с числителем `ch`, указатель на массив со знаменателем — `zn`, и указатель на целевую структуру — `itg`.

```
int msv_vrats(ch, zn, itg)
char *ch, *zn;
rtsnln *itg;
{
    while (star_bit(ch + 3) ^ sosedn_bit(ch + 3) == 0
        && star_bit(zn + 3) ^ sosedn_bit(zn + 3) == 0)
    {
        sdv_vlevo (ch);
        sdv_vlevo (zn);
    }
    itg -> chislit = *(ch + 3) << 8 + *(ch + 2);
    itg -> znamenat = *(zn + 3) << 8 + *(zn + 2);
    return 0;
}
```

И в числителе, и в знаменателе сравниваются два старших (соседних) бита[1]. Если они одинаковые (как 00 , так и 11) — сдвигаем биты влево, на 1 бит. Когда они начнут отличаться — это будет означать, что незначимых битов больше нет; поэтому можно записать старшие 16 бит каждого значения в структуру, соответственно в "числитель" и в "знаменатель".

Видим, что осталось написать функции `star_bit()`, `sosedn_bit()` и `sdv_vlevo()`.

```
int star_bit(a)
char *a;
{
    return ( *a & 0x80 ? 1 : 0 );
}
```

(0x80 — это 128 в десятичном виде, или 1000 000 в двоичном. — прим. перев.)

Таким образом, эта функция возвращает 1, если самый старший бит в массиве равен 1, и в остальных случаях возвращает 0.

Так же работает и `sosedn_bit()`:

```
int sosedn_bit(a)
char *a;
{
    return( *a & 0x40 ? 1 : 0 );
}
```

А вот функция сдвига влево:

```
int sdv_vlevo (a)
char *a;
{
    int scht;
    a += 3;
    for (scht = 0; scht < 4; scht++)
    {
        *a <<= 1;
        if (scht == 3)
            break;
        *a += star_bit(a - 1);
        a--;
    }
    return 0;
}
```

1 Выражение ...+ 3 нужно потому, что массив состоит из 4-х байтов (знаков ввода), и 4-я ячейка содержит старший байт; следовательно, надо обращаться к 4-й ячейке. (прим. перев.)

Первым делом, очевидно, сдвигается влево старший байт; при этом, опять же очевидно, самый младший его бит заполняется нолём. При сдвигании влево всех 4-х байтов в этот заполнившийся нолём разряд должен попасть самый старший бит следующего байта.

Значение этого бита читается с помощью функции `star_bit()`. Всё это повторяется 4 раза, т.е. для каждого байта; в последнем случае не требуется читать старший бит следующего байта, поэтому на половине прохода срабатывает прерывание цикла.

Что касается переносимости

Все эти функции были заранее рассмотрены перед созданием основных функций потому, что они непереносимы. Конечно, особого труда не составит сделать так, чтобы они обрабатывали массивы с А байтами, и чтобы байты содержали Б бит (и затем предоставить возможность определять величины А и Б с помощью предобработчика `#define`). Но вопрос — стоит ли оно того.

Например, если компилятор поддерживает настоящее "целое число большой величины" (`long int`), вся эта затея с массивами не нужна вообще. Так или иначе, за бóльшую переносимость скорее всего будет заплачено скоростью выполнения; а нам следует не забывать, что именно скорость выполнения — конечная цель всего, что здесь происходит.

Поэтому более предпочтительно будет обратиться к разработчику компилятора для целевой системы с просьбой создать такие же функции для неё, и предоставить ему исходный код, над которым мы здесь трудимся. Да, кстати, код надо продолжать писать.

```
int slozhen_rts(per, vtor, itg)
rtsnln *per, *vtor, *itg;
{
    char m1[4], m2[4], m3[4], m4[4];
    stolbik(per -> chislit, vtor -> znamenat, m1);
    stolbik(per -> znamenat, vtor -> chislit, m2);
    stolbik(per -> znamenat, vtor -> znamenat, m3);
    slozhit(m1, m2, m4);
    msv_vrats(m4, m3, itg);
    return (itg -> chislit ? itg -> chislit/abs(itg ->
chislit) : 0);
}
```

Собственно, здесь воплощается рассмотренный ранее способ сложить два рациональных числа. Обратить внимание следует на последнюю строчку, которая возвращает знак получившегося значения (-1, 0 или 1 для соответственно отрицательного, нолевого и положительного значений).

Числитель проверяется на равенство нолю; если равен — то возвращается 0, в противном случае числитель делится нацело на самого себя. Конечно же, получится либо 1, либо -1 , в зависимости от знака числителя. Знаменатель всегда остаётся положительным.

Да кому нужно ваше программирование

А точно остаётся? В начале главы я смело это утверждал, поскольку мы можем предусмотреть это в функции `strk_vrch()`, а функция `slozhen_rts()` образует знаменатель уже из

готовых неотрицательных значений. Так же будет и в функциях `vichit_rts()` и `umnozh_rts()`.

Но функция деления `delen_rts()` встаёт особняком, ведь внутри неё знаменатель рационального числа получается путём умножения знаменателя на числитель — который может быть отрицательным.

Но унывать ещё рано. Требуется лишь доработать функцию `msv_vrats()`, чтобы она проверяла знаменатель на отрицательность и в случае обнаружения оной меняла бы знак у полей `chislit` и `znamenat`.

Задача 1

Внесите означенные изменения в функцию `msv_vrats()`.

Вычитание

Тут всё просто:

```
int vichit_rts(per, vtor, itg)
rtsnln *per, *vtor, *itg;
{
    rtsnln dubl;
    dubl.chislit = -(vtor -> chislit);
    dubl.znamenat = vtor -> znamenat;
    return slozhen_rts(per, &dubl, itg);
}
```

Числитель вычитаемого получает знак "минус", после чего вызывается функция рационального сложения. Заметьте только, что функция работает с дублями исходных значений, поскольку в противном случае сами исходные значения в отведённых им ячейках памяти будут изменены. Как мы помним, это обусловлено тем, что мы обращаемся к ним через указатели.

Умножение

Просто выполняем его по порядку, предложенному ранее.

```
int umnozh_rts(per, vtor, itg)
rtsnln *per, *vtor, *itg;
{
    char *m1, *m2;
    stolbik(per -> chislit, vtor -> chislit, m1);
    stolbik(per -> znamenat, vtor -> znamenat, m2);
    msv_vrats(m1, m2, itg);
    return (itg -> chislit : itg -> chislit/abs(itg ->
chislit) ? 0);
}
```

Деление

И здесь поступаем ровно так же.

```
int delen_rts(per, vtor, itg)
rtsnln per, vtor, itg;
{
    char *m1, *m2;
    stolbik(per -> chislit, vtor -> znamenat, m1);
    stolbik(per -> znamenat, vtor -> chislit, m2);
    msv_vrats(m1, m2, itg);
    return (itg -> chislit : itg -> chislit/abs(itg ->
chislit) ? 0);
}
```

Преобразование туда-сюда

Осталось только позаботиться о приведении чисел к рациональному виду и обратно. Напишем функцию `strk_vrch()`:

```
int strk_vrch(strk, rch)
char *strk;
rtsnln *rch;
{
    char dubl [10], *pc, *s;
    strcpy(dubl, strk);
    rch -> znamenat = 1;
    pc = dubl;
    s = pc + strlen(pc) - 1;
    while(*s != '.')
    {
         rch -> znamenat *= 10;
         s--;
    }
    *s = '\0';
    strcat(dubl, s+1);
    rch -> chislit = atoi(dubl);
    return 0;
}
```

Создаётся дубль входного строкового значения, чтобы исходное значение оставить нетронутым. В знаменатель будущего рационального числа записывается 1, указателю `s` даётся такой адрес, чтобы он указывал на самый конец строкового значения (дубля). Затем от этого

положения влево ищется точка, при этом с каждым шагом, пока она не будет найдена, знаменатель умножается на 10. Затем мы объединяем часть строки до точки с частью строки после точки, полученное обрабатываем функцией `atoi()` и записываем выданное значение в числитель.

Понятно, что необходимо наличие во входной строке точки. Также записанное в ней число не должно превышать 32767 , что обусловлено разрядностями видов данных, с которыми мы работаем.

Теперь возьмёмся за `rch_vstrk()`.

```
int rch_vstrk(rch, strk)
rtsnln *rch;
char *strk;
{
    rtsnln dubl;
    int scht;
    *strk = ' ';
    dubl.chislit = rch -> chislit;
    dubl.znamenat = rch -> znamenat;
    if (dubl.chislit < 0)
    {
         dubl.chislit = -dubl.chislit;
         *strk = '-';
    }
    sprintf(strk + 1, "%5d.", dubl.chislit / dubl.znamenat);
    dubl.chislit %= dubl.znamenat;
    strk = strk + strlen(strk);
    for (scht = 1; scht < 5; scht++)
    {
         while (dubl.chislit > 3276)
         {
              dubl.chislit >>= 1;
              dubl.znamenat >>= 1;
         }
         dubl.chislit *= 10;
         sprintf(strk, "%1d", dubl.chislit / dubl.znamenat);
         strk++;
         dubl.chislit %= dubl.znamenat;
    }
    return 0;
}
```

Сперва делается дубль рационального числа, проверяется на отрицательность и при необходимости делается положительным. При этом в выводное строковое значение соответственно записывается либо пробел, либо минус. Далее туда записывается целая часть числа,

полученная путём обычного деления числителя на знаменатель, и точка. Затем мы берём остаток от такого деления, умножаем числитель на 10 и снова повторяем это деление, чтобы получить цифру для записи в очередной разряд после запятой. По сути мы занимаемся делением в столбик.

К сожалению, при умножении числителя на 10 может возникнуть переполнение битов; от этого нас предохраняет небольшой цикл `while()`, который, в случае если числитель превышает одну десятую от 32767 , пошагово уменьшает числитель и знаменатель вдвое до тех пор, пока числитель не станет меньше этого значения.

Внеочередная пичалька

Испытайте работу всего нами созданного. В большинстве случаев всё будет работать ожидаемо хорошо. Однако в остальных случаях точность ответов, внезапно, оставит желать лучшего.

Задача 2

Постарайтесь определить, почему это происходит. (Намёк: дело не в ошибке. В не совсем ошибке.) Когда определите — возможно, вам удастся также и найти решение.

Ответы

Задача 1

Всё довольно заурядно. Вначале с помощью функции `star_bit()` проверяется знак знаменателя. Если он окажется отрицательным — функция `otrits()` меняет знак и у числителя, и у знаменателя. Это можно сделать где угодно, главное — до того, как данные будут записаны в структуру рационального числа; однако если это сделать в самом начале, то цикл `while` будет проще, поскольку мы уже знаем, что старший бит в знаменателе не может иметь значение 1.

```
int msv_vrats(ch, zn, itg)
char *ch, *zn;
rtsnln *itg;
{
    if (star_bit(zn + 3))
    {
         otrits (zn);
         otrits (ch);
    }
    while (star_bit(ch + 3) ^ sosedn_bit(ch + 3) == 0 &&
!(star_bit(zn + 3)))
    {
         sdv_vlevo (ch);
         sdv_vlevo (zn);
    }
    itg -> chislit = *(ch + 3) << 8 + *(ch + 2);
    itg -> znamenat = *(zn + 3) << 8 + *(zn + 2);
    return 0;
}
```

Задача 2

Неплохо бы заглянуть в промежуточные итоги вычислений в рациональном виде, а также в 4-байтовые массивы. Для этого можно создать несколько дополнительных выводов наподобие такого:

```
printf("числитель в itg: %x знаменатель в itg: %x\n", itg ->
chislit, itg -> znamenat);
```

Удобно выводить значения в 16тиричном виде, поскольку это короче и нагляднее. Вы обнаружите, что значения, где значимыми являются более 30 битов, обрабатываются ровно как и задумано. С другой стороны можно будет увидеть, что неточности в вычислениях появляются, если значение знаменателя невелико. При наличии 10 значимых бит его величина — около 1000 , и погрешность вычислений составит примерно 0.1% . При наличии 5 бит (величина знаменателя около 30) погрешность будет уже весьма ощутима, примерно 3%.

ЗАДАНИЯ

Как же с этим быть? У меня несколько предложений.

1. Удвоить объём памяти для хранения данных. Либо вам понадобится процессор, способный выполнять 32-разрядные вычисления, либо придётся писать дополнительный код, который ощутимо замедлит выполнение.

2. Повысить точность значений, предназначенных для записи в структуру `rtsnln`, не усекая их сразу до 16 значимых бит, а обрабатывая подобно числам с точкой, и только самым конечным действием приводить значение к 16-битному виду. Однако такой подход ну очень близко подходит к обычной работе с "числом с плавающей точкой".

3. В конце вычисления вообще не преобразовывать 4-байтовые массивы в структуру `rtsnln`, а сразу переводить их в итоговое строковое значение, отдельной заготовленной функцией. Но тогда встанет вопрос деления в столбик (см. создание функции `rch_vstrk()`). Скорее всего, решение получится громоздким и медленным.

4. Определить область значений, для которых погрешности вычисления будут наименьшими, и хранить значения рациональных чисел по возможности в этой области. Где-то в сторонке потребуется хранить дополнительные множители.

Послесловие

Изложение в этой главе соответствует без прикрас тому, как я сам занимался решением поставленной задачи. Ошибки в начале главы (например, утверждение о всегда положительном знаменателе) действительно были, и только позже я их обнаружил.

Я не стал это исправлять, поскольку так можно было нагляднее показать преимущества сборного (или, если не православно, модульного) подхода к разработке программ. Вопрос положительности знаменателя свёлся к внесению изменений в отдельную функцию. Если бы программа не содержала собственноручно написанных функций, всё было бы далеко не так просто.

Это было бы не так важно, если бы разработчики не совершали ошибок. Но увы, эти ошибки будут всегда. На мой взгляд, многие авторы слишком усердны в желании представить взору читателя безупречный код; они умалчивают о подготовке и переделках, которые им довелось совершить, а между тем из этого можно было бы извлечь немало пользы.

ГЛАВА 15

Воплощение "черепашьей графики"...

ВАЖНОЕ ПРИМЕЧАНИЕ. В этой и следующей главе авторы используют сторонние функции рисования, добавляемые в некоторых сторонних исполнениях Си, таких как Борланд. В стандартном языке Си функций графики не предусмотрено. Поэтому эти главы следовало бы рассматривать как чисто теоретические; возможно, стоило бы вообще их выбросить, а возможно что стоит и оставить. (прим. перев.)

"Черепашью графику" в 1960-х создал Сеймур Паперт, как часть своего же обучающего языка программирования Лого. Суть подхода заключается в том, что по экрану движется воображаемая рисующая точка ("черепаха"), способная двигаться в заданных направлениях с заданной длиной пути, и при этом — либо рисовать (оставлять след), либо нет. "Черепахой" также может быть робот, рисующий на каком-либо запечатываемом материале.

"Черепаха" движется в смешанной системе координат, сочетающей прямоугольную и полярную.

В этой главе мы подробно разберём создание ряда функций для "черепашьей графики", использующих только целочисленную арифметику — поскольку некоторые компиляторы Си поддерживают графику высокого разрешения, но не поддерживают "число с плавающей точкой". Позднее немного поговорим о том, как это исправить и получить более высокую точность.

Ворочая черепахами

Нам потребуются следующие простые указания:

`ne_sled` Выключить режим рисования (оставления следа)

`sled` Включить режим рисования

`dovern` Довернуть "черепаху" против часовой стрелки на указанное количество градусов

`na_ugol` Повернуть "черепаху" на угол против часовой стрелки, отсчитываемый от положения "стрелки на 3 часа"

`pomestit` Поместить "черепаху" в положение, заданное координатами в прямоугольной системе

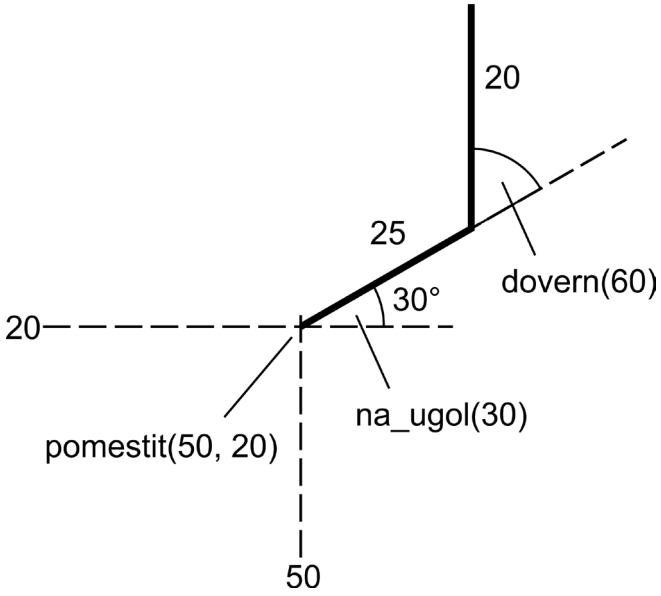
`dvigat` Двигать "черепаху" на заданную длину пути в её текущем направлении

Например, такая последовательность указаний

```
pomestit(50, 20);
sled();
na_ugol(30);
dvigat(25);
dovern(60);
dvigat(20);
```

выдаст след, показанный далее на рисунке. Если вы знакомы с языком Лого, то заметили,

что названия наших функций не совпадают с теми, что используются в этом языке. Причины — во-первых, стандарта на эти названия нет, а во-вторых я поступил именно так, чтобы уменьшить количество функций. В Лого есть функции `LEFT` и `RIGHT` — соответственно поворот против и по часовой стрелке, и их мы заменили одной функцией `dovern()`.



В любом случае уже сейчас можно ожидать, что эти функции дадут нам широкие возможности рисования на экране.

А загвоздка вот где

В наше время[1] среднее разрешение экрана домашнего компьютера — 250x200 пикселей. В то же время 2х-байтное "целое число" даёт возможные значения от -32768 до 32767[2]. Координаты нашего маленького экрана вполне укладываются и в 8 бит, поэтому у нас на руках — размах представления чисел, вдвое бóльший от требуемого. Надо использовать его с умом.

Представьте, что мы храним не действительные значения координат на экране, а увеличенные в 256 раз. Если точка в середине экрана имеет координаты 0,0 , а координаты лежат в пределах от -125 до 125, то хранимые нами значения — от -32000 до 32000 с шагом 256. Действительные значения понадобятся нам лишь в случае, когда требуется нарисовать точку (закрасить один пиксель); чтобы получить действительное значение, требуется увеличенное в 256 раз значение сдвинуть вправо на 8 бит.

Таким образом мы получили бóльшую в 2,56 раза точность, чем дала бы точность до двух знаков после запятой, и притом — в целых числах, что исключает необходимость округления и возникающие при том погрешности.

Как будет видно далее, значения синусов и косинусов углов мы также будем умножать на 256; ведь мы будем извлекать из этих значений целую часть, и это умножение даст нам такое же повышение точности. Подытожим правила, которых будем придерживаться:

1. Координаты экрана хранить в значениях, превышающих действительные в 256 раз.

2. Величину пути, который должна пройти "черепаха", хранить в действительных значениях.

3. Выделять целую часть не из синуса А, а из (256 * синусА). То же самое касается косинуса.

---

1 1987 год. (прим. перев.)

2 Для 32-разрядных архитектур объём "целого числа" составляет 4 байта. (прим. перев.)

Расчищаем дорогу

Пункт 3 озадачивает больше других. Самое простое (и не самое красивое) решение для него — подготовить массив требуемых целочисленных значений синуса для целочисленных углов от 0 до 90 градусов. Это решение рассмотрим первым, а более изящные — позднее. В нижеследующей таблице эти значения приведены.

Значения выражения "sin(A) * 256", округлённые до целого

угол A	син(A)*256	угол A	син(A)*256	угол A	син(A)*256
0	0	30	128	60	222
1	4	31	132	61	224
2	9	32	136	62	226
3	13	33	139	63	228
4	18	34	143	64	230
5	22	35	147	65	232
6	27	36	150	66	234
7	31	37	154	67	236
8	36	38	158	68	237
9	40	39	161	69	239
10	44	40	165	70	241
11	49	41	168	71	242
12	53	42	171	72	243
13	58	43	175	73	245
14	62	44	178	74	246
15	66	45	181	75	247
16	71	46	184	76	248
17	75	47	187	77	249
18	79	48	190	78	250
19	83	49	193	79	251
20	88	50	196	80	252
21	92	51	199	81	253
22	96	52	202	82	254
23	100	53	204	83	254
24	104	54	207	84	255
25	108	55	210	85	255
26	112	56	212	86	255
27	116	57	215	87	256
28	120	58	217	88	256
29	124	59	219	89	256

Всем функциям понадобится доступ к этим данным, поэтому массив должен быть объявлен до тела функции `main()`. Заодно вспомним, что нескольким функциям также понадобятся данные о положении "черепахи", о её направлении и о том, оставляет ли она след. Поэтому до начала `main()` объявляем следующее:

```
int s[91], polozhen_x, polozhen_y, napravl, sled;
```

s — это массив со значениями синуса угла, polozhen_x и polozhen_y — текущие координаты "черепахи" в прямоугольной плоскости, napravl — содержит её направление в виде угла. Наконец, sled содержит значение 1, если включен режим рисования, и 0 — если он выключен.

Также потребуется функция obnulit(), которая бы помещала "черепаху" в середину экрана, давала ей направление 0° и включала режим рисования. А также она должна подготавливать массив s.

Что касается s, то его можно создать нудным и долгим способом:

```
s[0] = 0;
...
s[90] = 256;
```

Получится 91 строка кода. Или, возможно, ваш компилятор поддерживает такое объявление:

```
int s[91] = {0, ... , 256};
```

Но это всё долго. Сдаётся мне, что у вас под рукой имеется интерпретатор Бейсика; поэтому я покажу, как подготовить наш массив с его помощью[1].

```
10 CREATE # 10, "sin_tsel.dat"
20 FOR ugol = 0 to 90
30 PRINT # 10, INT (256 * SIN (ugol * 3.142/180) + 0.5)
40 NEXT ugol
50 CLOSE*10
```

Обратите внимание, что в строке 30 для правильного округления к значению синуса прибавляется 0,5 .

Теперь содержимое полученного файла можно записать в массив s, и всё это проделать внутри функции obnulit():

```
int obnulit()
{
    FILE *ptk;
    int ugol;
    napravl = 0;
    sled();
    pomestit(SERED_X, SERED_Y);
    ptk = fopen("sin_tsel.dat", "r");
    for (ugol = 0; ugol < 91; ugol++)
         fscanf(ptk, "%d", &s[ugol]);
    fclose(ptk);
```

1 Внезапно. (прим. перев.)

```
    return 0;
}
```

Как и задумывалось, направление сбрасывается в 0°, включается режим оставления следа, "черепаха" перемещается в середину экрана. Конечно, расположение этой середины зависит от того, какой экран используется, поэтому её координаты предполагается задавать обработчиком `#define`. В конце читается файл со значениями синуса угла, и значения записываются в массив `s`.

Теперь возмёмся за тригонометрию.

```
int sin_tsel(a)
int a;
{
    if(a <= 90)
        return s[a];
    if (a <= 180)
        return s[180 - a];
    if (a <= 270)
        return -s[a - 180];
    if (a > 270)
        return -s[360 - a];
}

int cos_tsel(a)
int a;
{
    a = 90 - a;
    if(a<0)
        a += 360;
    return sin_tsel(a);
}
```

Как видно, подразумеваемая величина угла `a` не превышает 360. Поэтому в функции `dovern()` предстоит предусмотреть, чтобы этого превышения не происходило.

Управляющие функции

Когда почва подготовлена, можно приступать к написанию намеченных функций для управления "черепахой".

```
int dovern(a)
int a;
```

```
{
    napravl += a;
    napravl %= 360;
    if (napravl < 0)
         napravl += 360;
    return 0;
}
```

К значению направления прибавляется значение угла `a`, и полученное делится с остатком на 360. Казалось бы, всё что нужно — но требуется предусмотреть ещё одну вещь, а именно — случай, когда остаток от деления отрицательный. В этом случае он оказываенся вне пределов требуемых значений, поэтому к любому отрицательному остатку необходимо прибавлять 360 градусов.

Будем учитывать этот вопрос в дальнейшем.

```
int na_ugol(a)
int a;
{
    napravl = a % 360;
    if (napravl < 0)
         napravl += 360;
    return 0;
}
```

Включение и выключение режима рисования — это просто переключение переменной между двумя состояниями.

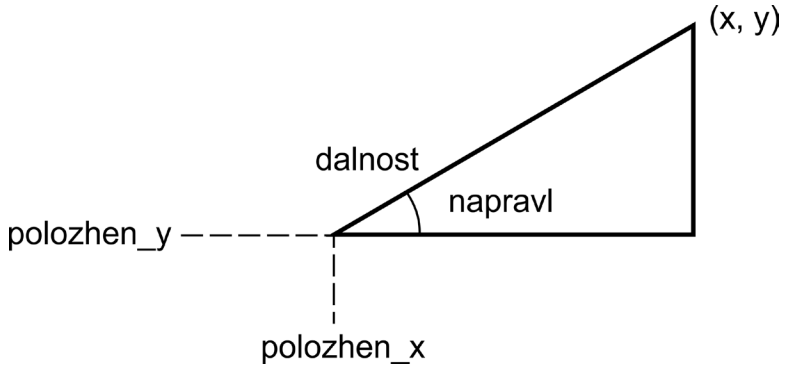
```
int sled()
{
    sled = 1;
    return 0;
}

int ne_sled()
{
    sled = 0;
    return 0;
}
```

Функция `dvigat()` будет уже чуть сложнее.

```
int dvigat(dalnost)
int dalnost;
```

```
{
    int x, y;
    x = polozhen_x + dalnost*cos_tsel(napravl);
    y = polozhen_y + dalnost*sin_tsel(napravl);
    if (sled)
        line(polozhen_x > > 8, polozhen_y > > 8, x > > 8, y >
> 8);
    polozhen_x = x;
    polozhen_y = y;
    return 0;
}
```

Взглянем на рисунок и рассмотрим соотношения в нём (пока не принимая во внимание введённый нами множитель 256):



Справедливы следующие равенства:

```
x - polozhen_x = dalnost * cos(napravl)
y - polozhen_y = dalnost * sin(napravl)
```

Если мы подставим в них наши увеличенные в 256 раз и округлённые до целых значения синуса и косинуса угла, то всё находящееся в левой части равенств тоже необходимо увеличить в 256 раз. Что и делается в функции.

И если у "черепахи" включен режим рисования, то при движении она оставляет след: используется функция `line()`[1], рисующая линию на экране. Вам следует выяснить, как выполнена данная функция в графической библиотеке вашего компилятора. Здесь предполагается, что эта функция вызывается следующим образом:

```
line(x1, y1, x2, y2);
```

, где сначала вводятся исходные координаты (откуда чертить), а затем конечные координаты.

Осталось написать функцию `pomestit()`.

---

1 По стандарту функций графики в Си не предусмотрено вообще. Некоторые исполнения Си (напр. Борланд) содержат подключаемый файл данных `graphics.h`, содержащий описание функций графики, предлагаемый этим исполнением. Именно о таком случае идёт речь в этой главе, о чём авторы почему-то умалчивают. Поэтому увы, в стандартном Си ничего этого делать нельзя. (прим. перев.)

```
int pomestit(x, y)
int x, y;
{
    polozhen_x = x << 8;
    polozhen_y = y << 8;
    if (sled)
        point(x, y);
    return 0;
}
```

Незнакома здесь только функция `point()`[1]; она закрасит целевой пиксель, если у "черепахи" включен режим рисования.

Вы видите, что вместо обычного действия умножения или деления на 256 я постоянно двигаю биты — то влево, то вправо. Очевидно, это делается для ускорения; суть не в этом, а в том, что тут есть подводный камень, который ещё даст о себе знать в следующей главе.

Когда "целое число" побитово сдвигается вправо, то бит, отвечающий за знак, должен быть переписан в крайний бит слева. Как правило, всё именно так и происходит, однако в некоторых исполнениях Си сдвиг битов вправо является логическим (т.е. бит знака числа теряется), вне зависимости от вида данных переменной.

Обойти этот вопрос довольно просто — требуется избавиться от самой возможности наличия отрицательных координат, сделав всё пространство координат единым (когда точка отсчёта расположена в углу экрана). Вполне возможно, что графические функции вашего Си работают именно так. В этом случае переменные для координат (`x`, `y`, `polozhen_x` и `polozhen_y`) следует сделать беззнаковыми.

Встраивание "черепашьей графики"

Что ж, у нас на руках несколько функций, которые мы вольны добавлять в любые программы, где они понадобятся, путём прописывания их в коде этих программ. Но это же грубый и неправильный способ, противоречащий мышлению, которое предлагает Си. Большинство Си-компиляторов предоставляют не менее трёх более приемлемых решений.

1. Включать функции в код с помощью предобработчика `#include`. Сначала требуется все функции "черепашьей графики" записать в файл с названием, допустим, `cher-gr.c`. Теперь в начале кода программы остаётся сделать такое включение:

```
#include <cher-gr.c>
```

Благодаря этому функции графики будут добавлены в файл компилятором.

Недостаток здесь в следующем. В код программы будет включено всё содержимое файла с функциями, даже если не все функции использованы в программе. Вследствие этого может образоваться множество лишнего кода; в данном случае, когда функций немного и все они очень малы, на это можно закрыть глаза.

2. Расширить имеющиеся библиотеки. То есть добавить новые функции в код стандартных поставляемых с вашим компилятором библиотек, и перекомпилировать их. Таким образом "черепашья графика" на вашем компьютере станет частью вашего Си. Разумеется, перед этим

1 Также является сторонней. (прим. перев.)

следует убедиться в надёжности созданных функций, поскольку неотлаженный код включать в библиотеку крайне нежелательно.

В каких-то компиляторах имеется средство добавления скомпилированных функций в уже скомпилированную библиотеку. Это спасает от дополнительных временных затрат, но ни от чего более.

3. Воспользоваться построителем. Про построитель я говорил только, что он создаёт ссылки на библиотечные функции, но как правило он умеет больше. А именно — создавать ссылки на функции из любых файлов `.c`, если такой файл был подан ему на ввод. Также построитель умеет сообщать о неудаче в случае, если функция не найдена; сообщать, что это за функция; даёт возможность указать другой файл для её поиска. Подробности всего этого зависят от исполнения.

В заключительной части главы мы рассмотрим, как применение "числа с плавающей точкой" повлияет на задачу создания "черепашьей графики".

В погоне за синусами

Привычный подход к вычислению функций наподобие синуса и косинуса числа — находить приближённое значение через более простое выражение, например через многочлен, или через частное двух многочленов (рациональное число). Этому посвящён отдельный раздел математики под названием "теория приближения".

Первое, что приходит в голову — вычислить приближение синуса числа "икс" через ряд Тейлора:

$$x - x^3/6 + x^5/120 - x^7/5040 + \ldots$$

Возможно, это не самый лучший выбор. Некоторые другие способы приближения менее затратны и более точны. Но у этого способа есть и свои преимущества.

Возьмём ряд до многочлена $x^5/120$ включительно, и преобразуем его к такому виду:

$$x(1 + x^2/6(x^2/20 - 1))$$

Воплотить такое вычисление синуса числа в Си очень просто:

```
double sinch_tlr(x)
double x;
{
    double a;
    a = x*x;
    return (x * (1 + 0.166667*a * (0.05*a - 1)));
}
```

В нижеследующей таблице приведено сравнение значений (от 0 до 1) синуса числа[1] со значениями, высчитанными через приближения Тейлора и Паде. При значениях "икс" от 0 до -1.6 значения по модулю остаются теми же, просто становятся отрицательными (это вы можете проверить самостоятельно[2]).

---

1 Синус числа `a` — это синус угла, равного `a` радианам. (прим. перев.)

2 В исходной книге авторы вновь предлагают проделать это на Бейсике. (прим. перев.)

Сравнение значений синуса числа со значениями, полученными рядом Тейлора до степени 5, а также приближением Паде

		Тейлор	Паде
x	sin(x)	sinch_tlr(x)	f(x)
0	.0000	.0000	.0000
.1	.0998	.0998	.0998
.2	.1987	.1987	.1987
.3	.2955	.2955	.2955
.4	.3894	.3894	.3894
.5	.4794	.4794	.4794
.6	.5646	.5646	.5646
.7	.6442	.6442	.6442
.8	.7174	.7174	.7173
.9	.7833	.7833	.7832
1.0	.8414	.8416	.8413
1.1	.8912	.8916	.8908
1.2	.9320	.9327	.9313
1.3	.9635	.9648	.9624
1.4	.9854	.9874	.9834
1.5	.9975	1.0007	.9944
1.6	.9996	1.0007	.9948

Для вычисления 4-х разрядов после запятой — в пределах от 0 до 1, а также от -1 до 0 — функция `sinch_tlr()` отлично подходит. Для 3-х разрядов — она подойдёт для всего пространства значений, от -π/2 до π/2 .

Легко обнаружить, что ряд Тейлора как приближение гораздо лучше срабатывает для малых знаений "икс" (около ноля), чем для больших. Также, когда с помощью ряда Тейлора достигается необходимая точность для значения "икс" около 1 (например), то точность для значения около 0 теряется. Множество вычислений проводятся впустую. Так или иначе, используемая нами формула (см. выше) всё равно проста.

Другой путь — использовать алгебраическую функцию в виде рационального выражения, например такую:

$$(60x - 7x^3)/(60 + 3x^2) = f(x)$$

Она тоже подойдёт в качестве приближения для нахождения синуса числа на промежутке от -π/2 до π/2 (см. таблицу выше). Также функция удовлетворяет необходимым требованиям:

$$f(-x) = -f(x)$$

$$f(0) = 0$$

Знаменатель $60 + 3x^2$ не может оказаться равным нолю. Если желаете знать — эта функция использует приближение Паде, о котором понаписано много чего много где.

Таким образом, помимо написания кода, требуется решить два вопроса:

1) преобразования градусов в радианы

2) обработки углов, лежащих вне обозначенных пределов — вне которых каждое приближение будет стремительно терять свою точность.

При решении первого вопроса учтём следующее:

а) `x` градусов = `x/180` радиан

б) значение `355/113` очень близко к числу пи.

Если точнее, то `355/113` = `3.141593` , то есть точность составляет 6 знаков после запятой. Чтобы (приблизительно) получить угол, считаем:

```
355*x / 113*180 = 71*x / 113*36
```

Немного преобразуем:

```
71/113 * x/36
```

Тогда получится сохранять точность как можно дольше. Допустим, что "икс" равен 45°. Тогда

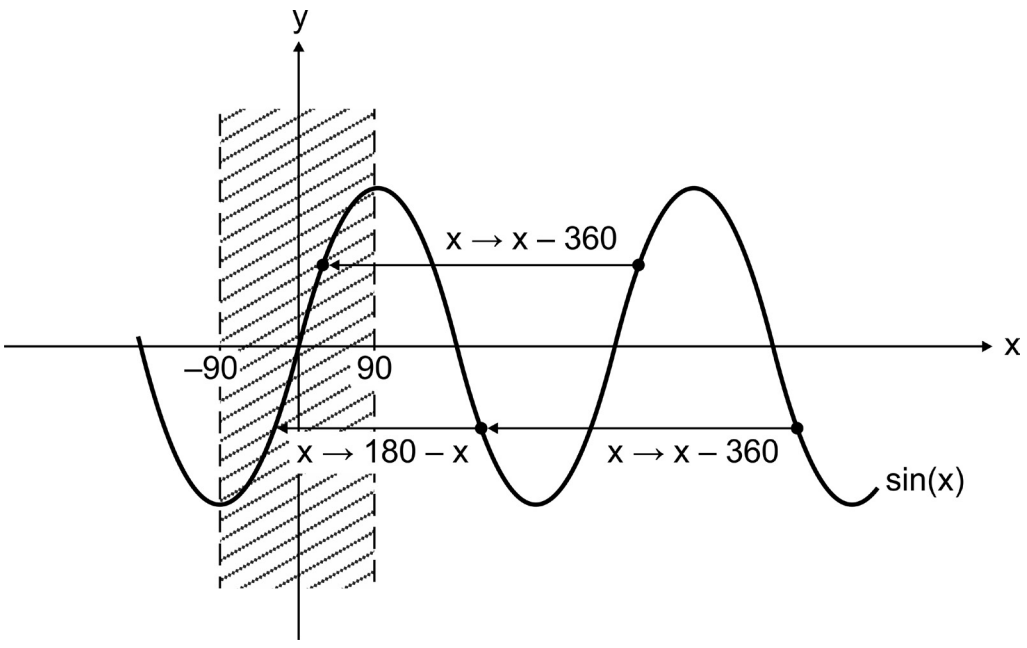
```
          x/36     ~ 1.25
     -x/36 * 71    ~ 89
71/113 * x/36      ~ 0.78
```

Второй вопрос решается с помощью того же способа ограничения предела значений, который мы применили в функциях `sin_tsel()` и `cos_tsel()`. Рассуждая в градусах (ибо 360 — это целое, а `2п` — нет), мы имеем два утверждения:

1) sin(x + 360) = sin(x)

2) sin(180 - x) = sin(x)

Рассмотрите рисунок.



Воспользовавшись действием из первого утверждения, можно уместить "икс" в пределы от 0 до 360, или, что предпочтительней, от -90 до 270. Второе утверждение показывает, как уместить "икс" в пределы от -90 до 90, в которых хорошо работает приближение `f(x)` по подобию приближения Паде, рассмотренное выше.

Даёшь тригонометрию[1]

Теперь в Си можно воплотить вычисление синуса числа. Сначала предположим, что "икс" лежит в пределах от -90 до 90.

```
double sinch_tlr(x)
double x;
{
    double a;
    x = x/36 * 71/113;
    a = x*x;
    return (x * (1 + 0.166667*a * (0.05*a - 1)));
}
```

Теперь допустим, что это может быть не так. Тогда прибавим к "икс" (или вычтем) значение, кратное 360 градусам, чтобы уложиться в пределах от -90 до 270; затем воспользуемся получившейся выше функцией, чтобы уложиться в пределы от -90 до 90.

```
double sin(x)
double x;
{
    while (x < - 90)
         x += 360;
    while (x > 270)
         x -= 360;
    return sinch_tlr(x);
}
```

Как быть с косинусом? Можно было бы снова прибегнуть к приближению Паде, но для косинуса. Но можно поступить проще, поскольку, как нам уже известно,

```
cos(x) = sin(90 - x)
```

(если "икс" — значение в градусах)

Поэтому решение — проще некуда:

```
double cos(x)
double x;
{
    return sin(90 - x);
}
```

ЗАДАНИЕ

Напишите функции для приблизительного вычисления тригонометрических функций:

tan(x) = sin(x) / cos(x)

cot(x) = cos(x) / sin (x)

sec(x) = 1 / cos(x)

cosec(x) = 1 / sin(x)

---

1 Нет. (прим. перев.)

ГЛАВА 16

...и её применение

Настало время развлекаться и заставлять рисующую "черепашку" работать на благо художников, учёных и всего прогрессирующего человечества. Эта глава предполагает использование целочисленной "черепашьей графики"; но все представленные здесь программы заработают с пройдённой нами высокоточной графикой, если соответствующим образом поменять вид данных переменных и их объявление.

Подразумевается также, что в начале кода с помощью предобработчика `#include` добавлены все созданные нами функции "черепашьей графики", дабы мы свободно могли ими пользоваться.

По мере написания новых функций вы можете добавлять их в файл с "черепашьей графикой", тем самым расширяя возможности получающегося набора.

В главе рассмотрено создание прямоугольников, обычных и наклонённых многоугольников, окружностей, звёзд, спиралей и угловатых спиралей, а гвоздём программы станет снежинка Коха.

Замечательность "черепашьей графики" в том, что вы вольны создавать на экране что захотите, не задумываясь, как это всё работает.

Прямоугольники

Сначала запасёмся четырьмя ожидаемыми указаниями:

```
int vlevo(dalnost)
int dalnost;
{
    na_ugol(180);
    dvigat(dalnost);
    return 0;
}

int vpravo(dalnost)
int dalnost;
{
    na_ugol(0);
    dvigat(dalnost);
    return 0;
}

int vverh(dalnost)
int dalnost;
{
    na_ugol(90);
    dvigat(dalnost);
```

```
    return 0;
}

int vniz(dalnost)
int dalnost;
{
    na_ugol(270);
    dvigat(dalnost);
    return 0;
}
```

Очень надеюсь, что они предельно понятны!

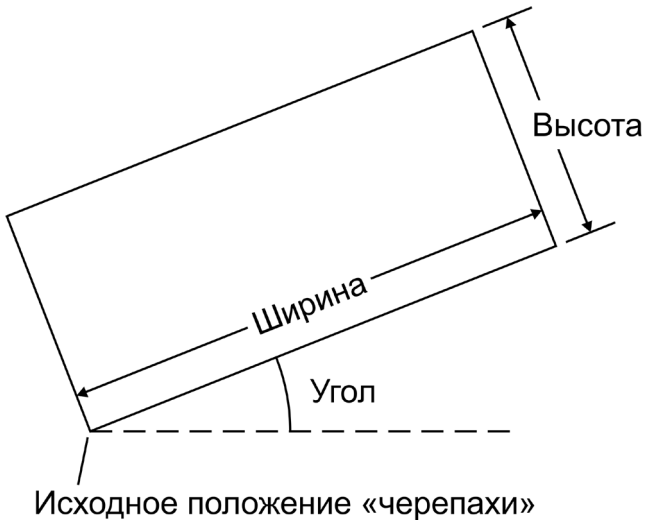
Теперь напишем функцию отрисовки прямоугольника, начиная от левого нижнего его угла (с текущего положения "черепахи"), с заданными высотой и шириной:

```
int pryamoug(visota, shirina)
int visota, shirina;
{
    na_ugol(0);
    vpravo(shirina);
    vverh(visota);
    vlevo(shirina);
    vniz(visota);
    return 0;
}
```

Задача 1

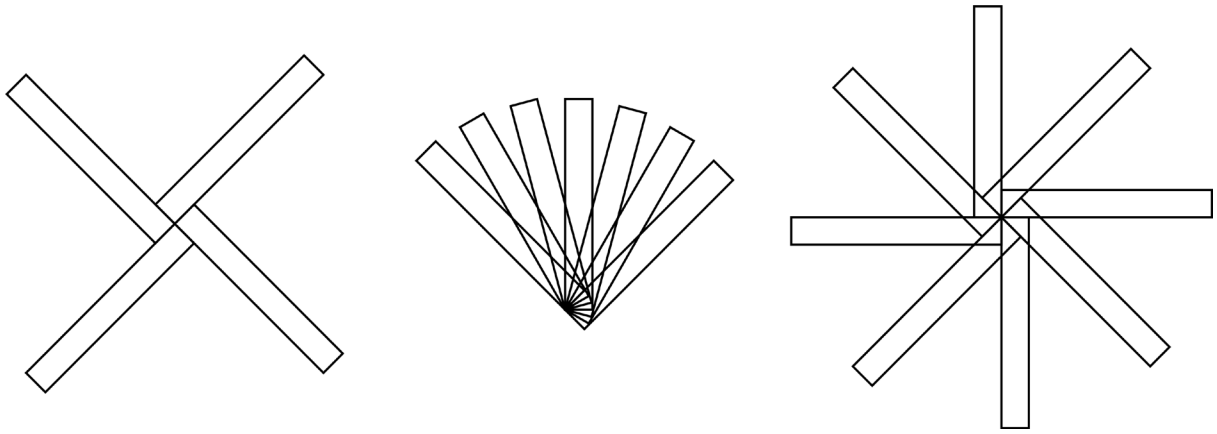
Здесь я предложу вам создать функцию более общего применения. Пусть она имеет вид `nakl_prmgk(visota, shirina, ugol)`, и отрисовывает прямоугольник с возможностью его наклона на заданный угол, как показано на рисунке ниже.

Выполните эту задачу сейчас, поскольку эта функция понадобится нам далее. (Если вдруг у вас затруднения, то загляните в "ответы".)



Движение лопастей

Наклонёнными прямоугольниками мы можем рисовать мельницы, лопасти моторов и пропеллеры, а также основания, на которых они закреплены. См. рисунок.



С этим справится одна-единственная функция следующего вида:

```
lopasti (ugol_perv, otkl, kolvo, visota, shirina)
```

ширина прямоугольника-лопасти

высота прямоугольника-лопасти

количество лопастей

отклонение (угол) между лопастями

угол наклона первой рисуемой лопасти

Сделаем её.

```
int lopasti(ugol_perv, otkl, kolvo, visota, shirina)
int ugol_perv, otkl, kolvo, visota, shirina;
{
    int scht, a;
    a = ugol_perv;
    for (scht = 0; scht < kolvo; scht++)
    {
         nakl_prmgk(visota, shirina, a);
         a += otkl;
    }
    return 0;
}
```

Или, возможно, вы предпочтёте такой её вид:

```
lopasti2(ugol_perv, otkl, ugol_kon, visota, shirina)
```

угол наклона конечной лопасти

Не беда.

```
int lopasti2(ugol_perv, otkl, ugol_kon, visota, shirina)
int ugol_perv, otkl, ugol_kon, visota, shirina;
{
    int a;
    for (a = ugol_perv; a <= ugol_kon; a += otkl)
        nakl_prmgk(visota, shirina, a);
    return 0;
}
```

То, что изображено на рисунке выше, было получено программами с такими вызовами этой функции:

```
а) lopasti( 45, 90, 4, 20, 40 )
б) lopasti( 45, 10, 10, 5, 40 )
в) lopasti( 0, 45, 8, 20, 30 )
```

Задача 2

Повторите на экране эти рисунки. Затем посмотрите, что получится из вот такого кода:

```
int main()
{
    int scht;
    obnulit();
    for (scht = 0; scht < 8; scht++)
    {
        na_ugol(0);
        dvigat(10);
        sled();
        lopasti( 30, 5, 24, 60, 4 );
        ne_sled();
    }
    return 0;
}
```

Многоугольники

Вот так можно нарисовать многоугольник, имеющий А сторон, с величиной стороны Б:

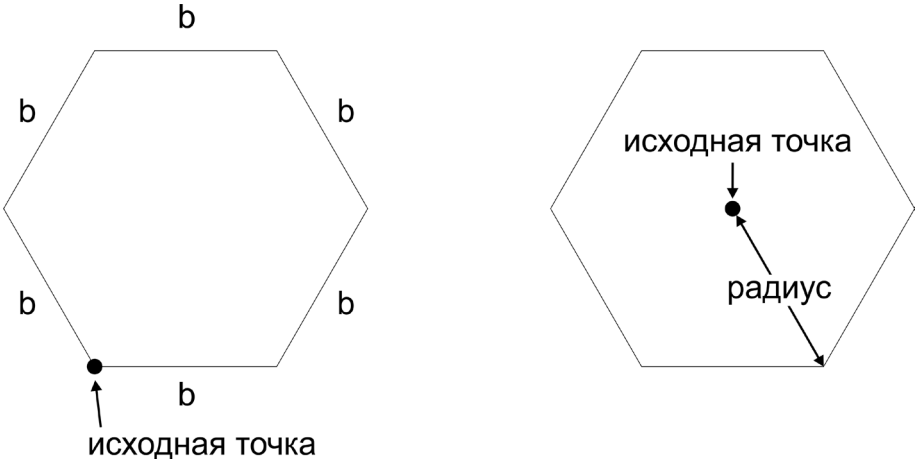
```
int mnogoug(a, b)
int a, b;
{
    int scht = 0;
    while (scht++ < a)
    {
```

```
            dvigat(b);
            dovern(360 / a);
        }
        return 0;
    }
```

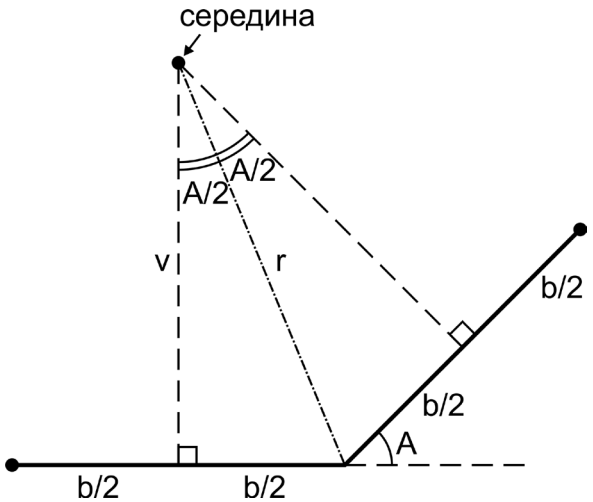
После этого можно проделывать вещи наподобие таких:

```
int main()
{
    int b;
    obnulit();
    for (b = 10; b < 100; b += 10)
        mnogoug(5, b);
    return 0;
}
```

Тем не менее, применительно к многоугольникам полезно знать расположение середины многоугольника и его радиус. См. рисунок.



Мы можем выразить сторону многоугольника через радиус, а также понять как работать с серединой, с помощью тригонометрии. Рассмотрим это на примере двух последовательных сторон многоугольника, показанных на следующем рисунке.



Исходя из рисунка:

```
sin(A/2) = (b/2)/r ,
cos(A/2) = v/r
```

Если количество сторон — `a`, то угол A равен:

```
A = 360/a
```

, а также

```
A/2 = 180/a
```

Таким образом можем записать выражение:

```
b = 2 * r * sin(A / 2) =
= 256 * r * sin(A / 2) / 128 =
= r * s[180/a] / 128
```

(`s[]` — это массив с целочисленными значениями синуса, описанный в главе 15. Он доступен всем функциям, исходя из предупреждений в начале главы. — прим. перев.)

Похожим образом можно выразить высоту `v`:

```
v = r * s[90 - 180/a] / 256
```

(использовано свойство `cos(x) = sin(90 - x)`)

Всё готово для написания функции.

```
int mnogoug_sered(a, r)
int a, r;
{
    int b;
    b = r * s[180/a] / 128;
    ne_sled();
    na_ugol(-90);
    dvigat(r * s[90 - 180/a] / 256);
    dovern(-90);
    dvigat(b / 2);
    dovern(180);
    sled();
    mnogoug(a, b);
    return 0;
}
```

Окружность

Изобразить окружность можно многоугольником с количеством сторон, скажем, 30.[1]

Вот так.

```
mnogoug_sered(30, r)
```

Чем выше радиус этой "окружности", тем выше вероятность, что придётся увеличивать количество сторон для зрительной её гладкости.

Завитушки

Если долго двигать и поворачивать "черепаху", притом увеличивая дальность двигания и (возможно) угол поворота, то нарисуется угловатая спираль. Вот пример функции рисования спиралей:

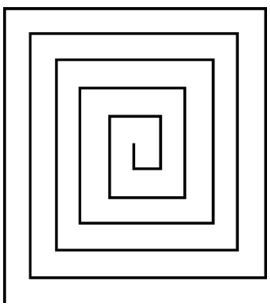
```
int spiral(kolvo, dalnost, prir, ugol)
int kolvo, dalnost, prir, ugol;
{
    while (kolvo-- >= 0)
    {
        dvigat(dalnost);
        dovern(ugol);
        dalnost += prir;
    }
    return 0;
}
```

(`prir` — приращение. — прим. перев.)

Такой вызов этой функции

```
spiral(40, 10, 1, 90);
```

нарисует то же, что показано на следующем рисунке.



Если слегка изменить угол наклона, то они также начнут оказывать влияние на степень заворачиваемости спирали.

---

1 "И так сойдёт" (с). (прим. перев.)

Задача 3

Постарайтесь написать функцию рисования спиралей с углом поворота 90 градусов, вызывающую саму себя (т.е. в которой использована рекурсия).

Снежинка Коха

В начале 20-го века Хельге фон Кох создал бесконечную замкнутую кривую, которую назвал "снежинкой". Она образуется равносторонними треугольниками (без основания), а в последующих проходах каждая сторона каждого треугольника разбивается новым (меньшим) равносторонним треугольником.

Способов нарисовать эту "снежинку" множество. Я воспользуюсь способом, предполагающим строковое значение, в котором содержится длинная последовательность однобуквенных обозначений "влево" и "вправо"[1] — соответственно, `l` и `p`.

Начальный треугольник создаётся так:

```
"Ill"
```

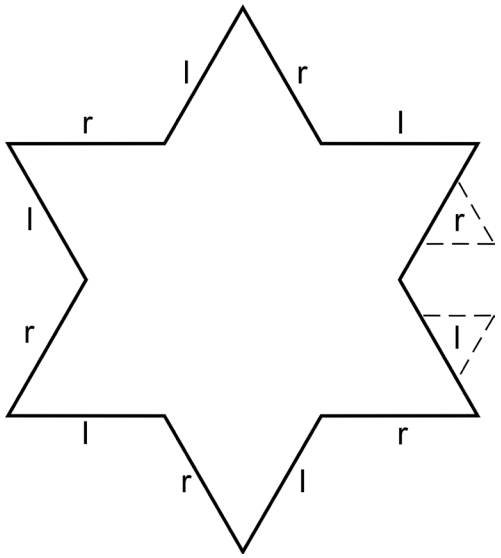
Далее создаётся замкнутая кривая в виде шестиконечной звезды:

```
"lplplplplplp"
```

То есть для каждого последующего прохода в управляющей строке требуется заменить каждую букву `l` на последовательность `lplp`, а каждую букву `p` — на последовательность `pplp`.

Рассмотрите рисунок, чтобы лучше понять вышенаписанное. (Если "черепаха" повёрнута на три часа, то звезда на этом рисунке начинает создаваться с нижнего треугольника, с его вершины, смотрящей строго вниз. — прим. перев.)

Перед началом требуется решить одну вещь. Строковое значение, содержащее последовательности `lp`, назовём `snezh`; также потребуется временная строка `vrem`. Какой объём этим строкам отвести?



При каждом проходе объём строки `snezh` увеличивается вчетверо:

проход	объём
1	3
2	12

1 Поворот влево \ вправо, т.е. против \ по часовой стрелке на 60 градусов, и рисование одной стороны равностороннего треугольника (прим. перев.)

```
3            48
4            192
5            768
```

Примем, что объёма 800 байт нам будет достаточно.

Вот код самой программы.

```
#include <stdio.h>
char snezh[800], vrem[800];
int main()
{
    int prhd, b;
    obnulit();
    printf("Введите длину стороны: ");
    scanf("%b", &b);
    strcpy(snezh, "lll");
    for (prhd = 0; prhd < 5; prhd++)
    {
        pomestit(50, 50);
        na_ugol(-120);
        if (prhd > 0)
            sled_prohod();
        risovat(b);
        zaderzhka(ZADERZHKA);
    }
    return 0;
}
```

Здесь видим три новые функции. `zaderzhka()` говорит сама за себя:

```
int zaderzhka (v)
int v;
{
    while(v--)
    ;
    return 0;
}
```

Величина `ZADERZHKA` задаётся через предобработчик `#define` по вкусу. Она задаёт то, как долго на экране продержится до следующего прохода нарисованная снежинка.

Функция `sled_prohod()` выполняет замену буковок в управляющей строке.

```
int sled_prohod()
{
    int scht = 0;
    strcpy(vrem, "");
```

```
    while (snezh [scht])
        if (snezh[scht++] == 'l')
            strcat(vrem, "lplp");
        else
            strcat(vrem, "pplp");
    strcpy(snezh, vrem);
    return 0;
}
```

И наконец, функция выполняющая собственно рисование:

```
int risovat(b)
int b;
{
    int scht = 0;
    clearscreen();
    while (snezh[scht])
    {
        if (snezh[scht++] == 'l')
            dovern(120);
        else
            dovern(-60);
        dvigat (b);
    }
    return 0;
}
```

Здесь на месте функции `clearscreen()` в своём коде вставьте предусмотренную в вашем исполнении языка возможность очистки экрана. Возможно, это будет вывод какого-либо управляющего знака функцией `printf()`, или отдельная функция.

Раздумья вдогонку

Всё это только начало. Если вы поняли всё вплоть до этой точки, то и сами справитесь с дальнейшим расширением возможностей своей домашней "черепашьей графики". А также заточить её на бóльшую точность с помощью поддержки дробных значений.

Ответы

Задача 1

```
int nakl_prmgk(visota, shirina, ugol)
int visota, shirina, ugol;
{
    int scht;
    na_ugol(ugol);
```

```
    for (scht = 0; scht < 2; scht++)
    {
        dvigat(shirina);
        dovern (90);
        dvigat(visota);
        dovern (90);
    }
    return 0;
}
```

Обратите внимание, что после выполнения "черепаха" останется повёрнутой в последнем использованном направлении.

Задача 2

а)

```
int main()
{
    obnulit();
    lopasti( 45,90,4,20,40 );
    return 0;
}
```

б), в) просто измените значения, поданные в функцию `lopasti()`, на нужные.

Задача 3

```
int spiral_rek(kolvo, dalnost, prir)
int kolvo, dalnost, prir;
{
    dvigat(dalnost);
    dovern (90);
    dalnost += prir;
    if (kolvo--)
        spiral_rek(kolvo, dalnost, prir);
    return 0;
}
```

ЗАДАНИЕ

Если вы пробовали создавать многоугольники с числом сторон, которое не кратно 3600, то видели, что рисуемые стороны не сходятся. Причина ясна — при целочисленном делении теряется остаток, что шаг за шагом выливается в значительную погрешность.

Можете ли вы поправить это, сделав проверку наличия остатков от деления (включая сдвиги битов вправо) с последующим округлением значений с остатком?

ГЛАВА 17

О случайных числах

Каждому программисту известно, что случайные числа могут быть в высшей степени полезны, особенно в моделировании физики и в играх. И воспользоваться ими просто — требуется лишь обратиться к нужной функции, которая может называться `RND`, `rand()`, `random()` и т.п. — после чего функция будет пачками выдавать их вам в любых количествах. В библиотеке вашего Си, без сомнений, хотя бы одна такая функция есть.

Но задумывались ли вы когда-нибудь над тем, насколько действенны эти функции, и насколько хороши, насколько случайны получаемые "случайные" числа? Самое время подумать над этим сейчас.

Случайность изначальная

Разумеется, компьютер не может выдавать последовательности случайных числел. По крайней мере если мы хотим получить последовательность `s1,s2,s3,s4,...` так, чтобы каждое следующее число невозможно было предугадать.

В общепринятом виде программа, выдающая "случайные" числа, производит ряд весьма простых арифметических действий над очередным числом `s(a)`, чтобы получить следующее число `s(a+1)`; так что если вы знаете эти действия, то знаете наперёд, какие числа будут выданы. Поэтому про такие программы говорят, что они создают псевдослучайные числа.

Часто используется способ расчёта сравнением первой степени, по следующей обманчиво простой формуле:

$$s_{a+1} = (a*s_a + c) \bmod m$$

Здесь `a`, `c` и `m` — постоянные значения. Деление нацело на `m` образует остаток. Таким образом, если `a`, `c` и `m` объявлены до тела функции `main()`, либо значение им задаёт предобработчик `#define` — то соответствующая функция в Си может выглядеть так:

```
int psevdo_sluch()
{
    s=(a*s + c)%m;
    return s;
}
```

Что касается `s`, то конечно, она должна быть объявлена до тела `main()` — ведь её значение должно сохраняться до следующего вызова функции. Либо, если ваш компилятор это поддерживает, ей может быть задан режим хранения в памяти `static`[1].

Очевидно, что принудительно заданные значения `a`, `c` и `m` вряд ли обеспечивают по-настоящему случайные числа. Например, если их значения — соответственно 4, 2 и 6, то начиная со значения `s` = 1 будет происходить следующее:

```
s
1   (4*1+2)%6 = 0
0   (4*0+2)%6 = 2
2   (4*2+2)%6 = 4
```

1 Подробнее о режимах хранения в памяти — в приложении 2. (прим. перев.)

```
4   (4*4+2)%6 = 0

...
```

То есть последовательность 0, 2, 4 будет повторяться. Чисел, скажем, 3 и 5 мы никогда не дождёмся. Пределы "случайности" сузились как никогда! Даже если бы мы получили последовательность 1, 0, 2, 3, 5, 4 — она бы тоже повторялась. К тому же в двух соседних выдачах невозможно получить одно и то же число.

Этот вопрос решить просто — требуется дать `m` очень большое значение, и тогда длина последовательности (до её повторения по кругу) станет так же очень велика; помимо того, повторение одного и того же числа в соседних выдачах становится возможным.

Однако при больших величинах `m` (например, несколько миллионов) получаемую последовательность изучать тяжеловато. Поэтому пока, рассматривая ещё несколько вопросов, не станем её увеличивать.

Итак, условия `a = 4, c = 2` сработали плохо. Попробуем `a = 1, c = 1`. Получится следующее:

```
s
1   (1*1+1)%6 = 2
2   (1*2+1)%6 = 3
3   (1*3+1)%6 = 4
4   (1*4+1)%6 = 5
5   (1*5+1)%6 = 0
0   (1*6+1)%6 = 1
```

Едва ли это можно назвать последовательностью случайных чисел.

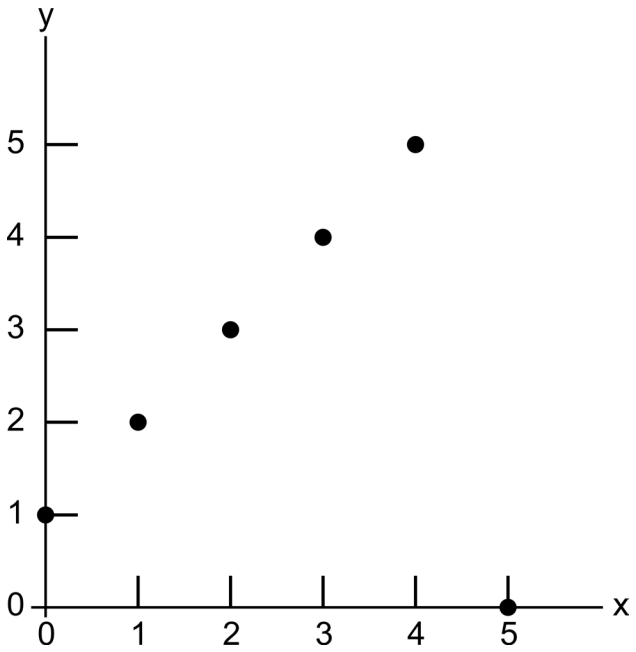
У нас на руках два основных вопроса. Первый: как нам убедиться, что в полученной последовательности присутствуют все возможные значения (далее будем называть такой случай "полная последовательность")? Второй: откуда нам знать, что никакой скрытой связи между числами в последовательности нет?

В книге под названием "Получисленные алгоритмы" (второй том "Искусства программирования") Дональд Кнут приводит отношения чисел `a`, `c` и `m`, с которыми можно получить полные последовательности. Здесь я их повторять не буду, а предложу вам заглянуть в означенную книгу. Данному вопросу Кнут посвятил более сотни страниц.

Проверка на случайность

А кроме того, он приводит несколько способов проверки полученной последовательности чисел на степень полезности — с точки зрения случайных чисел. (И вдовесок пишет, что на такие проверки можно легко затратить больше времени, чем на разработку программ выдачи чисел.) Помня об этом, я ограничусь лишь одним способом такой проверки, который считаю наиболее удачным.

Будем стремиться к случаю, когда между данным и последующим числом в выдаче нет связи. Пойдём от противного: предположим, что они взаимосвязаны. Обозначим их как координаты `x` и `y`, и отразим все такие пары чисел на графике. Для рассматриваемой нами последовательности 1-2-3-4-5-0 график получится такой:



Предсказуемость очевидна. Однако если какая-либо последовательность не проявит такой предсказуемости, то нам всё равно следует проверять её дальше. Наблюдается ли связь между данным числом и вторым после него? Третьим, четвёртым и так далее? Каждый раз требуется подставлять новые значения `x` и `y`. Для последовательности из миллиона чисел графиков понадобится очень даже немало.

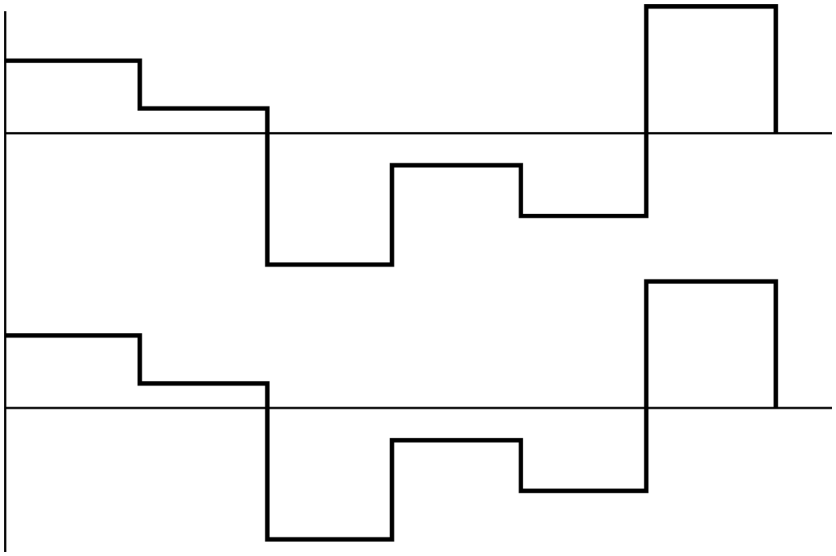
Вычисления приходят на помощь

Даже если мы станем (или заставим компьютер) долго и упорно рисовать, то как нам определить, что наша оценка была верной, и никакой явной зависимости действительно нигде нет? Требуется некий надёжный приём.

Подойдём к вопросу с другой стороны. Представим, что выдаваемая последовательность чисел — это некий шум (вообразите радиоприёмник, не настроенный на какую-либо станцию). При этом среднее значение этого шума — ноль, то есть график должен быть уравновешенным.

Изобразить это просто: требуется вычесть среднее значение последовательности из каждого числа этой последовательности. Например, для последовательности 4-3-0-2-1-5 получатся значения 1.5, 0.5, -2.5, -0.5, -1.5 и 2.5.

Посмотрим на график этой полученной последовательности (график продублирован):



Перемножим значения на совпадающих промежутках графиков друг на друга, и полученные значения сложим:

```
1.5 x 1.5    =    2.25
0.5 x 0.5    =    0.25
-2.5 x -2.5  =    6.25
-0.5 x -0.5  =    0.25
-1.5 x -1.5  =    2.25
2.5 x 2.5    =    6.25

                  _____
                  17.5
```

Так как по сути мы возводили значения в квадрат, то получили только положительные значения. Если попадётся очень длинная последовательность, то итоговое значение быстро вырастет. Однако если второй график сдвинуть вправо на один промежуток (рассматривая его как зацикленный), то получится следующее:

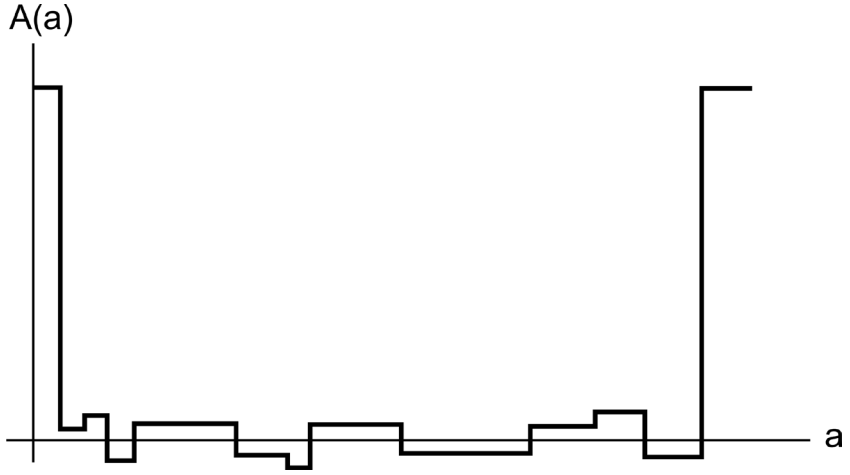
```
1.5 x 2.5    =    3.75
0.5 x 1.5    =    0.75
-2.5 x 0.5   =    -1.25
-0.5 x -2.5  =    1.25
-1.5 x -0.5  =    0.75
2.5 x -1.5   =    -3.75

                  _____
                  1.5
```

Теперь, как видно, отрицательные значения не исключаются. И если последовательность действительно случайна, то вероятность появления отрицательных значений равна вероятности появления положительных. Таким образом сумма всех значений должна или составить ноль, или как можно более к нему приблизиться.

Проделанный нами сдвиг графика позволяет сравнивать соседние значения в последовательности, как в предыдущем разделе. Если же сдвинуть его на два промежутка — сравнение производится со вторым последующим числом. И так далее. Всё это можно выразить в следующем виде:

$$A(a)=\sum_{t} s(t).s(t+a)$$

, где `s(t)` — это случайное число, выданное на промежутке времени `t`, а `s(t+a)` — число, выданное на `a` промежутков позднее. Означенная сумма должна быть близкой к нолю для всех значений `a`, кроме ноля. Если построить график функции для всех значений `a`, то мы получим следующее:



Второе наивысшее значение получается, когда `a` становится равной длине выдачи, то есть когда начинается повторение по кругу.

Если хотите знать, то мы получили подобие того, что называется автокорреляционной функцией (зависимость взаимосвязи функции и её задержанного во времени дубля от величины задержки).

Обратите внимание — я не утверждаю, что функция `A(a)` при `a` не равном нолю должна всегда выдавать ноль. Как отмечает Кнут, это было бы подозрительно хорошо. Непредсказуемость стала бы предсказуема! Тогда встаёт вопрос — какова же нужная степень приближённости к нолю? Простого ответа на этот вопрос нет. Можно лишь сказать, что относительно значения при `a` = 0 значение должно быть явно малым.

Задача 1

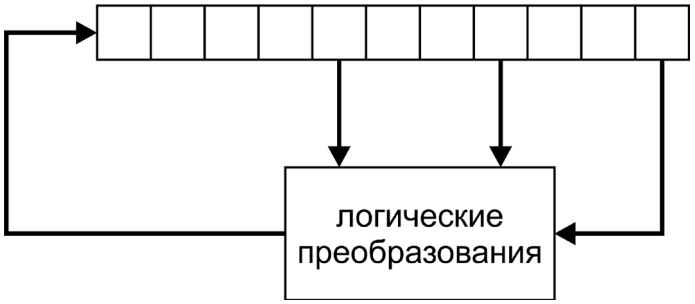
Напишите программу для вычисления функции `A(a)` для первых двухсот чисел, выдаваемых библиотечной функцией выдачи случайных чисел.

Преобразования в регистре

Подход, использующий сравнение первой степени, обладает двумя основными недостатками. Первый: для выдачи каждого числа требуются действия умножения, сложения и деления (с целью получения остатка). Это снижает скорость выполнения, что плохо — ведь как правило, любая программа, использующая случайные числа, обращается за их выдачей множество раз, чтобы получить более-менее ясную картину происходящего. Ведь если вы бросаете монетку шесть раз, и все шесть раз выпал орёл, то вы не делаете вывод, что это будет происходить всегда, а продолжаете её бросать (какая-то депрессивная картина — прим. перев.).

Второй недостаток вытекает из предыдущего раздела. Последовательность случайных чисел достаточной длины придётся проверять на "случайность" слишком долго.

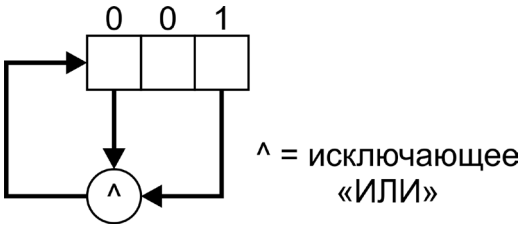
Рассмотрим второй подход, который позволит решить оба недостатка. Представим, что у нас есть регистр (временное хранилище), из которого мы можем извлекать биты и обрабатывать их логическими преобразованиями. После извлечения бита регистр сдвигается на один бит вправо, и извлечённый и обработанный преобразованиями бит записывается в освободившуюся ячейку. Вот таким образом:



Таким способом можно получить последовательность чисел. Но будут ли они случайны? Ответ: да, при соблюдении определённых условий.

Во-первых, следует применять только преобразование "исключающее ИЛИ". Во-вторых, нельзя, чтобы при выдаче мог быть выдан ноль (подумайте, почему). В-третьих — требуется, чтобы выдавалась полная последовательность. Для этого подойдут только преобразования с битами из определённых ячеек; существует сложная математическая теория, в которую не станем здесь закапываться, позволяющая определить, какие именно ячейки требуются. Для нас более важно то, что если мы добьёмся выдачи полной последовательности, то наша функция автокорреляции будет работать как надо.

Рассмотрим пример:



Здесь производится только одно преобразование между старшим и младшим битами 3-битового байта. Как видим на рисунке, происходит сравнение 0 ^ 1 = 1, и после него в регистре образуется значение 100. Полная выдача будет выглядеть так:

```
001 выдаваемое значение =  1
100                        1
110                        1
111                        0
011                        1
101                        0
010                        0
001
```

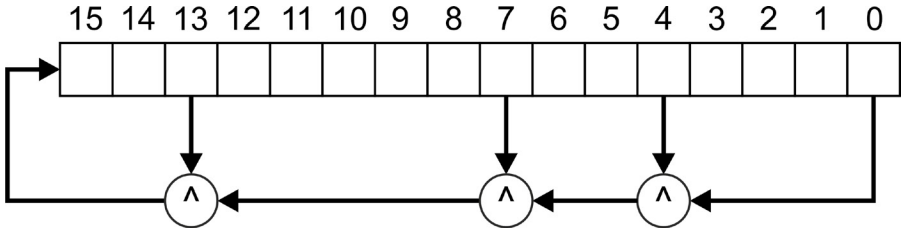
После чего начинается повторение по кругу. Имеем полную последовательность, притом ноль не выдаётся. Проверку на случайность оставляю вам.

Теперь видно, что в условиях Си такой подход заслуживает внимания, поскольку использует действия на машинном уровне — исключающее "ИЛИ", сдвиг, и (предположительно) преобразование "И" для создания маски. Работать всё это должно быстро.

Наконец-то пишем код

Так как в математическую подоплёку дела мы не полезли, то придётся подбирать положения ячеек битов посредством обычного тыка. (Даже если бы полезли — возиться бы всё равно пришлось.)

Предположим для начала, что работаем с 16-битным значением в регистре[1], и что у нас есть возможность вводить номера целевых битов для обработки. Например, на этом рисунке изображено, что мы выбрали биты 0, 4, 7 и 13:



По этим данным будет подготовлен массив соответствующих масок. Для простоты массив будет отовсюду доступным, т.е. объявленным до тела `main()`. Затем 65536 раз будет вызвана функция выдачи случайных чисел.

---

1 Вся дальнейшая работа подразумевает работу с 16-битными "целыми числами". Сегодня "целое число" имеет объём 32 бита, а "целое малой величины" (`short int`) — 16 бит. Поэтому в дальнейших примерах в некоторых местах использован вид `short int`. (прим. перев.)

Каждое число будет сравниваться с исходным значением регистра. Если его исходное значение встретится в значении номер 65536 — это будет означать, что мы получили полную последовательность. Если к этому времени исходного значения вообще не встретится — значит, последовательность с какого-то места начала ходить по кругу.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
short int maski [10];
unsigned short int registr = 1;

int main()
{
    char vhod_stroka [40], *uk;
    unsigned scht;
    uk = vhod_stroka;
    printf("Введите номера битов, отделяя их двоеточиями: ");
    scanf("%s", uk);
    maski_podg(vhod_stroka);
    for (scht = 1; scht < 65535; scht++)
    {
         sluch_chislo();
         if (registr == 1)
         {
              printf("Длина последовательности - %d чисел",
scht);
              exit (0);
         }
    }
    sluch_chislo();
    if (registr == 1)
         printf("Полная последовательность!");
    return 0;
}
```

Не раскрыты две функции. `maski_podg()` создаёт в массиве маски из введённых чисел, а `sluch_chislo()` получает на основании текущего значения регистра очередное случайное

число, и перезаписывает значение в регистре. Обратите внимание — переменная `registr` объявлена беззнаковой. Я позже объясню, почему.

```
int maski_podg(u)
char *u;
{
    int scht = 0;
    strcat(u, ":");
    while (*u)
    {
         maski [scht++] = 1 << atoi(u);
         while(*u++ != ':')
              ;
    }
    maski[scht] = 0;
    return 0;
}
```

Помним, что строка, подаваемая в эту функцию, состоит из чисел, разделённых двоеточиями. Например:

```
0 : 7 : 12 : 13
```

Вы скорее всего заметили, что первым делом функция дописывает в конец этой строки ещё одно двоеточие. Таким образом даже одно введённое число, без введённого разделителя-двоеточия, будет успешно обработано.

Первый цикл `while` ищет окончание строки; в это время в его теле очередная ячейка массива `maski` получает значение, равное сдвинутой влево на какое-то количество бит единице. В нашем примере — сдвинутой на 0 бит, поскольку первое значение адреса указателя — 0 (первое значение во входной строке). Функция `atoi()` рассматривает любые знаки кроме цифр как разделители.

Второй цикл `while` переставляет указатель на знак, следующий за разделителем (двоеточием) — то есть либо на следующее число, которое следует обратить в маску, либо на "нуль", знаменующий окончание строки. После выхода из циклов в массиве также проставляется "нуль".

Осталось написать собственно функцию `sluch_chislo()`.

```
int sluch_chislo()
{
    short int scht = 0, itog_bit = 0;
    while ( maski[scht++] )
         itog_bit ^= izvlech_bit ( maski[scht - 1] );
    registr >>= 1;
    itog_bit <<= 15;
    registr |= itog_bit;
    return 0;
}
```

Итоговый бит вычисляется путём преобразования "исключающее ИЛИ" со значением, которое выдаст ещё одна функция `izvlech_bit()`. Она извлекает из регистра бит по текущей маске, и в соответствии со значением этого бита выдаёт 0 или 1.

Затем значение регистра сдвигается вправо, чтобы освободить место для итогового бита. Здесь можно вернуться к вопросу, почему переменная `registr` объявлена беззнаковой. Во многих случаях сдвиг на один бит вправо является делением значения на 2 (но не во всех исполнениях Си); при этом, если значение отрицательное, то значение самого старшего бита (единица) необходимо всегда сохранять. В соответствии с нашими целями гораздо проще объявить переменную беззнаковой, чем пытаться искать другие пути.

Наконец, итоговый бит сдвигается на 15 разрядов влево, дабы он расположился в старшем бите байта, и регистр скрещивается с ним путём преобразования "ИЛИ".

Конечно, требуется ещё написать функцию `izvlech_bit()`.

```
int izvlech_bit(maska)
short int maska;
{
    if (registr & maska)
         return 1;
    return 0;
}
```

Задача 2

Оцените работу получившейся программы. Постарайтесь добиться выдачи полной последовательности.

Ответы

Задача 1

```
#include <stdio.h>
#include <stdlib.h>
#define SREDN 16284
int sluch_chisla [200];
double avtokor_fun(zaderzhka)
int zaderzhka;
{
    double a = 0;
    int scht;
    for (scht = 0; scht < 200; scht++)
         a += sluch_chisla[scht] * sluch_chisla[ (scht +
zaderzhka) % 200 ];
```

```
    return a;
}
int main()
{
    int scht, zaderzhka;
    for (scht = 0; scht < 200; scht++)
        sluch_chisla[scht] = rand() - SREDN;
    for (zaderzhka = 1; zaderzhka < 200; zaderzhka++)
        printf("задержка = %d значение = %f\n", zaderzhka,
avtokor_fun(zaderzhka));
    return 0;
}
```

Как обычно, необходимые примечания. Условимся, что библиотечная функция выдачи случайных чисел называется `rand()`, и выдаёт числа в пределах от 0 до 32767[1]. Отсюда и значение `SREDN` — оно должно составлять половину от верхнего предела, и тогда в программе значения, записывающиеся в массив со случайными числами, будут то положительными, то отрицательными. Можете менять это значение под имеющийся у вас Си.

Кроме того, подразумевается, что вы можете воспользоваться "числом с плавающей точкой", потому как при умножении вы получите большие значения.

Задача 2

По мере наблюдений вырисовываются следующие положения. Во-первых, самый младший бит (бит 0) обязательно должен быть среди целевых битов, и притом на первом месте; в противном случае последовательности чисел выдано не будет, поскольку после первой же перезаписи значения в регистре оно станет равным нолю.

Во-вторых, есть и другие случаи, когда регистр может обнулиться, поэтому стоит ввести проверку значений в нём. В-третьих — последовательность получается длиннее, если ввести чётное число битов.

В-четвёртых, малое число введённых битов всегда приводит к неудовлетворительной выдаче. Это неудивительно: в таком случае велика вероятность, что получится ряд значений, последовательно уменьшающихся вдвое, потому что происходит сдвиг битов вправо и ничего больше. При этом хорошей автокорреляционной функции не получится, и соответственно, не получится и полной последовательности.

Мне удалось подобрать два набора номеров битов, которые позволяют получить 16-битные полные последовательности.

```
0 : 2 : 4 : 6 : 10 : 12 : 14 : 15
0 : 6 : 10 : 12 : 14 : 15
```

Вторая, очевидно, позволяет добиться более быстрого выполнения.

---

1 Этот предел задаётся в файле данных `stdlib.h`, и обозначается именем `RAND_MAX`. В среде "VisualC++" верхний предел также равен 32767. (прим. перев.)

ЗАДАНИЯ

1. На основании обоих вышеприведённых наборов номеров битов напишите программы выдачи случайных чисел. Теперь вы можете добиваться наивысшей скорости их выполнения, поскольку более нет нужды делать маски в виде переменных. Если ваш компилятор Си поддерживает битовые поля (см. Приложение 1), то и сами маски вам не требуются, что, впрочем, не означает ускорения выполнения кода.

Сравните то, что получится, с библиотечной функцией выдачи случайных чисел. Что быстрее? Что более успешно проходит проверку на "случайность" полученных чисел?

2. Для простоты вышеприведённые примеры кода подразумевают использование компьютера 16-битной разрядности. Строго говоря, это плохо. На Си следовало бы писать переносимые программы, в которых бы через предобработчик #define задавались все показатели, необходимые для работы на данной машине. Перепишите программы проверки "случайности" чисел так, чтобы возможно было работать с машинным словом любой размерности.

3. А также внесите в них такие изменения, чтобы обрабатывались и значения вдвое большей разрядности (причём постарайтесь не использовать уточняющее слово `long`, а то будет неинтересно).

ПРИЛОЖЕНИЕ 1

Оставшиеся "хвосты"

Главной целью этой книги было ненапрягающее повествование. Это привело к тому, что некоторые возможности языка мы обошли стороной или затронули только вскользь. В этом приложении частично восстановим справедливость, кратко описав наиболее значимое из того, о чём умалчивалось ранее.

а) Битовые поля

Это ещё один подход к работе с определёнными битами байта, помимо приёма создания маски с использованием сдвига битов. Число можно разбить на поля из того или иного количества бит, по-особому объявив его структурой. Например:

```
struct
{
    unsigned odin : 1;
    unsigned tri      : 3;
    unsigned chetire  : 4;
} bait;
```

Здесь создан байт с именем `bait`, который состоит из (в порядке от младших к старшим битам) одного бита (поле под названием `odin`), трёх битов (`tri`) и четырёх битов (`chetire`). Теперь можно обращаться к полям числа, как к полям в обычной структуре. Например так:

```
if (!bait.odin)
{
    ...
}
```

Или так.

```
if (bait.chetire < 10)
{
    ...
}
```

(Биты распределяются в порядке от младших к старшим (первое объявленное поле занимает младшие биты, следующее — более старшие, и т.д.). Кроме того, такая структура может содержать и обычные переменные.

Если поле не помещается в биты, оставшиеся свободными в данном байте после предыдущего поля, то оно размещается в следующем байте.

Созданная таким образом структура может иметь произвольный объём в памяти, но всегда кратный 8 битам.

Любой вид данных поля, где не написано слово `unsigned`, имеет знак. Прочие же обозначения видов данных теряют своё значение, и могут использоваться любые из них.

При этом каждое битовое поле в зависимости от своего объёма и вида данных (со знаком или без знака) приобретает пределы возможных хранимых значений.

Нельзя получить объём битового поля действием `sizeof`, а также получить его адрес, т.к. битовые поля очень часто нельзя измерить байтами. — прим. перев.)

б) Явное приведение (преобразование) видов данных

Как правило, программисту не нужно задумываться о промежуточном преобразовании вида данных, необходимом для тех или иных действий.

```
double a;
int b = 3;
a = b;
```

В этом примере без участия программиста выполняется преобразование "целого числа" в "число с плавающей точкой".

Тем не менее, бывают случаи, когда необходимо вручную прописать переменной целевой вид данных, к которому она должна быть преобразована. Это принято называть приведением вида данных (или более топорно — "приведением типов". — прим. перев.). Для вышеприведённого примера явное приведение выглядело бы так:

```
a = (double) b;
```

То есть общий вид у явного приведения следующий:

```
(видДанных) выражение
```

в) Запятая как знак действия

Действие с запятой записывается так:

```
a, b
```

, где `a` и `b` — какие-либо выражения. Оба выражения вычисляются в порядке слева направо, после чего возвращается значение второго выражения.

```
for (scht = 0; scht < 100; x[scht] = 0, scht++)
```

В этом примере, независимо от тела цикла, массив `x` будет заполнен нолями.

г) Входные данные (аргументы, ключи) из командной строки

В большинстве операционных систем предоставляется возможность запускать программы с аргументами. Если требуется, чтобы у программы на Си была возможность такого запуска, то функция `main()` в качестве входных данных должна содержать две переменные — `argc` и `argv`.

```
main(argc, argv)
int argc;
char *argv [];
{
    ...
}
```

argc — это число введённых аргументов (при вводе указания на запуск программы в командной строке количество введённых агрументов подставляется в эту переменную без участия программиста; указание на запуск программы считается первым аргументом. — прим. перев.).

argv — указатель на массив указателей на строковые значения (введённые в командной строке в качестве аргументов), каждое из которых является возможным аргументом.

Для примера предположим, что в каждой функции некоторой программы prog1 содержится такое действие:

```
if(pitanie) printf("Питание включено\n");
```

Переключать значение "питания" между 0 и 1 было бы удобно прямо при запуске программы из командной строки, с помощью соответствующих ключей:

```
exec prog1 pitanie_vkl
exec prog1 pitanie_otkl
```

Воплощается это следующим образом.

```
int pitanie;
int main (argc, argv)
int argc;
char *argv[];
{
    if (argc != 1)
    {
        printf("Введите один аргумент\n");
        exit (0);
    }
    if (!strcmp(argv[1], "pitanie_vkl"))
        pitanie = 1;
    else
        if (!strcmp(argv[1], "pitanie_otkl"))
            pitanie = 0;
        else
        {
            printf("Аргумент введён неверно\n");
            exit(0);
        }
        ...
```

(Функция strcmp() сравнивает два строковых значения и выдаёт 0 при их одинаковости, 1 — если найдены отличия. — прим. перев.)

Проверяется, что введён ровно один ключ, а в противном случае выдаётся ошибка.

В первой ячейке массива `argv` обычно содержится название программы, однако в некоторых исполнениях этот вопрос не определён.

д) Управление распределением памяти "extern"

Мы знаем, что исходный код программы на Си может содержаться в нескольких файлах.

Представим, что в одном из файлов находится подробное объявление переменной до тела функции `main()`. Понятно, что эту переменную уже нельзя объявлять в других файлах, потому что память будет выделена под неё повторно (и это уже будет другая область памяти). А с другой стороны — без её объявления компиляция кода из файла, где она упоминается, завершится с ошибкой, т.к. файлы обрабатываются компилятором раздельно.

Поэтому требуется средство, сообщающее компилятору, что переменная уже была объявлена в другом файле, и под неё уже выделена память.

```
extern int a;
```

Как вы уже поняли, за это отвечает ключевое слово `extern`. Обратите внимание — запись ничем, кроме слова `extern`, не отличается от объявления переменной.

е) Указатели на функции

Вспомним, что имя массива является указателем на первую ячейку этого массива. Точно так же имя функции является указателем на эту функцию. Следовательно, такие указатели можно передавать в другие функции, чтобы по этим указателям происходил вызов функций, на которые они указывают, и возврат значений от них.

Обычное применение для этого — функция поиска, где запросом может быть как число, так и строковое значение. Пример начала объявления такой функции:

```
poisk (analiz_zaprosa, zapros)
int (*analiz_zaprosa) ();
char *zapros;
{ ...
    if ((*analiz_zaprosa)(zapros) == 0)
    {
        ...
    }
    ...
```

Имя функции `analiz_zaprosa` объявлено указателем на функцию, возвращающую "целое число".

(При этом в скобках, которые в данном случае пустые. требуется перечислить виды данных для всех аргументов целевой функции, если таковые есть. Количество перечисляемых видов данных должно соответствовать количеству аргументов у функции.

По сути, создаваемый указатель способен указывать на любые функции, которые соответствуют ему по виду данных возвращаемого значения, количеству аргументов и их видам данных. — прим. перев.)

Также в "поиск" передан указатель на значение запроса (собственно запрос передать нельзя, т.к. поначалу вид данных в нём неизвестен). Назначение этому указателю вида данных "знак ввода" позволит прочитать как число, так и строковое значение, хоть это и несколько ненадёжно. Более предпочтительный подход описан в разделе **и)**, последнем в этом приложении.

Далее в коде можно вызвать две функции — сравнения строк и сравнения чисел. Первая есть в библиотеке Си (см. Приложение 2), это `strcmp()`. Вторую придётся написать, обозначим её как `sravn_chisel()` (и добавим — пусть она возвращает 0, если значения одинаковы, 1 — если значение запроса меньше, и -1 — если значение запроса больше). Означенные вызовы могут выглядеть так:

```
poisk(sravn_chisel, &chislo);
poisk(strcmp, "Джон Смит");
```

ё) Работа с предобработчиком

1) Подстановка переменных. Указание предобработчика `#define` умеет работать с переменными в скобках, за которыми скрывается некоторое выражение. Например:

```
#define polovina(x) (x)/2
```

Тогда в коде каждое вхождение, к примеру, `polovina(a-3)`, будет заменено на `(a-3)/2`. Обратите внимание — скобки остаются независимыми, `x` заменяется на `a - 3`. Использовать именно `x` в качестве имени переменной необязательно.

2) Указание `#undef`. Используется, когда необходимо для дальнейшего участка кода прекратить работу указания `#define` для того или иного обозначения. Пример:

```
#undef BUFER
```

3) Указание `#ifdef`. Является оператором условия; проверяет, находится ли участок кода, в котором оно записано, под прицелом указания `#define` для того или иного обозначения. Если да — оставляет в программе участок кода под веткой "истина" и удаляет код под веткой "ложь", если нет — наоборот.

Например, предположим, что обозначению `BUFER` через указание `#define` задана замена на `100`, а нам требуется, чтобы на каком-то участке оно начало заменяться не на 100, а на `256`. Тогда поможет `#ifdef`.

```
#ifdef BUFER
    # undef BUFER
    # def BUFER 256
#else
    # def BUFER 256
#endif
```

Также существует указание `#ifndef`, то есть "if not defined" — "если данное обозначение не заменяется". Пример:

```
#ifndef SLOVO
    ...
# else
    ...
```

```
#endif
```

И ещё есть указание `#if`.

```
#if выражение
    ...
#endif
```

Если записанное после него выражение[1] выдаёт "истину" (не ноль), то код программы после условия будет оставлен, в противном случае будет выброшен из программы.

ж) Область действия переменных

Переменную можно объявить в начале любого составного действия. В этом случае она действует только внутри него. Любая переменная с таким же именем, объявленная выше, на время выполнения этого составного действия недоступна, и снова начинает работать после его окончания.

```
int y = 1, x;
for (x = 0; x < 99; x++)
{
    int y = 0;
    y++;
}
```

В этом примере значение `y` при выходе из цикла — 1.

з) Функция "sizeof()"

Выдаёт объём поданной в неё переменной в байтах. Например:

```
int a;
a = sizeof(a);
```

На компьютере с 8-битным байтом и 16-битным "целым числом" будет выдано значение 2.

Наиболее очевидное использование этой функции — распределение места для чего бы то ни было.

и) Объединения

Объединением называется структура, в которой все переменные начинаются с одного и того же адреса. Ячейки переменных таким образом пересекаются — в зависимости от одинаковости \ неодинаковости видов данных полностью или частично, — и в этих ячейках переменные пользуются одними и теми же значениями.

Например, в разделе **е)** выше указатель `zapros` может указывать как на "целое число", так и на "знаки ввода". Применённый там способ — сделать `zapros` указателем на "знаки ввода" — подходит в обоих случаях; если произойдёт указание на "целое число" — будет прочитано соответствующее количество байт.

Но такой подход небезопасен. Пример: приращение адреса указателя `zapros++` произве-

1 Выражение может быть составлено только либо из чисел, либо из обозначений, которые заменяются на числа указанием `#define`. Допустимые действия в выражении — арифметические и сравнительные. (прим. перев.)

дёт увеличение адреса только на один байт, что конечно же неправильно, если работа производится с "целым числом".

Написание объединения очень похоже на написание структуры.

```
union
{
    int *tseloe;
    char *znakvv;
} zapros;
```

То бишь переменная `zapros` в разделе е) будет объявлена (в режиме "либо-либо") указателем на "целое число", либо указателем на "знаки ввода". Ссылка на одно или другое делается так же, как и ссылка на поле в структуре:

```
zapros.tseloe
zapros.znakvv
```

ПРИЛОЖЕНИЕ 2

Краткий справочник-напоминалка

1. Действия

а) С одним участником

Обозначение	Смысл	Примеры	Описание
*	сделать указателем	`a = *u;`	`a` получает значение, на которое указывает `u`
&	извлечь адрес	`u = &a;`	`u` получает адрес `a`
-	отрицательное значение	`b = -a;`	`b` получает значение `-a`
!	обратить смысл выражения	`if(b != 3)`	если значение `b` не равно 3-м ...
		`a = !b;`	если значение `b` — "ложь" (0), то `a` получает значение "истина" (1), иначе — "ложь" (0)
~	преобразование "НЕ"[1]	`a = ~b;`	биты `b` обращаются и записываются в `a`
++	приращение	`b++;`	значение `b` увеличивается на 1
		`if(s[a++]='')`	проверка равенства значения в массиве с пробелом, затем приращение `a`
		`if(s[++a]='')`	приращение `a`, затем проверка равенства значения в массиве с пробелом
--	отрицательное приращение	`b--;`	значение `b` уменьшается на 1

б) С двумя участниками

Обозначение	Смысл	Примеры	Описание
*	умножение	`v = a * b;`	умножить `a` на `b` и записать итог в `v`
/	деление	`v = a / b;`	разделить `a` на `b` и записать итог в `v`
%	остаток от деления	`v = a % b;`	разделить `a` на `b` нацело, остаток записать в `v`
+	сложение	`v = a + b;`	сложить `a` и `b` и записать итог в `v`
-	вычитание	`v = a - b;`	выполнить вычитание `a - b` и записать итог в `v`
<<	сдвиг влево	`v = a << 3;`	сдвинуть значение `a` на 3 бита влево и записать итог в `v`

1 Преобразование "НЕ" также обозначается как "обратный код" (имеется в виду двоичный код числа, и более точным было бы понятие "обращённый код"), или "one's complement" \ "1's complement" в англоязычной литературе. (прим. перев.)

Обозначение	Смысл	Примеры	Описание
>>	сдвиг вправо	`v = a >> 2;`	сдвинуть значение `a` на 2 бита вправо и записать итог в `v`
&	преобразование "И"	`v = a & b;`	выполнить преобразование "И" для `a` и `b` и записать итог в `v`
^	исключающее "ИЛИ"	`v = a ^ b;`	выполнить преобразование "исключающее ИЛИ" для `a` и `b` и записать итог в `v`
|	"ИЛИ"	`v = a | b;`	выполнить преобразование "ИЛИ" для `a` и `b` и записать итог в `v`
&&	"и" (логическая связка)	`if(a!=0 && a < 5)`	если значение `a` не равно 0, а также меньше 5 ...
||	"или" (логическая связка)	`if(a<3 || a > 6)`	если значение `a` меньше 3 либо больше 6 ...
<	меньше	см. предыдущ.	
<=	меньше либо равно	`if (a <= 120)`	если значение `a` меньше либо равно 120 ...
>	больше	см. выше	
>=	больше либо равно	см. "меньше либо равно"	
==	равно (сравнение)	`if (a == b)`	если значение `a` равно значению `b` ...
? :	проверка условия	`g = a?b:v;`	если значение `a` — не "ложь" (0), то `g` получает значение `b`, иначе — получает значение `v`
,	действие "запятая"	`v = a++ , b;`	действия выполняются слева направо, затем возвращается значение второго действия. Здесь `a` приращается, а `v` получает значение `b`
->	указатель на структуру	`u -> mes`	если `u` — указатель на структуру, то он указывает на поле `mes` в структуре
.	выбор поля в структуре	`data.mes`	обращение к полю `mes` в структуре `data`

в) Присвоение значения

Знак = можно читать как "получает значение". Перед ним можно записывать любой знак действий с двумя участниками (см. выше). Например,

```
a += 5;
```

равнозначно

```
a = a + 5;
```

Ещё пример:

```
a <<= 3;
```

означает то же, что и

```
a = a << 3;
```

2. Виды данных

Вид	Примечания
`char` ("знак ввода")	Как правило (необязательно), имеет объём 8 бит.
`int` ("целое число")	Арифметика считается в дополнительном коде[1]. Дополнительные слова — `short` ("малой величины") и `long` ("большой величины").
`unsigned` ("беззнаковое целое")	Дополнительные слова — `short` ("малой величины") и `long` ("большой величины").
`float` ("с плавающей точкой")	
`double` ("с плавающей точкой удвоенной точности")	
`struct` ("структура")	Возможность задать пользовательское имя для обращения к одному \ нескольким полям любых видов данных.
`union` ("объединение")	Возможность читать одни и те же ячейки как разные виды данных.

Примеры:

`char znak, bukva;`	объявлены переменные `znak` и `bukva`, имеющие вид "знак ввода"
`int a, b, v;`	объявлены 3 переменные вида "целое число"
`unsigned chislo;`	объявлена переменная вида "беззнаковое целое". Слово `int` писать необязательно
`int a[10],b[5][7];`	объявлен одномерный массив `a` из 10 ячеек, затем — двумерный массив `b` с таблицей ячеек 5x7
`char *uk;`	объявлен указатель на "знак(и) ввода"
`int *uk[50];`	объявлен массив указателей на "целые числа"
`char **uk[20];`	объявлен массив указателей на указатели на "знак(и) ввода"

`struct data`
`{`
`int den;`
`int mes;`
`int god;`
`char nazv_mes[3];`
`};`
	объявлена структура, состоящая из 3-х "целых чисел" и одного массива "знаков ввода"

`struct data d1;`	объявлена структура, имеющая "видом данных" уже имеющуются структуру
`struct data m[25];`	объявлен массив из структур

1 Дополнительный код (в англояз. литературе "two's complement" и "2's complement") — способ представления отрицательных чисел, позволяющий заменить действие вычитания действием сложения. Благодаря этому используется для арифметики с "беззнаковым" видом данных.

Чтобы преобразовать отрицательное число в двоичном коде в дополнительный код, требуется обратить его биты (преобразование "НЕ") и увеличить получившееся двоичное значение на единицу; дополнительный код неотрицательного числа совпадает с обычным кодом. (прим. перев.)

3. Режимы хранения переменной в памяти

Режим	Описание
`auto`	переменная действует в пределах функции, в которой она объявлена
`static`	значение переменной сохраняется между вызовами функции, содержащей её (сохраняется всё время работы программы). Переменная может действовать в пределах функции, в которой она объявлена, либо в пределах файла, где она объявлена в заголовке
`extern`	указание компилятору, что переменная объявлена в другом файле исходного кода, и в этом файле искать её объявление не требуется. Значение переменной сохраняется всё время работы программы
`register`	если возможно, переменная хранится в регистре процессора. В остальном тождественен режиму `auto`

4. Действия

Простейшие действия всегда завершаются точкой с запятой ; .

Составные действия представляют собой последовательность простейших, заключённую в фигурные скобки `{}` , и могут быть написаны везде там же, где могут быть простейшие действия.

Запись	Примеры	Примечания

if (выраж)дейст	`if (a > 3)`
	`{`
	`  b++;`
	`  perem = 1;`
	`}`
		действие выполняется, если выражение имеет значение "истина" (отличное от 0)

if (выраж)
  дейст1
else
  дейст2
	`if(a == 0)`
	`  b = c;`
	`else`
	`  b += 2;`
		если выражение имеет значение "истина", то выполняется действие 1, в противном случае — действие 2

while (выраж)дейст	`while (a[b])`
	`{`
	`  b++;`
	`  c--;`
	`}`
		действие выполняется до тех пор, пока выражение не примет значение "ложь". Проверка условия (выражения) происходит перед действием

do дейст
...
while (выраж)
	`do`
	`  a += 2;`
	`while (b[a])`
		действие выполняется до тех пор, пока выражение не примет значение "ложь". Проверка условия (выражения) происходит после действия

for (выраж1; выраж2; выраж3) дейст	`for(a=1;a<8; a++)` `  b[a] = 0;`	Выражение 1 выполняет самое первое действие перед началом цикла. Сразу после "действия" при каждом проходе выполняется выражение 3. Выражение 2 является условием
break;	`break;`	выход из цикла или переключателя, в котором находится это слово
continue;	`continue;`	переход к началу цикла (включая проверку условия), в котором находится это слово

Запись	Примеры	Примечания

switch (выраж)дейст	`switch(a)`
	`{`
	`case -1:`
	`b=4;`
	`break;`
	`case 0:`
	`b = -2;`
	`break;`
	`default:`
	`b++`
	`}`
		Значение "выражения" указывает на значение, соответствующее ветке выбранного действия. После выбора "действие" выполняется до слов `break` или `return`.
		Ветка `default` необязательна

return выраж;	`return perem;`	передача управления в вызывавшую функцию. "Выражение" необязательно, но если оно есть — его значение также передаётся в вызвавшую функцию
goto метка;	`goto metka;`	передача управления действию, обозначенному указанной меткой (метка указывается перед целевым действием в следующем виде: "метка: действие")

5. Указания предобработчика

`#include <имяФайла>`	включает в исходный код содержимое указанного файла
`#define pervoe vtoroe`	заменяет в исходном коде любое первое вхождение вторым
`#if выражение`	условие. Выполняется, если значение выражения — "истина"
`#ifdef obozn`	условие. Выполняется, если указанию `#define` задано заменять указанное обозначение
`#else`	действие "иначе", используемое в условии
`#endif`	требуется для завершения написания условия
`#undef obozn`	прекращает действие указания `#define` по замене указанного обозначения

6. Постоянные значения

а) Целые

десятичное — последовательность цифр без ноля на первом месте. Пример — `5729`

8-ричное — последовательность цифр с нолём на первом месте. Пример — `0571`

16-ричное — последовательность цифр, начинающаяся с `0x`. Пример — `0x5fa3`

б) С плавающей точкой

обыкновенный вид `321.85`

экспоненциальный вид `3.2185e2`

в) Знаки ввода

Одиночные знаки ввода берутся в одинарные кавычки. Пример:

```
znak = 'a';
```

Байт `znak` получает значение знака `a` в коде "ASCII".

Строковые значения берутся в двойные кавычки, например "`строковое значение`".

Некоторые служебные знаки ввода обозначаются условными сочетаниями с косой чертой.

перевод строки	`\n`
горизонтальная табуляция	`\t`
ввод косой черты "\"	`\\`
удаление предыдущего знака	`\b`
одинарная кавычка	`\'`
возврат каретки	`\r`
8-ричный номер знака	`\xxx` (где `xxx` — цифры)

7. Часто используемые библиотечные функции

Функция	**Примечания**
Терминальный ввод \ вывод	
`getchar()`	возвращает следующий знак ввода из буфера потока ввода по умолчанию (клавиатура)
`putchar(a)`	выводит указанный знак ввода на поток вывода по умолчанию (экран)
Файловый ввод \ вывод	
`fopen(имяФайла, режим)`	открывает поток ввода \ вывода для указанного файла: `r` — в режиме чтения, `w` — в режиме (пере)записи, `a` — в режиме дозаписи. "Имя файла" и "режим" должны быть указателями на строковые значения нужного содержания. Функция возвращает номер потока ввода \ вывода, либо 0 в случае ошибки
`fclose(поток)`	очищает буфер указанного потока ввода \ вывода и закрывает его
`getc(поток)`	возвращает один знак с потока ввода, либо признак конца файла `EOF`
`putc(a, поток)`	поданный в функцию знак выводится на указанный поток вывода
`fflush(поток)`	очищает буфер указанного потока
`unlink(имяФайла)`	удаляет указанный файл
`fseek(поток,номБайта,откуда)`	устанавливает головку жёсткого диска на указанный порядковый номер байта: от начала файла, если `откуда` = 0; от текущего положения в файле, если `откуда` = 1; от конца файла, если `откуда` = 2.
Упорядоченный ввод \ вывод	
`scanf(строка, знач1, знач2, ...)`	читает данные с потока ввода и записывает их по адресам, предоставляемым "значениями", в виде, определённом входной строкой
`fscanf(поток,строка,знач1, знач2, ...)`	подобна функции `scanf()`, но данные читаются с указанного потока
`printf(строка,знач1,знач2, ...)`	выводит перечисленные значения на поток вывода в виде, определённом входной строкой
`fprintf(поток, строка, знач1, знач2, ...)`	выводит перечисленные значения на указанный поток в виде, определённом входной строкой

Функция	Примечания
Преобразование данных	
`atoi(строка)`	возвращает цифру(-ы), прочитанную(-ые) из входной строки, в виде "целого числа"
`sprintf(буфер, строка, знач1, знач2, ... )`	подобна функции `printf()`, но выводит данные в указанный буфер
`sscanf(буфер, строка, знач1, знач2, ... )`	подобна функции `scanf()`, но читает данные из указанного буфера
Обработка строковых значений	
`strcat(основная, присоединяемая)`	присоединяет одну строку к другой
`strcmp(строка1, строка2)`	возвращает отрицательное значение, 0 или положительное значение, если — соответственно — первый отличающийся знак строки 1 по своему коду меньше чем знак на этом же месте в строке 2, если строки одинаковы, и если знак в 1-й строке больше знака во 2й строке
`strcpy(массив, строка)`	записывает введённую строку в указанный массив
`strlen(строка)`	возвращает длину строки, на которую указывает входной указатель
Обработка знаков ввода	
`isalpha(a)`	возвращает 1, если поданный знак — это буква
`isdigit(a)`	возвращает 1, если поданный знак — цифра
`islower(a)`	возвращает 1, если поданный знак — строчный
`isupper(a)`	возвращает 1, если поданный знак — заглавный
`isspace(a)`	возвращает 1, если поданный знак — пробел, табуляция или перевод строки
Преобразование знаков ввода	
`tolower(a)`	если поданный знак — заглавный, то возвращает этот же строчный знак
`toupper(a)`	противоположна предыдущей
Прочее	
`abs(a)`	возвращает поданное целое число в неотрицательном виде
`calloc(число, объёмВбайтах)`	распределяет и заполняет нолями в памяти объём, равный произведению входных значений. Возвращает указатель на образовавшееся пространство, либо 0, если в памяти недостаточно места
`free(указатель)`	освобождает от использования пространство памяти, распределённое через функцию `calloc()`. На вход принимает указатель на это пространство
`exit(a)`	возвращает управление операционной системе. Подаваемое значение — как правило 0, но можно вводить 1, чтобы обозначить завершение с ошибкой