А. Крупник

ИЗУЧАЕМ Си

ПИТЕР®
Санкт-Петербург • Москва • Харьков • Минск
2001

Содержание

Глава 5. Функции, указатели, массивы

Глава 6. Файлы

Глава 7. Строки

Глава 8. Основные типы на сборочном конвейере

Глава 9. Большие программы

Глава 10. Макросы и переходы

Предисловие

Казалось бы, не так важно, какой язык программирования учить первым. Зная один язык, легко выучить любой другой. Но первый язык становится «родным» и для освоения других понадобятся лишние усилия. Так зачем же начинать с устаревшего Бейсика?

В этой книге делается попытка познакомить читателя с программированием на примере языка профессиональных программистов Си. Этот язык не только важен сам по себе, но и открывает дорогу к другим современным и очень популярным языкам, таким как C++, JAVA, PERL и JavaScript.

Язык Си считается трудным для изучения, и это отчасти так, если его учить после Бейсика, Фортрана или даже Паскаля. Но если Си станет первым изучаемым языком, все будет гораздо проще.

Пусть читателя не пугает, что после короткого введения (Глава 1 «Введение в программирование»), мы сразу перейдем к описанию среды программирования Turbo C (глава 2 «Первые шаги») и к простейшим программам на Си. Такой порядок изложения только облегчит его первые шаги. Уже в главе 3 «Странные типы» начинается знакомство с указателями — трудной, но крайне важной для понимания языка темой. Это знакомство углубляется на протяжении всей книги, и к ее концу указатели должны стать родными читателю, как становится родным язык, который окружал человека с детства.

Эта книга задумана и написана с целью рассказать пытливому читателю о самых трудных, самых принципиальных особенностях языка, что, конечно, не значит, что она не учит практическому программированию на Си. Книга построена на простых коротких примерах, которые помогут начать программировать уже с первых ее страниц.

Мне хотелось, чтобы эта книга стала одной из первых в длинном ряду других книг по программированию, операционным системам и алгоритмам, которые нужно прочитать, чтобы стать программистом-профессионалом. Насколько исполнилось это желание — судить читателю.

Александр Крупник
krupnik@sandy.ru
http://www.piter.com/display.phtml?a_id=19312&web_ok=all
Нижний Новгород, 16 июля 2001 года.

Глава 1. Введение в программирование

Программы

Чтобы научиться программировать, нужно понимать, как работает компьютер. А для этого совсем не обязательно изучать прохождение электрических сигналов по микросхемам. Достаточно вообразить ряд пронумерованных ячеек, в каждой из которых может храниться число или команда.

Представим себе маленького чумазого мальчика (назовем его *процессором*), который выполняет команды, хранящиеся в последовательно расположенных ячейках. Чтобы начать работу, мальчику необходимо знать, в какой ячейке хранится первая команда.

Предположим, что первая команда всегда хранится в одной и той же ячейке, например, под номером 37. Открывая ячейку (вообразим, что это ящичек с дверцей), мальчик достает оттуда лист бумаги, на котором записана команда: «поместить в ячейку 1 число 2». Выполнив первую команду (написав на клочке бумаги требуемое число и бросив его в первую ячейку), мальчик переходит к ячейке 38, где записана следующая команда: «поместить число 3 в ячейку 2». После ее выполнения в ячейке 2 оказывается число 3. Далее мальчик переходит к ячейке 39, где содержится команда посложнее: «прибавить к содержимому первой ячейки число из второй и поместить результат в ячейку номер 3». Чтобы выполнить эту команду, мальчику придется открыть первую и вторую ячейки, достать оттуда числа, сложить их и поместить результат (число 5) в третью ячейку. Выполнив эту команду, мальчик переходит к следующей, 40-й ячейке, где записана команда «СТОП». Увидев ее, он заканчивает работу и ждет. Команды и результаты их выполнения показаны в табл. 1.1.

Таблица 1.1. Простейшая программа и результат ее выполнения на каждом шаге

Ячейка	Команда	Ячейки памяти
		1-? 2-? 3-?
37	Поместить в ячейку 1 число 2	1-2 2-? 3-?
38	Поместить в ячейку 2 число 3	1-2 2-3 3-?
39	Сложить содержимое ячеек 1 и 2, результат записать в ячейку 3	1-2 2-3 3-5
40	СТОП	

Только что выполненная мальчиком последовательность действий, называется *программой*. И хотя ее результат — сумма двух чисел — смехотворен, в ней есть многое из того, что необходимо знать начинающему программисту. Становится, например, понятно, что программы и данные хранятся в одной и той же памяти (последовательности ячеек). Чтобы их различать, нужно заранее знать, где начинается программа и где можно хранить данные. Нельзя записать число 2 в ячейку 38, потому что это испортило бы программу. Программа выполняется последовательно по шагам. Выполнив команду из 37-ой ячейки, мальчик переходит к следующей, 38-й и так до тех пор, пока ему не встретится команда СТОП.

Правда, естественный порядок команд может нарушаться, как, например, в программе, показанной в табл. 2, при следующем исходном состоянии ячеек:

```
Ячейка      Содержимое
2           5
5           'П'
6           'Р'
7           'И'
```

```
8           'B'
9           'E'
10          'T'
11          '0'
```

Таблица 1.2. Вторая программа

Ячейка	Команда
37	Прочитать содержимое ячейки, *адрес* которой хранится в ячейке 2
38	Если оно равно 0, перейти к ячейке 42
39	Напечатать содержимое ячейки, адрес которой хранится в ячейке 2
40	Увеличить содержимое ячейки 2 на 1
41	Перейти к ячейке 37
42	СТОП

Самая трудная команда в этой программе, — конечно же, первая. Она состоит из нескольких действий. Первое — чтение содержимого ячейки 2. Прочитав число 5 из ячейки 2, мальчик знает, что это совсем не та пятерка, которую он получил в результате выполнения первой программы, а *адрес* некой ячейки памяти. Такие адреса в языке Си называются *указателями*. Второе действие уже проще: мальчик читает содержимое ячейки 5 и переходит к следующей команде, записанной в ячейке 38. Здесь проверяется, равно ли нулю содержимое пятой ячейки. Если да, мальчик переходит к 42-й ячейке и останавливается. Если нет, — рисует мелом на грифельной доске букву, хранящуюся в пятой ячейке, увеличивает адрес в ячейке 2 на единицу, и переходит к началу программы, то есть к ячейке 37.

Ячейки и типы

В предыдущем разделе наш мальчик-процессор, когда ему нужно было поместить в ячейку какое-то число, просто писал его на бумажке и клал ее в соответствующий ящик. Как вы понимаете, на самом деле все устроено иначе. Ячейки реального компьютера не безразмерны, и в них можно поместить числа, не превышающие, скажем, 256 или 65535. Происходит это потому, что ячейка компьютера — не пустой ящик, куда бросают бумажки, а набор из нескольких переключателей, каждый из которых находится в одном из двух состояний: верхнем (его часто обозначают единицей) или нижнем (его обозначают нулем). В ячейке машины таких переключателей, как правило, восемь (каждый переключатель называют *битом* и говорят, что в ячейке 8 бит или 1 *байт*). То, что мог бы увидеть мальчик, открыв дверцу ячейки, показано на рис. 1.1

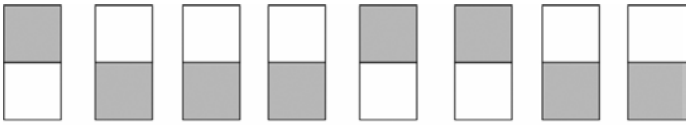

Рис. 1.1. Внутренности ячейки

Чтобы не рисовать каждый раз внутренности ячейки, положение переключателей обозначают цифрами. Содержимое ячейки, показанное на рис. 1.1, может быть представлено в виде последовательности нулей и единиц:

```
1 0 0 0 1 1 0 0
```

Попробуем подсчитать, сколько разных чисел способна вместить такая ячейка. Для этого вообразим ячейку, в которой всего один переключатель с двумя состояниями: 1 и 0 (ВЕРХ и НИЗ). Значит, в такой ячейке могут храниться два числа. Если теперь добавить к ячейке еще один переключатель, число ее состояний удвоится: теперь на каждое состояние первого переключателя приходится два состояния второго. Значит, в ячейке из двух переключателей всего 4 состояния, в ячейке из

трех переключателей — 8 (для каждого из четырех состояний, в которых может находиться ячейка из двух переключателей, есть два состояния третьего переключателя), а в ячейке из восьми переключателей (битов) число возможных состояний уже 2×2×2×2×2×2×2×2=256.

Значит, в ячейку из восьми бит можно записать числа от 0 до 255 (всего 256). Чтобы это сделать, можно, например, пронумеровать все состояния ячейки и считать, что 57-е состояние соответствует числу 57. Но проще воспользоваться позиционным представлением чисел, с объяснения которого начинается школьный курс информатики. Суть его в том, что каждому биту приписывается определенный вес: крайний правый бит это 1(2^0), второй справа бит — 2 (2^1),третий справа — 4 (2^2), четвертый — 8 (2^3), пятый — 16 (2^4), шестой — 32 (2^5), седьмой — 64 (2^7), восьмой — 128 (2^8). Как видим, вес каждого бита — это двойка, возведенная в степень, показатель которой равен расстоянию от правого края числа. Первый справа бит находится на нулевом расстоянии и поэтому его вес равен 2^0. Вес второго справа бита равен 2^1 и т.д. Чтобы понять, какое число записано в ячейке, нужно просуммировать произведения весов на значения самих битов. Так, число 10000001 равно

$1*2^8+0*2^7+0*2^6+0*2^5+0*2^40*2^3+0*2^2+0*2^1+1*2^0$= 128+1=129,

а число 01010101 равно 64+16+4+1=85.

Когда в предыдущем разделе нашему мальчику нужно было поместить в ячейку 1 число 2, он, открыв дверцу ячейки, видел перед собой не просто ящик с клочком бумаги, а набор переключателей, положение которых может быть любым (говорят, что в этом случае ячейка содержит «мусор»). Задача мальчика (поместить число 2 в ячейку 1) состоит в том, чтобы установить второй справа переключатель в верхнее положение, все же остальные — в нижнее. Это и будет записью числа 2 в ячейку 1. В результате выполнения первой программы в ячейке 3 окажется число пять, то есть 4+1: 00000101.

Теперь мы понимаем, что внутренности ячейки памяти выглядят всегда как последовательность нулей и единиц,

независимо от того, что в них находится: число, буква или адрес другой ячейки. Значит, для правильного выполнения программы процессор должен знать, *данные какого типа* хранятся в ячейках, и в соответствии с этим выполнять свою работу.

Наш воображаемый мальчик узнавал о типе хранящихся в ячейке данных из самой программы. Если он встречал команду «поместить *число* 2 в ячейку 1», то устанавливал в верхнее положение второй переключатель справа, остальные же устанавливал в нижнее (или нулевое) положение. При выполнении команды «поместить *знак* '2', в ячейку 1» мальчик поместит в ячейку число 50, то есть 00110010, потому что знаки представляются в машине не так, как цифры.

Итак, в ячейках памяти есть только двоичные цифры, но в зависимости от того, что (какой тип данных) хранится в ячейке, эти цифры будут разными. С каждой ячейкой памяти связан определенный тип.

Программирование и трансляторы

Не знаю, как вам, а мне уже становится жаль нашего мальчика — ведь ему приходится бегать от ячейки к ячейке, читать сначала команды, потом обращаться к ячейкам с данными, снова переходить к командам. Возможно, малый рост не позволяет ему добраться до всех ячеек, и ему приходится двигать тяжелую приставную лестницу. И все это без малейшего перерыва, пока не кончится программа.

Впрочем, жалость здесь неуместна — ведь никаких мальчиков, на самом деле не существует. Вместо воображаемого мальчика всю черную работу выполняет *процессор* — кремниевая пластина размером с пару коробков, на которой расположены миллионы транзисторов. За одну секунду процессор в состоянии выполнить сотни миллионов простых операций: записи/считывания чисел из ячеек памяти, сложений, вычитаний и т.д. Процессор способен работать годы напролет, ему не требуются выходные и перерывы на обед, и всю свою жизнь он не расстается со своей «мамой» — материнской платой (рис. 1.2).

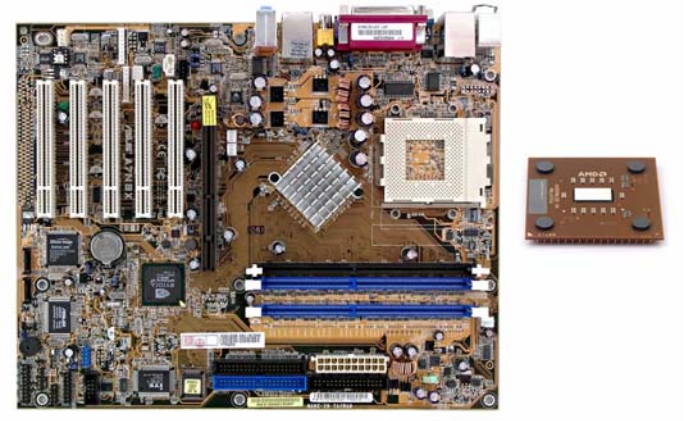

Рис. 1.2. Процессор и его «мама»

Жалеть на самом деле нужно не того, кто бездумно исполняет программы, а того, кто их пишет, то есть программистов. Ведь им приходится выделять ячейки, где будут храниться данные и программа. Им нужно помнить, что в первой ячейке записано число 2, а во второй — 3. Им же необходимо заботиться о том, чтобы данные не «налезли» на программу. А если это все-таки произойдет, придется не только перемещать программу в другую область памяти, но и менять в ней некоторые команды. Вспомним, что во второй нашей программе была команда :

```
38 Если содержимое ячейки равно '0',
перейти к ячейке 42
```

Если предположить, что программа будет перенесена в другое место памяти, придется менять эту команду, потому что переход уже будет не к 42-й, а какой-то другой ячейке. Менять программу придется и тогда, когда между 37-й и 42-й ячейками будет вставлена дополнительная команда.

Иными словами, работа непосредственно с ячейками памяти и командами процессора требует от программиста огромного труда и чудовищного внимания. Программист не может дать имена содержимому ячеек, например, назвать ячейку 3 из первой нашей программы `sum`, чтобы показать, что в ней будет записана сумма чисел. Ему очень трудно менять что-то в

программе, потому изменение программы в одном месте неизбежно потребует изменений во многих ее местах, а это значит, что программисту очень трудно будет избежать ошибок. Первые, сейчас уже легендарные, программисты хлебнули горя, ведь им приходилось не только распределять вручную память, но и писать каждую команду в двоичном коде — нулями и единицами! Им приходилось помнить, что в ячейке 3 находится сумма, а, скажем, в ячейке 4 — начало фразы. Им приходилось при поиске ошибок проверять двоичные коды программ, и когда находилась ошибка, менять сразу несколько команд, что в свою очередь приводило к новым ошибкам.

Поэтому очень скоро сами программисты стали писать программы иначе: вместо номеров ячеек они вводили обычные имена, например, `first` (первый), `second` (второй), `sum` (сумма), а сами команды процессора записывали не двоичными цифрами, а обычными буквами. Программы из последовательностей нулей и единиц превратились в тексты, которые можно создавать в обычном редакторе. Такую программу гораздо легче было понять и исправить программисту, но ее уже не понимал процессор. Поэтому перед исполнением программы, написанной на языке, удобном для программистов, ее нужно было пропустить через специальную служебную программу, так называемый *ассемблер*, которая переводила машинные команды, записанные буквами, в последовательности нулей и единиц, понятные машине.

Ассемблеры были огромным шагом вперед в программировании, но они только иначе записывали команды процессора. Ассемблерная программа (условные обозначения команд и ячеек, применяемые программистами, называются *языком ассемблера*) могла работать на компьютере только одного типа. Исполнить ее на процессоре другого типа с другой системой команд было невозможно.

Поэтому были изобретены так называемые *языки высокого уровня* (такие, например, как BASIC или PASCAL), программы на которых могут быть исполнены самыми разными компьютерами. На самом деле, такие программы не могут быть непосредственно выполнены ни на одном компьютере,

который, как мы это уже знаем, понимает только команды собственного процессора. Поэтому программа, написанная на языке высокого уровня, испытывает перед исполнением несколько превращений. Сначала специальная *программа-компилятор* переводит ее в ассемблерный код, а затем уже *ассемблер* превращает ее в последовательность машинных команд. То есть теперь, вместо того, чтобы каждый раз писать программу на машинном языке, нужно один раз помучиться и написать на этом языке компилятор и ассемблер, а затем писать программы на языке высокого уровня, что ускоряет процесс программирования, делает его гораздо более надежным и, что самое главное, написанные таким образом программы, могут быть легко перенесены на другие компьютеры, с другими командами процессора.

Прочитав эту книгу, вы научитесь писать программы на языке Си — одном из самых популярных и мощных. Язык Си был создан в начале 70-х годов прошлого века Деннисом Ритчи и Кеном Томпсоном. К моменту создания языка Си эти легендарные люди уже были опытными программистами, и хорошо знали, каким должен быть новый язык. Быть может, поэтому Си стал таким популярным. За тридцать лет язык Си существенно изменился, но и сейчас многое в нем отражает неповторимые личности создателей — их любовь к простоте и краткости, их своеобразный юмор. Си — язык профессиональных программистов. Это значит, что освоить его чуть труднее, чем Бейсик или Паскаль. Некоторые конструкции языка Си неожиданны и требуют определенных усилий для понимания. Но, приложив эти усилия, вы будете щедро вознаграждены.

Первая программа на Си

Наша первая программа на Си, как и первая программа на языке ячеек, с которой мы познакомились в начале этой главы, складывает два числа (см. листинг 1.1).

Листинг 1.1

```
main() {
```

```
int first, second, sum;
first=2;
second=3;
sum=first+second;
}
```

Слово `main`, которое есть абсолютно в любой Си-программе, показывает компилятору, откуда она начинается. Следом всегда идут открывающая и закрывающая скобки `()`, внутри которых могут быть дополнительные параметры, но сейчас они нам ни к чему. Сама программа помещается между фигурными скобками `{}`, которые также обязательны. Строчка

```
int first, second, sum;
```

командует компилятору выделить ячейки памяти для переменных `first`, `second` и `sum`. Имена переменных в языке Си начинаются обязательно с латинской буквы, далее могут идти как буквы, так и цифры. Строчные буквы отличаются от прописных, поэтому `Sum` и `sum` — разные имена.

Словечко `int` перед именами переменных показывает, что в выделенных ячейках памяти будут храниться целочисленные переменные, которые могут быть только положительными и отрицательными целыми и нулем. Три последующие строчки

```
first=2;
second=3;
sum=first+second;
```

в отличие от предыдущих, где только объявлялись переменные и выделялась память для них, содержат *инструкции*, то есть описания *действий*, которые должен выполнить компьютер.

Две строчки

```
first=2;
second=3;
```

засылают в переменные `first` и `second` нужные значения. Целочисленным переменным можно присваивать и дробные значения, например

```
second=3.5;
```

Результат (присвоенное значение переменной) все равно будет целым, то есть переменная `second` окажется равной 3. Это означает, что знак равенства в языке Си используется не так, как в алгебре. Запись `second=3.5` означает в алгебре равенство переменной **`second`** числу 3,5. Такая же запись в языке Си означает присваивание переменной **`second`** значения 3.5. Как мы поняли, результат присваивания зависит от типа переменной.

Для завершения знакомства с первой программой на языке Си нам осталось только сказать о точке с запятой, которая ставится в конце каждой инструкции. Именно точка с запятой показывает компилятору, что текущая инструкция закончилась и пора переходить к следующей. Это, между прочим, означает, что инструкция может занимать несколько строчек. Например, та же самая инструкция может выглядеть и так:

```
sum=

first

+

second;
```

Язык Си предоставляет программисту большую свободу и не навязывает ему какой-то определенный стиль. Главное — разумно пользоваться этой свободой и писать программы так, чтобы их легко могли читать другие. Но об этом — в следующих главах.

Глава 2. Первые шаги

Turbo C — среда программирования

А потом, когда вроде бы все затихло, я вдруг всеми своими кодами почувствовал, что меня рассматривают. Мерзопакостное это ощущение, когда ты сам сделать ничего не можешь, а неизвестно кто, пользуясь этим, пытается заглянуть за каждую твою команду, изучить заголовок. А потом произошло что-то необыкновенное — Я БЫЛ БРОШЕН В ЖИЗНЬ. Тот, кто меня рассматривал, вдруг, обратившись молитвами к могучему ДОСу, схватил меня за заголовок и вытащив из могилы забвенья , швырнул прямо в мир процессора.

С.Расторгуев «Программные методы защиты информации в компьютерах и сетях»

Наша первая программа на языке Си, с которой мы познакомились в предыдущей главе (см. листинг 1.1), вышла настолько простой, что ее безо всякого компьютера может выполнить даже первоклассник.

Но бывают программы настолько сложные, что с ними не справятся и миллионы первоклассников, даже если они будут заниматься вычислениями все время, оставшееся от беганья с бешеными криками по школьным коридорам.

Сложные программы может выполнить только компьютер. Как мы уже говорили, программа, написаная на языке, удобном для программиста, должна быть пропущены через компилятор, преобразующий ее в последовательность команд процессора.

Из великого множества компиляторов, которые используются на компьютере IBM PC, для изучения языка Си мы выберем компилятор Turbo C 2.01 фирмы Borland. Эту программу, не будь она классической, можно было бы назвать древней. Turbo C 2.01 распространяется бесплатно, найти ее можно по

адресу
http://community.borland.com/article/images/20841/tc201.zip

Установка компилятора очень проста. Прежде всего нужно распаковать архивный файл tc201.zip. Для этого (предварительно перейдя в папку, где хранится этот файл) наберите в командной строке файловой оболочки[1]

```
pkunzip -d tc201.zip
```

и нажмите Enter.

После распаковки архива в той папке (директории), где он находился, появятся три новых папки — DISK1, DISK2, DISK3. Для установки компилятора можно просто переписать содержимое папок DISK2 и DISK3 в папку DISK1 и затем запустить находящуюся в ней программу install.exe. Далее программа установки спросит вас название диска, с которого устанавливается компилятор. В зависимости от того, на каком логическом диске находится папка DISK1, нужно ввести одну букву — 'C', 'D', 'E' или другую. Далее появится меню, в котором нужно выбрать пункт «Install Turbo C on a hard drive» (установить Turbo C на жесткий диск). Далее вам покажут, куда будет установлен компилятор. Если, например, компилятор устанавливается с диска E, то по умолчанию он будет установлен в папку E:\TC. Можно выбрать и другое место для компилятора. Для этого нажмите клавишу Enter, введите новый путь и снова нажмите Enter. Теперь с помощью стрелки ↓ выберите пункт Start Installation и еще раз нажмите Enter. Компилятор установится в нужной папке. Остался последний шаг: найти на диске C: файл AUTOEXEC.BAT и вставить в него путь к компилятору. Если компилятор установлен на диске E в папке TC, то путь к нему будет выглядеть следующим образом:

PATH E:\TC;…

Внеся изменения в AUTOEXEC.BAT, не забудьте перезагрузить компьютер. Теперь все готово к работе.

---

[1] в системе DOS это будет, скорее всего, Norton Commander, в системе Windows 95/98 лучше использовать FAR

Перейдите в папку, где находится исходный текст нашей первой программы (пусть он хранится в файле l11.c), наберите в командной строке файловой оболочки две буквы TC, нажмите Enter и на экране монитора появится окно компилятора. Чтобы загрузить в него исходный текст программы, нажмите клавишу 'F', в появившемся меню выберите пункт LOAD, далее выберите шаблон *.c и, наконец, сам файл L11.c[1]. В окне компилятора появится текст программы (рис. 2.1).

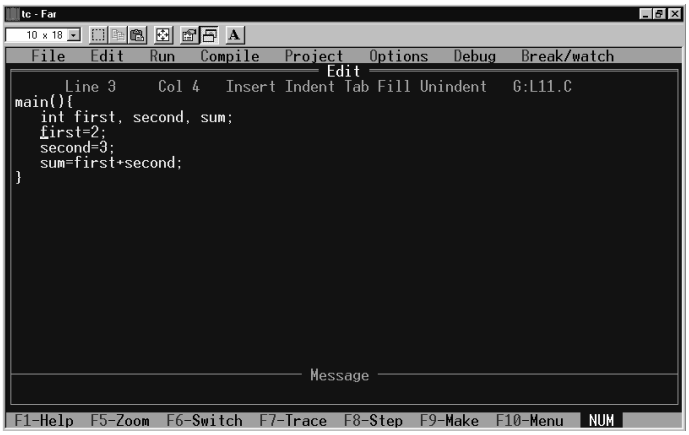

Рис. 2.1. Компилятор Turbo C

Как мы уже знаем, перед запуском программу нужно откомпилировать, то есть превратить текст, приведенный на рис. 2.1, в понятную процессору последовательность команд. Но Turbo C устроен так, что компиляция происходит почти мгновенно, а возможность связать исходный текст программы с порожденной им последовательностью машинных команд почти убеждает нас в том, что Turbo C понимает исходный

[1] Этот файл с исходным текстом программы можно создать в редакторе Блокнот (Notepad). Программы нельзя писать в редакторе MS Word — там другой формат, непонятный компилятору.

текст программы. Действительно, программу можно выполнять по шагам и при этом следить за состоянием любых переменных.

Чтобы, например, следить за переменной `first`, подведем к ней курсор (как на рис. 2.1), нажмем клавишу Ctrl и, не отпуская ее, нажмем F7 (такая комбинация клавиш обозначается как Ctrl-F7). В результате на экране появится окно с именем переменной. При нажатии Enter переменная окажется в нижней части окна Turbo C, под заголовком Watch (рис. 2.2).

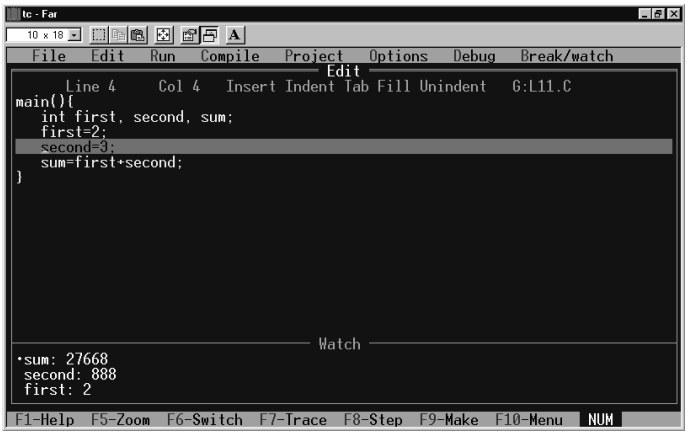

Рис. 2.2. Следим за переменными

Теперь программу можно выполнить по шагам: каждому нажатию клавиши F8 соответствует один шаг. На рис. 2.2 показано состояние переменных после первого шага, на котором переменной `first` было присвоено значение 2.

Как видим, переменные `second` и `sum` равны на этом шаге 888 и 27668, то есть содержат «мусор». На втором шаге переменная `second` станет равной 3, а переменная sum по-прежнему будет равна 27668. И лишь на третьем шаге `sum` (убедитесь в этом сами!) станет равной 5, а переменные `first`

и `second` как были, так и останутся равны 2 и 3 соответственно.

Связь с внешним миром

Карло и виду не подал, что испугался, только ласково спросил:

— *Деревянные глазки, почему вы так странно смотрите на меня?*

Но кукла молчала — должно быть потому, что у нее еще не было рта.

А.Толстой, «Приключения Буратино»

Компиляторы, подобные Turbo C, как заботливые няньки следят за каждым шагом программы (для этого существуют клавиша F8 и окно Watch). Это необходимо, пока программа не вышла из детского возраста и еще нуждается в опеке. Но когда она взрослеет и уже хорошо справляется с поставленной задачей, ей предстоит покинуть среду программирования и начать самостоятельную жизнь. Программа должна работать на любом компьютере — независимо от того, есть на нем компилятор Turbo C или нет.

Хорошо знает об этом и компилятор, поэтому он создает специальный файл с расширением .exe. Загляните в папку, где хранится текст нашей первой программы (файл L11.c), и вы увидите там еще два файла — L11.obj и L11.exe. Файл L11.obj — вспомогательный, так называемый *объектный* файл, создаваемый компилятором. Файл L11.exe и есть та самая скомпилированная программа, которую можно выполнить на любом компьютере, где есть системы DOS или Windows. Чтобы выполнить программу, нужно перейти в содержащую ее папку, набрать на клавиатуре имя L11 и нажать Enter[1]. Перешли в директорий? Набрали L11? Нажали Enter? И что?

---

[1] Так запускается программа в файловой оболочке Norton Commander (для системы DOS) и FAR (для Windows 95/98). Можно воспользоваться и Проводником (Explorer) в системе

А ничего. Молчание. И это понятно — ведь в нашей первой программе нет никакой связи с внешним миром. Пока о программе заботился компилятор, мы могли следить за тем, что творится у нее внутри. Но когда программа попыталась жить самостоятельно, выяснилось, что она еще не умеет говорить — совсем как Буратино, которому папа Карло не успел вырезать рот.

Чтобы заставить программу говорить, необходимо дополнить ее *функцией* `printf()` и строкой `#include <stdio.h>`[1] (см. листинг 2.1).

Листинг 2.1 Программа, которая умеет говорить

```
#include <stdio.h>
main(){
     int first, second, sum;
     first=2;
     second=3;
     sum=first+second;
     printf("Сумма=%d\n",sum);
}
```

Загрузив программу, показанную в листинге 2.1 в Turbo C и скомпилировав ее простым нажатием клавиши F9, получим exe-файл, запустив который увидим на экране компьютера:

```
Сумма=5
```

Значит, добавленная нами строчка

```
printf("Сумма=%d\n",sum);
```

Windows. Для этого перейдите в нужную папку и запустите программу двойным щелчком мыши.

[1] Об этой строке поговорим чуть позже

выводит на экран символы `Сумма=` и само значение переменной `sum` (в нашем случае символ `'5'`).

С помощью `printf()` можно выводить на экран самые разные сообщения, например, вывод слова «Мама» выглядит следующим образом:

```
printf("MAMA");
```

причем курсор остается справа от последней буквы 'А'. Чтобы курсор оказался в начале следующей строчки, добавим в конец фразы символ перевода строки `'\n'`:

```
printf("MAMA\n");
```

Строка `printf("Сумма=%d\n",sum)` в программе, показанной в листинге 2.1, устроена чуть сложнее. Кроме букв и управляющего символа `'\n'` в ней есть еще значок `%d`. Это шаблон, который обозначает место выводимой на экран переменной (переменная `sum` будет показана сразу за символами «Сумма=»), причем это будет целое число (буква `d` задает печать переменной целочисленного типа). Сама переменная указывается правее и отделяется от других знаков, заключенных в кавычки `"…"`, запятой.

С помощью `printf()` можно выводить на экран значения нескольких переменных. Для этого нужно сначала поместить в кавычки символы и шаблоны для этих переменных, затем поставить запятую и сами переменные, также разделенные запятыми. Чтобы, например, вывести на экран значения переменных `first`, `second`, `sum`, соответствующая строка программы должна выглядеть так:

```
printf("first=%d second=%d sum=%d\n",
first, second, sum);
```

В результате мы получим на экране

```
first=2 second=3 sum=5
```

Функция `printf()`, с которой мы только что познакомились, похожа на отдельную программу. На ее вход поступают *аргументы*, разделенные запятыми. В функции

```
printf("MAMA\n");
```

всего один аргумент — последовательность символов, заключенная в кавычки. В функции `printf()`, выводящей три переменные, уже четыре аргумента, один из которых — это строчка в кавычках, содержащая текст и шаблоны, а три других — разделенные запятыми названия переменных. Аргументы огораживаются в функции круглыми скобками `()`. Если посмотреть в начало нашей программы, то станет ясно, что `main()` — это тоже функция, которой можно передавать аргументы, но о том, что это за аргументы и для чего их нужно передавать, мы узнаем чуть позже.

А теперь вернемся к первой строчке листинга 2.1

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

Этой строчки не было в нашей первой программе и легко догадаться, что ее появление связано с функцией `printf()`. Дело в том, что *в языке Си все переменные и функции должны быть описаны*, иначе компилятор не сможет правильно перевести программу в понятную процессору последовательность команд. До начала работы с переменными компилятор должен знать их тип, иначе он не поймет, сколько ячеек памяти выделять под переменную и как кодировать хранящееся в ней двоичное число. То же самое относится и к функциям. Чтобы правильно обращаться с функцией, компилятор должен знать, какие аргументы ей могут передаваться и как воспользоваться результатами ее работы. Поэтому каждая функция должна быть описана еще *до того*, как компилятор встретит ее в тексте программы. Если вы сами создаете функцию (а в языке Си это вполне возможно), то сами же должны ее описать. Но если функция стандартная, ее описание хранится в специальном файле. Так, описание `printf()` хранится в файле stdio.h. Этот файл — неотъемлемая часть компилятора, который, встретив строчку

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

находит файл `stdio.h` и вставляет его в текст программы (именно туда, где стоит #include (включить), а уж потом превращает программу, написанную на языке Си в последовательность команд процессора.

Простые вычисления

Когда в Самоа были открыты начальные школы, туземцы буквально помешались на арифметических вычислениях. Отложив в сторону копья и луки, они расхаживали, вооруженные грифельными досками и карандашами, предлагая друг другу и гостям из Европы простейшие арифметические задачи. Достопочтенный Фредерик Уолпол сообщает, что его пребывание на прекрасном острове было омрачено бесконечными умножениями и делениями.

Р. Бриффолт

Компьютер, как и туземцы с острова Самоа, умеет не только складывать числа, но и вычитать, умножать, делить. Для выполнения этих действий в языке Си используются обычные знаки арифметических операций. Программа из листинга 2.2, выполняет простые арифметические действия, а также получает остаток от деления одного числа на другое.

Листинг 2.2

```
/* простые арифметические действия */
#include <stdio.h>
main(){
int fst = 2, scnd = 3;
int df,prd,qt,rm;
df  = fst - scnd;
prd = fst * scnd;
qt  = fst / scnd;
fst  = 123;
scnd = 17;
rm = fst % scnd;
/* rm - остаток от деления fst на scnd */
printf("%d %d %d %d\n",df,prd,qt,rm);
```

```
}
```

Результат работы этой программы — четыре числа:

```
-1 6 0 4
```

Первое число (-1) получается в результате вычитания 3 из 2, второе (6) — результат умножения 2 на 3, третье (0) — частное от деления 2 на 3[1]. Вообще-то, мы привыкли, что 2/3=0.6666…., но все наши переменные — целочисленные, поэтому при присваивании результата деления переменной `qt` автоматически берется целая часть результата, и поэтому получается ноль. Четвертая цифра (4) есть остаток от деления 123/17. Легко проверить, что 123=17*7+4, то есть остаток действительно равен четырем.

В листинге 2.2 кроме знаков новых арифметических действий нам встретилась еще одна ранее незнакомая конструкция — слова, обрамленные значками `/*…*/`. Это *комментарии*. Компилятор игнорирует все символы, встреченные после наклонной черты и идущей следом звездочкой. Делает он это до тех пор, пока не встретит звездочку и идущую следом косую черту.

Комментарии помогают понять, что делает программа. Начинающие программисты часто ленятся писать комментарии, не понимая, что программа быстро забывается и уже через несколько месяцев никто, даже автор, не сможет понять, что она делает.

Задача 2.1 Поиграйте в туземцев с острова Самоа. Посмотрите, как работает программа из листинга 2.2 с другими значениями переменных `fst` и `scnd`. Что получится, если `fst=1000`, а `scnd=800`? Проверьте результат работы программы с помощью калькулятора. Найдите значение произведения `fst*scnd`, при котором программа еще выдает

---

[1] Значения хранятся в переменных до тех пор, пока мы сами их не поменяем. От того, что переменную `fst` *прочитали* в инструкции `df = fst - scnd`, ее значение *не меняется*. Читать переменную можно сколько угодно раз

правильный результат. Если это удалось, попробуйте догадаться, сколько бит в переменной типа `int`.

В чем преимущество программ

Огромной скоростью компьютеров можно воспользоваться только лишь с помощью программ, в которых большое число действий задается небольшим количеством строк.

Представим себе, что преподаватель информатики решил «занять» класс на некоторое время и предложил написать программу, которая складывает все числа от единицы до 100. Эта программа может выглядеть так:

```
#include <stdio.h>
main(){
int sum;
sum =0;
sum += 1; /* прибавить к sum единицу */
sum += 2; /* прибавить к sum двойку  */
sum += 3; /* прибавить к sum тройку  */
-----------
sum += 100; /*прибавить к sum сотню  */
#printf("Сумма=%d/n", sum);
}
```

Каждая строка `sum += i` — это команда прибавить к переменной `sum` число `i`. Эту команду можно записать и как `sum = sum + i`[1], но `sum += i` короче и понятнее.

---

[1] Запись `sum=sum+i` означает, что к переменной `sum` прибавляется переменная `i` и результат снова отправляется в перемнную `sum`. То есть, ‘=’ означает в Си не равенство, как в алгебре, а *присваивание*

Программу, набросок которой я только что показал, не так-то просто написать, потому что она состоит более чем из 100 строк, ни одна из которых не повторяется. Но 100 строк, в которых подсчитывается сумма, можно представить как последовательность одинаковых групп команд:

```
#include <stdio.h>
main(){
int sum,i;
sum = 0;
i = 0;
sum += i; /* прибавить к sum      i */
i += 1;   /* прибавить к i единицу  */
-------------
/* повторить sum+=1; i+=1; еще 99 раз */
#printf("Сумма=%d/n", sum);
}
```

Такую программу уже проще написать, если использовать возможности практически любого текстового редактора копировать фрагменты текста[1]. Но представим себе, что нужно будет сложить не 100, а тысячу или даже миллион чисел! Ясно, что в программе должна быть конструкция, позволяющая повторить одни и те же вычисления любое число раз. В языке Си этому служит цикл `while()`(пока или до тех пор, пока). Чтобы понять, как он работает, попробуем решить задачу

[1] В Turbo C нужно сначала пометить блок текста: подвести курсор к его началу, нажать Ctrl-K, затем B, подвести курсор к концу блока, нажать Ctrl-K, затем K. После того как блок помечен, подведите курсор к тому месту, куда нужно скопировать блок и нажмите Ctrl-K, затем C.

попроще и напишем фрагмент программы, складывающей все числа от 1 до 2:

```
sum=0;
i=1;
while (i <= 2){
     sum += i;
     i += 1;
}
```

Цикл `while()` начинается с проверки: превышает ли переменная `i` число 2. В нашем случае `i` равна 1, что явно не больше двух. Значит, будут выполнены инструкции, заключенные в фигурные скобки:

```
sum += i; /*sum теперь равна единице*/
i += 1;   /*i теперь равна 2         */
```

То есть, к `sum` прибавится переменная `i`, равная единице. Затем `i` увеличится на единицу и станет равной двум. Теперь мы возвращаемся к началу цикла `while()` и снова проверяем условие `i <= 2`. Поскольку `i` теперь равно 2, условие снова выполняется и программа прибавляет к `sum` уже двойку и снова увеличивает `i` на единицу (теперь `i` и `sum` равны 3), и мы вновь возвращаемся к началу цикла, но теперь проверка `i < =2` показывает, что условие не выполняется (`i` равно теперь 3, а это больше двух!). Значит, не будут выполняться и инструкции в фигурных скобках, а программа (не выполняя больше инструкций в фигурных скобках) перейдет к тому, что следует за циклом `while()`. При выходе из цикла `sum` равнялась трем, то есть сумме всех чисел от единицы до двух — это убеждает нас в правильности ее работы, и теперь мы готовы написать программу, складывающую числа от 1 до 100. Для этого нужно только изменить условие выполнения цикла while() {} (см. листинг 2.3)

Листинг 2.3

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

```
main(){
int sum,i;
sum=0;
i=1;
while (i<=100){
     sum += i; /* прибавить i к sum      */
     i++;      /* увеличить i на единицу */
}
printf("Сумма=%d\n",sum);
}
```

Прежде чем разбирать логику работы этой программы, обратим внимание на строчку `i++;` в цикле `while`. Такая запись часто используется в языке Си и означает увеличение переменной на единицу: `i++` и `i+=1` выполняют одно и то же, но запись `i++` лаконичней и более соответствует духу языка Си.

В цикле `while()` проверяется условие `i<=100`, то есть *не больше ли ста переменная* `i`*?* Это условие выполняется для всех `i` от единицы до ста включительно, то есть цикл будет прокручен ровно сто раз, причем при каждом обороте к переменной `sum` прибавляется `i`. Значит, по окончании цикла `sum` действительно будет содержать сумму всех чисел от 1 до ста. Эта сумма будет выведена на экран инструкцией `printf("Сумма=%d\n",sum)`и, дойдя до закрывающей фигурной скобки, программа завершится.

Задача 2.2 С помощью Turbo C исследуйте программу из листинга 2.3. Выполните программу по шагам, следя за переменными `sum` и `i`, чтобы понять, как действует цикл `while()`. Попробуйте перед `while()` присвоить переменной `i` значение –1. Сколько раз будет выполняться цикл при таком значении `i`?

Задача 2.3. Рассказывают, что в XIX веке один учитель действительно задал ученикам вычислить сумму всех чисел от

единицы до ста. Компьютеров тогда не было, и дети стали прилежно складывать числа. Только один ученик нашел правильный ответ всего за несколько секунд. Это был Карл Фридрих Гаусс — будущий великий математик. Как он это сделал?

Массивы

Компьютер считает гораздо быстрее человека, но, как мы это уже поняли из предыдущего раздела, реализовать свое преимущество в скорости он сможет только в том случае, когда удастся в немногих строчках программы задать большой объем вычислений.

В предыдущем разделе нам помогло то, что складывались 100 чисел подряд. Это позволило в цикле `while()` создавать новое число, прибавляя каждый раз к старому единицу.

Но представим себе, что нужно сложить сто произвольных чисел (например, при вычислении среднего). Если хранить каждое число в отдельной переменной, то задача оказывается неразрешимой, потому что придется перечислить в программе все сто (тысячу, миллион) переменных. А это значит, что прежде чем компьютер за миллионную долю секунды вычислит сумму чисел, нам придется несколько часов (суток, лет) писать бесконечную программу

```
sum=0;
sum += digit1;
sum += digit2;
sum += digit 3;
sum += digit 4;
…
```

Для решения этой проблемы были изобретены *массивы*. Массив — это ряд переменных, у каждой из которых есть специальный номер или *индекс*. Если сто чисел поместить в массив с именем `digits`, то первое число будет равно `digits[0]`, второе — `digits[1]`, третье — `digits[2]`,

сотое — digits[99]. Номер этих чисел в массиве может храниться в целочисленной переменной, и, скажем, третью переменную можно записать как digits[i], если, конечно, i=2 (нумерация переменных в массиве начинается с нуля). Если все сто переменных загнаны в массив, то их сумму вычислит следующий фрагмент программы:

```
i=0;
sum=0;
while(i<=99){
     sum += digits[i];
     i++;
}
```

При i=0 к переменной sum прибавится первое число, то есть элемент массива digits[0], затем индекс i увеличится на единицу и при следующем обороте цикла к sum прибавится число digits[1]. Последним к sum прибавится число digits[99], после чего i станет равным 100 и цикл завершится.

Итак, массивы позволяют перейти к следующему числу за счет увеличения на единицу индекса i. Получается, что произвольные числа, загнанные в массив, так же легко сложить, как идущие подряд целые числа от единицы до ста.

Числа в массиве можно не только складывать. Рассмотрим в качестве примера программу (см. листинг 2.4), которая определяет максимальное число в массиве. Строчка

```
int dig[10]={5,3,2,4,6,7,11,17,0,13};
```

не только задает массив из десяти чисел, но и определяет их начальные значения (dig[0] равно 5, dig[1] y 3 и т.д.), которые могут впоследствии измениться.

Листинг 2.4

```
#include <stdio.h>
main(){
```

```
int dig[10]={5,3,2,4,6,7,11,17,0,13};
int max,i;
max=dig[0];
i=1;
while(i<=9){
     if(dig[i]> max)
         max=dig[i];
     i++;
}
printf("max=%d\n",max);
}
```

Само максимальное значение `max` вычисляется в цикле `while()`. Перед входом в цикл считаем, что `max=dig[0]`. В цикле проверяется, действительно ли это так. Для этого используется *условная инструкция* `if (если)`

```
if(dig[i]> max)
     max=dig[i],
```

которая присваивает переменной `max` значение `dig[i]`, лишь когда `dig[i]` *больше* `max`. Если же `dig[i]` меньше или равно `max`, действие, показанное в инструкции `if()`, не выполняется.

Чтобы лучше понять приведенный пример, проследим за первыми несколькими оборотами цикла, показанного в листинге 2.4. Перед входом в цикл переменные имеют следующие значения: `i=1`, `max=dig[0]`, то есть 5. На первом витке цикла при `i=1` проверяется условие `dig[1] > max`. Раз `dig[1]=3`, а `max=5`, условие *не* выполняется, переменная `i` увеличивается на единицу и становится равной двум, и мы переходим ко второму витку цикла. В нем проверяется условие `dig[2] > max` и, поскольку `max` осталась равной 5, а `dig[2]` стала равна 2, условие вновь не выполняется. Следующий виток цикла пропустим, потому что

при i=3 ничего важного не происходит. Но при i равном 4 выполняется условие dig[i] > max, потому что текущее значение max равно 5, а dig[4] равно 6. Это значит, что выполнится действие, указанное в условной инструкции, max станет равной шести, и цикл уйдет на следующий виток, который снова поменяет значение max , но теперь с 6 на 7. Последний раз условие dig[i] > max выполнится при i=7: dig[7] равно 17, а текущее значение max — 11. Значит после выполнения цикла printf() выведет на экран max=17, и работа программы на этом завершится.

В программах, где заранее известно число оборотов цикла, удобнее использовать другой его вид, задаваемый ключевым словом for (для). Листинг 2.5 показывает, как применить новый цикл for() к отысканию максимального числа в массиве.

Листинг 2.5

```
#include <stdio.h>
main(){
int dig[10]={5,3,2,4,6,7,11,17,0,13};
int max,i;
max=dig[0];
for(i=1;i<=9;i++)
     if(dig[i] > max)
         max=dig[i];
printf("max=%d\n",max);
}
```

Циклом for() управляют три выражения, разделенные точкой с запятой. Первое (в нашем случае i=1) задает начальное значение переменной, второе выражение задает условие, при котором цикл выполняется (у нас это условие i < =9). Условие, как и в цикле while(), проверяется *до* выполнения каких-либо действий. Если оно не выполняется, о

цикл будет пропущен. Наконец, третье выражение задает действие, выполняемое перед обращением к следующему витку цикла. В нашем случае это увеличение переменной `i` на 1 (`i++`) после очередной проверки условия `if(dig[i] > max)`…

Научившись определять наибольшее число в массиве, можно решить следующую, очень распространенную задачу: расставить элементы массива в порядке возрастания (убывания). Эта задача называется *сортировкой* и очень широко применяется на практике (вспомните, что фамилии в телефонном справочнике, статьи в энциклопедическом словаре стоят в алфавитном порядке, то есть *отсортированы*).

Сортировку массива чисел можно провести так, как мы сделали бы это в обычной жизни: найти максимальный элемент и поменять его местами с тем элементом, который стоит в конце массива. Далее найти максимальный из оставшихся элементов и поменять его с тем, что стоит на втором месте от конца. И продолжать так до тех пор, пока не будут отсортированы все элементы. Если, например, массив состоит из десяти чисел и задается таблицей (напомним еще раз, что нумерация элементов массива в языке Си начинается с нуля)

0	1	2	3	4	5	6	7	8	9
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
5	3	2	4	6	7	11	17	0	13

то его максимальный элемент `a[7]` равен 17. Значит, на первом этапе сортировки поменяются местами седьмой и последний, девятый элемент массива:

```
5 3 2 4 6 7 11 13 0 |17
```

После обмена нужно найти максимальное значение в массиве уже из девяти элементов: `a[0]..a[8]`. Шаги сортировки показаны на рис. 2.3. Значения, которые будут меняться на

следующем шаге, обозначены прямоугольниками. Вертикальная черта показывает границу неотсортированного массива, которая сдвигается на один элемент влево после каждого шага.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
5	3	2	4	6	7	11	17	0	13
5	3	2	4	6	7	11	13	0	|17
5	3	2	4	6	7	11	0	|13	17
5	3	2	4	6	7	0	|11	13	17
5	3	2	4	6	0	|7	11	13	17
5	3	2	4	0	|6	7	11	13	17
0	3	2	4	|5	6	7	11	13	17
0	3	2	|4	4	6	7	11	13	17
0	2	|3	5	4	6	7	11	13	17

Рис. 2.3. Простейшая сортировка

Выполняющая сортировку программа показана в листинге 2.6.

Листинг 2.6

```
#include <stdio.h>
main(){
int dig[10]={5,3,2,4,6,7,11,17,0,13};
int i,j,N,mm,tmp;
N=10;
for(i=N-1;i>=1;i--){
     mm=0;
     for(j=1;j<=i;j++)
         if(dig[j] > dig[mm])
             mm=j;   /* где max            */
```

```
        tmp=dig[i];     /* запомнить конец   */
        dig[i]=dig[mm]; /* максимум в хвост */
        dig[mm]=tmp;/* конец - на место max */
    }
    for(i=0;i<N;i++)
        printf("%d ", dig[i]);
    printf("\n");
    }
```

Собственно сортировкой занимается цикл

```
for(i=N-1;i>=1;i--){
…
}
```

Обратите внимание на то, что в цикл входят все инструкции, обрамленные фигурными скобками `{}`. Если скобки убрать, то будет выполнена только первая инструкция `mm=0`, которая засылает в переменную mm, хранящую номер максимального элемента, первоначальное значение 0.

В цикле

```
for(j=1;j<=i;j++)
        if(dig[j] > dig[mm])
            mm=j;       /* где max */
```

проверяется, действительно ли нулевой элемент — максимальный. Если это не так, то положение максимального элемента будет запомнено в переменной `mm`.

Когда максимальный из двух сравниваемых элементов найден, нужно поменять его с элементом массива, номер которого хранится в переменной `i`. Сначала `i` равна `N-1` (то есть, девяти), и максимальный элемент среди `dig[0]`… `dig[9]`меняется с девятым элементом массива. Элементы массива меняют местами три инструкции:

```
tmp=dig[i];      /* запомнить конец */
dig[i]=dig[mm]; /* максимум в хвост*/
dig[mm]=tmp;/* конец - на место max */
```

в которых для временного хранения `dig[i]` используется переменная `tmp` (от английского temporary – временный).

Теперь нужно найти максимум среди оставшихся чисел `dig[0]`…`dig[N-2]`. Это делается на втором витке цикла при `i=N-2`. Найденный максимум помещается уже на второе место с конца, то есть замещает восьмой элемент `dig[8]`. И так продолжается до последнего оборота, где `i=1`. Здесь просто сравниваются нулевой (не забывайте, что нумерация в Си начинается с нуля!) и первый элементы. Если `dig[0]` больше `dig[1]`, они меняются местами. Если нет — массив уже отсортирован.

Наконец, инструкция

```
for(i=0;i<=9;i++)
    printf("%d ",dig[i]);
```

выводит на экран отсортированные числа, разделяя их пробелами, а `printf("\n")` — просто переводит строку.

Задача 2.4. Экспериментальная. Почему в программе сортировки (см. листинг 2.6) перед нахождением минимума запоминается положение нулевого элемента массива `mm=0`? Попробуйте удалить эту инструкции и посмотреть, что будет.

Задача 2.5. Даны две переменные `a` и `b`. Как поменять местами содержимое `a` и `b`, не используя временную переменную `tmp`?

Задача 2.6. Допустим, что программа из листинга 2.6 сортирует сто чисел за 2 секунды. Попробуйте оценить время сортировки двухсот, четырехсот, тысячи чисел.

Как это делается

Среди современных магов не перевелись хвастуны: «Я могу написать программы, которые управляют движением самолетов, перехватывают баллистические ракеты, ведут банковские счета, руководят производственными линиями». На это следует ответ: «И я могу, и каждый это может, но будут ли эти программы работать после того, как Вы их напишете?»

Ф. П. Брукс мл., «Мифический человеко-месяц».

Наверное, многие из вас, глядя на довольно длинную программу сортировки в предыдущем разделе, подумали, что самостоятельно такую ни за что не написать. Кажется почти чудом, что многочисленные переменные, индексы, циклы подошли друг к другу в нужной последовательности, и в результате получилась работающая программа. На самом деле никакого чуда нет, а есть довольно длинный путь от замысла до окончательного текста. Программы рождаются из воздуха и постепенно материализуются.

Сначала самой программы нет — есть только идея найти максимальный элемент массива, затем найти следующий максимальный элемент (без учета уже найденного) и так поступать до тех пор, пока все элементы не встанут по порядку. Но тут же возникает вопрос: куда девать найденные максимумы?

Самое простое решение — выделить дополнительный массив и заполнять его с конца найденными значениями — сначала записать первый найденный максимум в последний элемент дополнительного массива, следующий максимум — в предпоследний, и так поступать до тех пор, пока весь исходный массив не будет отсортирован.

Сделать так, конечно же, можно. Но использовать дополнительный массив, согласитесь, некрасиво. Хочется, чтобы сортировка происходила «на месте» — в том же массиве. Но если найденный максимум записать в конец того же массива, он уничтожит число, которое там находилось! Значит, это число нужно где-то запомнить. И тут нас посещает

идея: число нужно отправить туда, где был найденный максимум, ведь это место теперь свободно!

Теперь пришедшие мысли можно оформить в виде *псевдокода* — последовательности действий, записанной на упрощенном полумашинном языке:

```
max(0,N-1)
dig[mm] exch dig[N-1]
max(0,N-2)
dig[mm] exch dig[N-2]
…
max(0,1)
dig[mm] exch dig[1]
```

Здесь `max(0,N-1)` обозначает вычисление максимума по всему массиву `dig` из `N` чисел `dig[0]...dig[N-1]`, а `dig[mm] exch dig[N-1]` означает обмен элементов mm (максимального элемента) и `N-1` (от английского exchange — обмен).

Глядя на псевдокод, можно понять, что программа на Си должна состоять из двух вложенных циклов:

```
for(i=N-1;i>=1;i--){
      …
      for(j=0;j<=i;j++){
      …
      }
}
```

В первом индекс `i` меняется от `N-1` до `1` и показывает границы массивов, в которых вычисляются максимумы. Одновременно `i` равен номеру элемента, который должен меняться с максимальным. Внутренний цикл предназначен для вычисления максимума в массиве, простирающемся от `0` до `i`.

Теперь ядро программы можно записать более подробно:

```
for(i=N-1;i>=1;i--){
     max=dig[0];
     for(j=0;j<=i;j++){
         if(dig[j] > max){
             max=dig[j];
             mm=j;
         }
     }
     tmp=dig[i];
     dig[i]=max;
     dig[mm]=tmp;
}
```

В цикле

```
for(j=0;j<=i;j++){
     if(dig[j] > max){
         max=dig[j];
         mm=j;
}
```

вычисляется максимум, и в переменной `mm` запоминается номер максимального элемента. Далее инструкции

```
tmp=dig[i];
dig[i]=max;
dig[mm]=tmp;
```

меняют максимальный из сравниваемых элементов и крайний элемент массива, номер которого `i`.

Теперь можно снабдить ядро программы соответствующей оболочкой: объявить переменные (задать их тип), написать `main(){}`, — словом, сделать программу пригодной для компиляции. То, что могло бы получиться, показано в листинге 2.7

Листинг 2.7

```
#include <stdio.h>
main(){
int dig[10]={5,3,2,4,6,7,11,17,0,13};
int i,j,max,N,mm;
N=10;
for(i=N-1;i>=1;i--){
     max=dig[0]
     for(j=0;j<=i;j++){
         if(dig[j] > max){
            max=dig[j]; /* максимум */
            mm=j;    /* и его положение */
         }
     }
     tmp=dig[i]; /* запомнить конец */
     dig[i]=max; /* максимум в хвост
     dig[mm]=tmp;/* конец - на место max */
}
for(i=0;i<N;i++)
    printf("%d ", dig[i]);
}
```

Задача 2.7 Внимательно посмотрите на листинг 2.7 и найдите ошибки в программе.

Пропустив программу, показанную в листинге 2.7, через компилятор, получаем довольно длинный список сообщений об ошибках (Error) и предупреждениях (Warning). Ошибками компилятор считает то, что не позволяет перевести текст программы в последовательность инструкций процессора. Предупреждения не мешают компилировать программу, но могут указывать на какие-то более тонкие ошибки.

В длинном списке ошибок и предупреждений нужно прежде всего обратить внимание на первую ошибку, потому что остальные ошибки могут быть ее следствием. В нашем случае сообщение о первой ошибке выглядит следующим образом:

```
10: Statement missing ; in function main
```

В этом сообщении цифра 10 — это номер строки, в которой обнаружена ошибка, а слова «Statement missing» говорят о том, что в этой строке нет точки с запятой. Но на самом деле точка с запятой пропущена в предыдущей, девятой строке `max=dig[0]` (посмотрите еще раз на листинг 2.7).

Итак, ставим в девятой строке пропущенную точку с запятой и снова компилируем программу. На этот раз появляется всего одно сообщение об ошибке `Undefined symbol 'tmp' in function main`. То есть, компилятор сообщает, что не определена переменная `tmp`. Включаем `tmp` в список переменных, и вновь компилируем программу.

Сообщений об ошибках больше нет, компилятор создает файл с расширением .exe. С замиранием сердца запускаем его и получаем на выходе строчку

```
5 5 5 5 6 7 11 17 17 17,
```

очень слабо напоминающую отсортированный массив.

То, что получился хоть какой-то результат, можно считать удачей. Программа, прошедшая через компилятор, довольно быстро начинает работать правильно, если программист не теряет присутствия духа и умеет искать ошибки. В сущности, программистом можно назвать только того, кто умеет устранять ошибки, ведь написать неработающую программу может каждый.

Получив неверный результат на выходе программы, нужно успокоиться и еще раз взглянуть на ее текст. Еще раз изучив листинг 2.7, замечаем, что в самом начале программы

```
for(i=N-1;i>=1;i--){
     max=dig[0];
     for(j=0;j<=i;j++){
```

не запомнено начальное положение максимума. Нужно, чтобы этот участок программы выглядел так[1]:

```
for(i=N-1;i>=1;i--){
     max=dig[0];
     mm=0;
     for(j=0;j<=i;j++){
…
```

Вносим исправления, компилируем программу еще раз, запускаем, но увы — результат не меняется, он по прежнему неверен. Что же делать? Не сдаваться. Казалось бы, текст программы, прочитан несколько раз, каждая инструкция в нем понятна, но ошибки очень коварны, и часто мы не видим их потому, что «как надо» заслоняют «как есть». Еще один взгляд на исходный текст, и мы, наконец, замечаем строчку

```
dig[i]=max; /* максимум в хвост
```

в которой нет символов */, завершающих комментарий!! Последствия этой, казалось бы, ничтожной ошибки катастрофичны. Ведь компилятор считает комментарием все, что находится между открывающим символом /* и закрывающим */. Значит, инструкция

```
dig[mm]=tmp;/* конец - на место max */
```

будет просто съедена!

---

[1] подумайте, почему

Программисты — народ горячий. И никто не будет думать, к чему приведет пропуск инструкции в программе. Нужно поскорее поставить недостающие символы */, вновь откомпилировать программу, и — ура! — на этот раз результаты правильны. Но это не значит, что верна сама программа. Просто с этими конкретными данными она выдает правильный результат.

Программу, чтобы она работала надежно, нужно испытать на разных данных. Этот этап разработки надежных программ называется *тестированием*. Но прежде необходимо еще раз посмотреть на исходный текст программы и попытаться его улучшить. Программы похожи на прозаические произведения, а хорошие программисты — на писателей, которые многократно возвращаются к своим текстам и постоянно их совершенствуют.

Совершенству нет предела, и даже в небольшой программе сортировки можно изменить многое. Здесь мы остановимся лишь на очевидном: если вычисляется позиция максимума, то сам максимум запоминать не нужно, это лишняя операция. Вместо инструкций

```
max=dig[j]; /* максимум */
mm=j;     /* и его положение */
```

можно просто написать `mm=j`. Сделав соответствующие изменения, получим текст программы, показанный в листинге 2 6.

В заключение замечу, что в программе сортировки нигде не встречается конкретный размер массива, равный десяти. Программа работает с переменной `N` и это делает ее универсальной, то есть пригодной для сортировки массива любого размера (достаточно только изменить N и объявить массив).

Нужно с самого начала стремиться к тому, чтобы в программе не было «магических чисел». Сортировка, явно использующая размер массива:

```
for(i=9;i>=1;i--){
```

…

потребует многократных исправлений при переходе к массиву другого размера. Например, придется поменять цикл, в котором отсортированные числа выводятся на экран:

```
for(i=0;i<10;i++)
    printf("%d ", dig[i]);
```

В языке Си есть много возможностей сделать программу универсальной. С некоторыми из них мы познакомимся в следующих главах.

Глава 3. Странные типы

Странные типы-1

Переменные типа `int`(от английского integer — целое число), которые мы до сих пор использовали в программах, могут хранить только целые значения, что вполне понятно, когда речь идет о количестве детей в семье или числе оборотов цикла `for()`. Но уже отношение двух целых чисел — далеко не всегда целое (например, 2/3), и для хранения таких чисел необходимы другие переменные. В языке Си они называются переменными с плавающей точкой и объявляются как `float` (от английского — плавать). В программе из листинга 3.1 объявлены три переменные: целочисленная `c` и две переменные с плавающей точкой: `a` и `b`. Начальное значение переменной `b` задается константой `2.0` (`2.0` можно еще записать как `2E+0`, где буквой `E` обозначается число десять, а сама запись `2E+0` означает произведение двойки на десять в нулевой степени: 2×10^0, то есть 2, умноженное на 1).

Листинг 3.1.

```
#include <stdio.h>
main(){
float a, b=2.0;
int c=3;
a=b/c;
printf("%f\n",a);
}
```

Результат деления переменной с плавающей точкой `b` на целочисленную переменную `c` присваивается переменной `a`, которая выводится на экран функцией `printf()`. Значок `%f`, обрамленный кавычками, показывает функции `printf()`, что

`a` — это переменная с плавающей точкой. Полученный нами результат деления

0.666667

есть округленное значение бесконечной десятичной дроби

0.6666666666…,

получаемой при делении числа 2 на 3. Округление возникает из-за того, что переменные, хранящиеся в памяти компьютера, не могут вместить бесконечной дроби. Они, как мы узнали из первой главы, занимают всего несколько последовательных ячеек машинной памяти. Одна ячейка, состоящая из восьми бит (1 байт), имеет всего 256 различных состояний, и можно условиться, что в ней будут храниться целые неотрицательные числа от нуля до 255 (всего 256) или целые числа со знаком (от –128 до +127 (их общее число снова равно 256). Но больше 256-ти *разных* чисел один байт вместить не может. Целочисленные переменные `int` занимают несколько байт. Значит, диапазон их значений больше, но все равно ограничен.

Узнать возможности переменных языка Си помогает специальный оператор `sizeof()` (от английского size — размер), который определяет число байт в переменной определенного типа. Листинг 3.2 показывает программу, определяющую размеры новых и уже известных нам типов переменных языка Си.

Листинг 3.2

```
#include <stdio.h>
main(){
char a;
int b;
long c;
float d;
double e;
a=sizeof(char);
```

```
  printf("Размер char=%d\n",a);
  a=sizeof(int);
  printf("Размер int=%d\n",a);
  a=sizeof(long);
  printf("Размер long=%d\n",a);
  a=sizeof(float);
  printf("Размер float=%d\n",a);
  a=sizeof(double);
  printf("Размер double=%d\n",a);
  }
```

Результат ее работы (после компиляции в Turbo C)

```
Размер char=1
Размер int=2
Размер long=4
Размер float=4
Размер double=8
```

поможет понять, что это за переменные и каковы их возможности.. Итак, `char` — это целочисленная переменная размером в 1 байт (или восемь бит). В переменных типа `char` обычно хранятся *символы,* то есть, буквы, знаки препинания, пробел, символ перевода строки, табуляции и т.д. Число различных символов не может превышать 256. Переменная типа `int` занимают два байта[1], и число ее различных состояний равно 256x256=65536. В языке Си переменные типа `int` хранят целые числа со знаком. Обычно уславливаются, что в переменной `int` можно держать числа от -32768 до 32767 (всего 65536 различных чисел).

---

[1] Размер переменных зависит от компьютера и компилятора. В большинстве современных компиляторов переменная int занимает 4 байта

В переменной `long` уже четыре байта, и число ее состояний — 256x256x256x256=4294967296 кажется огромным. Эта переменная также хранит числа со знаком (от –2147483648 до 2147483647). Значения переменных типа `long` задаются константами с буковкой `L` (или l) на конце: `1002034L`.

Какие числа могут храниться в переменных типа `float` и `double` (от английского — двойной) — нам сейчас ни за что не понять, ведь для этого нужно знать, как кодируются числа с плавающей точкой. Но кое о чем можно догадаться, рассматривая принятую в языке Си запись константы типа `float`. Например, запись `1E-36`, означает, что число равно произведению 1 на 10^{-36}. Такая запись наводит на мысль, что в каких-то битах переменных с плавающей точкой хранится степень десятки (в нашем случае это -36), а в каких-то — все остальное (в нашем случае это 1). И действительно, в переменных `float` и `double` хранится пара целых чисел: степень десятки (называемая порядком) и множитель (мантисса). Обычно в переменной `float` хранят числа с порядками от –37 до +37 и шестизначной мантиссой (точность 6 знаков). Порядок переменных `double` почти в десять раз выше (около 308) при точности примерно в двенадцать десятичных знаков.

Числа с плавающей точкой задаются константами, в которых есть десятичная точка `0.00001` или буковка `E` с последующим порядком, например, `1E-5`. Можно сочетать десятичную точку и порядок, например, `1.05E+5` обозначает число $1.05*10^5=1.05*10000=10500$. Константы, задающие значения переменных с плавающей точкой, имеют по умолчанию тип `double`. Значение `float` помечается буковкой `f` в конце константы: `1.00003f`.

Странные типы-2

Казалось бы, зачем придумывать многочисленные типы переменных, когда можно иметь всего два типа: один — целочисленный (пусть это будут переменные длиной 64 байт, чтобы наверняка вместить все возможные числа), и один — с плавающей точкой (пусть он тоже займет как можно больше

ячеек машинной памяти, чтобы обеспечить необходимый диапазон и точность вычислений)?

Действительно, зачем? Затем, что память компьютера не резиновая. И если число можно удержать в двух байтах, не стоит хранить его в четырех или восьми. Создатели языка Си понимали это очень хорошо, ведь компьютер, на котором они работали, имел всего 24 килобайта ($24*2^{10}$=24576 байт) оперативной памяти!

За почти тридцать лет, что существует язык Си, средний объем памяти вырос примерно в тысячу раз, но ее по-прежнему не хватает. Это происходит потому, что программисты стали ее более расточительно расходовать, да и задачи, решаемые компьютерами, стали сложнее и требуют гораздо больших ресурсов.

Но даже если памяти достаточно, действия с длинными переменными занимают намного больше времени, чем с короткими, потому что процессор может обработать за один раз (или, как говорят, *такт*) 4 байта, иногда 8. Сложение 64-х байтовых переменных займет в 8 или 16 раз больше времени, чем сложение переменных длиной в 4 байта.

Вот почему программисты экономят память, стараясь учитывать все особенности переменных. Пусть, например, известно, что целочисленная переменная может принимать только неотрицательные значения (это может быть число оборотов цикла). Тогда в двух байтах, которые она занимает, можно уместить числа от 0 до 65535, но переменную `int` использовать для нее нельзя, так как она хранит положительные числа, не превышающие 32767.

Для экономного хранения неотрицательных чисел в языке Си вводятся беззнаковые переменные, их объявляют как

```
/* a хранит неотрицательные целые числа */
unsigned int a;
```

или просто

```
/* то же самое, что unsigned int a; */
unsigned a;
```

Переменная `long` (от английского — длинный), принимающая только неотрицательные значения, объявляется как

```
unsigned long a;
```

Кроме переменных `int` и `long` для большей гибкости вводят еще целочисленные переменные `short` (от английского — короткий):

```
short a;
```

В Turbo C переменные `short` занимают те же два байта, что `int` (убедитесь в этом сами с помощью `sizeof(short))`, но во многих других компиляторах переменные `int` занимают 4 байта, а переменные `short` — 2. Переменные `short` тоже могут быть беззнаковыми. Для этого их объявляют как

```
insigned short a;
```

Объявляя переменные `short`, `int` и `long` без слова «unsigned», можно быть уверенным, что компилятор будет их рассматривать как целые со знаком. Но переменные типа `char` одни компиляторы понимают как целые беззнаковые, другие — как целые со знаком. Чтобы не запутаться, перед словом `char` лучше ставить «signed», когда нужна переменная, хранящая целые числа от – 128 до +127 или «unsigned», если переменная содержит числа от 0 до 255.

Целочисленные переменные, как и любые другие, задаются константами. Считается, что константа имеет по умолчанию тип `int`. Если нужно задать константу типа `long`, после числа ставится буква ‘L’. Если нужна беззнаковая константа, после числа ставится буква ‘U’(или u), например `12354U`. Для константы типа `unsigned long`, ставятся обе буквы: `10007654UL`.

Анатомия типов

Язык Си — как нож мясника: прост, остр, и исключительно эффективен в умелых руках. Как и всякое острое орудие, Си может принести увечья тем, кто не умеет им пользоваться.

Эндрю Кениг «Си: ловушки и западни»

Программисты очень любопытны. Им хочется знать, как процессор выполняет программу, как устроены переменные. Эти знания лишь на первый взгляд кажутся лишними. Стоит программе повести себя странно, — и без них не обойтись. В этом мы убедимся уже в конце этого раздела.

А пока попробуем понять, каким образом хранятся целые числа. Для простоты предположим, что в переменной всего 4 бита. Таких маленьких переменных на самом деле не бывает, но принцип хранения чисел одинаков и не зависит от числа бит.

В первой главе мы узнали, что четырехбитовая переменная может находиться в 16 различных состояниях, и сейчас попробуем перебрать все эти состояния (каждое их которых представляется в виде двоичного числа), начав с нуля, и получая каждое последующее число прибавлением единицы к предыдущему.

Итак, сначала в переменной записан ноль:

0 0 0 0 /* число ноль */

Прибавив к нему единицу, получим единицу в двоичном коде:

0 0 0 1

Чтобы прибавить следующую единицу, нужно знать, как складываются двоичные числа. Как и десятичные, двоичные числа складываются поразрядно, начиная с младшего разряда. Причем правила сложения в пределах одного разряда, крайне просты:

0+0=0

0+1=1

1+0=1

1+1=0 (1 в уме).

Значит,

0 0 0 1 + 0 0 0 1 = 0 0 1 0 (складываем единицы в младших разрядах, ноль пишем, один в уме. Затем складываем нули в

следующих разрядах, получаем ноль плюс запомненная единица. Итог: единица во втором разряде). Результат последовательного прибавления единиц показан на рис. 3.1.

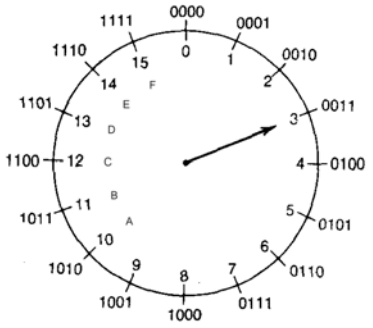

Рис. 3.1. Так хранятся целые без знака

Двоичным кодам, показанным на рисунке, сопоставлены целые неотрицательные числа от 0 до 15. То есть, на рис. 3.1 показана беззнаковая переменная `unsigned`. Числа идут по кругу не случайно. Если прибавить к 15 (в двоичном коде это 1111) единицу, получится совсем не 16, а ноль! Действительно, двоичное число 16 (10000) нельзя уместить в 4-х битах. Когда к числу 1111 прибавляется 1, все биты «валятся», оставляя за собой одни нули. То же самое будет происходить с любыми целочисленными переменными, только числа (в зависимости от размера переменной) будут другими. Если переменная типа `unsigned char`, то ее предельное значение равно 255. Для переменной `unsigned char` справедливо равенство: 255+1=0. А дальше все идет по кругу: 0,1,2,3,…255,0,1…

С помощью чисел, показанных на рисунке 3.1, можно записать любые последовательности бит. Например, 16-битовое число, в котором все биты равны единице, можно записать так:

15 15 15 15

Здесь каждое число 15 обозначает 4 идущих подряд единичных бита: 1111. При такой записи приходится ставить пробелы после каждой цифры, потому что некоторые

комбинации бит требуют одной цифры (например, число 3 обозначает биты 0011), а некоторые — двух (число 10, например, обозначает биты 1010). Для удобства каждая цифра, большая девяти, заменяется буквой. Например, латинская `B` соответствует битам 1011 или (в десятичном представлении) числу 11. При таких обозначениях число, в котором все 16 бит равны 1, можно записать как `FFFF` — пробелы теперь не нужны, потому что каждой комбинации из 4 бит соответствует один символ (см. рис. 3.1).

Такая запись чисел называется *шестнадцатеричной*, она очень компактна и часто используется, когда программиста интересует не значение числа, а расположение его бит. В языке Си шестнадцатеричные числа записываются специальными константами, которые начинаются символами `0x`: Программа, показанная в листинге 3.3, напечатает число 255 — именно такое десятичное число соответствует восьми идущим подряд единичным битам.

Листинг 3.3

```
#include <stdio.h>
main(){
int i;
i=0xFF;
printf("%d\n",i);
}
```

Поняв устройство переменных без знака, посмотрим, как кодируются переменные со знаком. Соответствующая круговая диаграмма показана на рис. 3.2.

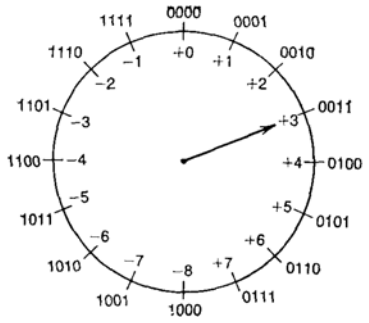

Рис. 3.2. Числа со знаком

Четыре бита, как ни крути, могут находиться лишь в 16 разных состояниях. Но теперь этим состояниям соответствуют другие числа. Любопытно, что за семеркой сразу следует –8(1000). Далее идут –7(1001), -6(1010), -5(1011), и все кончается нулем. Такой способ кодирования поначалу может показаться странным, но на самом деле он очень удобен, потому что позволяет заменить вычитание числа на прибавление такого же числа, но с противоположным знаком. Например, 5 – 4 = 5+ (–4)= 0101 + 1100=0001=1. На самом деле, при сложении чисел 5 (0101) и –4 (1100) получается 17, то есть в двоичной записи 10001. Но в нашей переменной всего 4 бита и старшая единица *вытесняется за ее пределы*, остается только единица в младшем разряде. Вот и получается, что 5 + (-4)=1.

Легко заметить, что отрицательное число получается, если из 16 вычесть такое же по абсолютной величине, но положительное число. Чтобы, например, получить код для числа –7, нужно из шестнадцати вычесть семь:

16-7=9=1001

Можно еще сказать, что отрицательное число получается *дополнением до шестнадцати соответствующего*

положительного. Например, -7 — это дополнение числа 7 до 16, то есть 9 (в двоичном представлении — 1001)

Теперь понятно, что сумма положительного числа и равного ему по абсолютной величине отрицательного (например 6+(-6)) всегда равна 16. Это число в двоичной записи равно 10000 и занимает 5 бит. Но в нашей переменной четыре бита, и старший разряд «сваливается» с ее левого конца, оставляя четыре нуля, что и требуется.

Таков принцип кодирования отрицательных чисел. Осталось сказать, что отрицательные числа будут кодироваться по-разному в зависимости от размера переменной. Если переменная 8-битовая, будет дополнение до 2^8=256, если в переменной 16 бит — дополнение до 2^{16}=65536 и так далее.

Теперь нам должно быть понятно, почему не работает программа из листинга 3.4.

Листинг 3.4.

```
#include <stdio.h>
main(){
unsigned char i;
for(i=0;i<=255;i++)
    printf("%d\n",i);
}
```

По замыслу эта программа должна вывести на экран все числа от нуля до 255 и затем прекратить работу. На самом же деле она будет работать вечно, выводя на экран все числа от 0 до 255 вновь и вновь.

Весь фокус в том, что переменная `i` объявлена как `unsigned char`, а мы знаем, что диапазон ее значений — от нуля до 255. Казалось бы, условие `i<=255` показывает, что `i` меняется в допустимых пределах. Но давайте посмотрим, что будет с циклом, когда `i` достигнет 255. Условие `i <= 255` в этом случае выполнится, `printf()` покажет число 255, а дальше `i` увеличится на единицу и снова будет проверено условие

`i < =255`. Чему будет равно `i` в этот момент, мы уже знаем: 255+1 означает для переменной `unsigned char` ноль! Условие `i <= 255` будет выполнено, и цикл будет прокручен снова 256 раз, затем `i` снова станет равна нулю и так до бесконечности.

Очень важно понимать, что компилятор *не может и не хочет обнаруживать такие ошибки*. Язык Си не вмешивается в замысел программиста, считая, что тот хорошо знает, что делает. Си создавался для программистов, которым не нужно мешать надоедливыми проверками и предупреждениями.

Подбирайте выражения

Настоящие мастера могут создавать очень сложных, многопрограммных, самообучающихся дублей. Такого вот супера Роман отправил летом вместо меня на машине. И никто из моих ребят не догадался, что это был не я. Дубль великолепно вел мой «Москвич», ругался, когда его кусали комары, и с удовольствием пел хором. Вернувшись в Ленинград, он развез всех по домам, самостоятельно сдал прокатный автомобиль, расплатился и тут же исчез прямо на глазах ошеломленного директора проката.

А. и Б. Стругацкие «Понедельник начинается в субботу»

Комбинируя переменные разных типов, можно экономить память и ускорять выполнение программ. Но при этом возникает новая трудность: нужно хорошо понимать, как переменные взаимодействуют друг с другом.

Пусть, например, необходимо поделить одну целочисленную переменную на другую:

```
int a=2, b=3;

float c;

c=a/b;
```

Поскольку результат объявлен как `float`, мы ожидаем, что переменная `c` после деления будет равна 0.666667 , то есть

округленному значению бесконечной дроби 2/3=0.66666… На самом деле `c` будет равна нулю, потому что две целочисленные переменные делятся по законам целочисленной арифметики, и в результате получается ноль, который преобразуется затем в число с плавающей точкой, оставаясь по-прежнему нулем.

Чтобы в переменной `c` оказалось дробное число, нужно заставить компилятор делить 2 на 3 по закону переменных с плавающей точкой, для чего необходимо явно указать значение одной из них:

```
int b=3;

float c;

c=2.0/b;
```

Когда значение делимого задается константой 2.0, а не просто цифрой 2, компилятор понимает, что это число с плавающей точкой и теперь перед ним совершенно другая задача: поделить число с плавающей точкой на целое. Для этого в языке Си существует правило: если встречаются две переменные разных типов, то переменная более «узкого» типа преобразуется к более «широкому» типу, и дальше выполняется нужное действие. В нашем примере более «узок» целый тип, ведь если присвоить целочисленной переменной значение с плавающей точкой, возможны потери. Поэтому компилятор создает временную переменную (дубль переменной `b`), которую преобразует в число с плавающей точкой, затем число 2.0 делится на эту переменную. В результате после выполнения такой программы `c` будет равна 0.6666667. При этом сама `b` не пострадает (не изменит свой тип) и останется такой, какой была.

Чтобы получить нужный результат, в языке Си есть возможность силой привести переменную к нужному типу. Во фрагменте программы

```
int a=2, b=3;

float c;

c=a/(float)b;
```

Переменная b приводится к типу `float`, а за ней уже автоматически к типу float приводится и переменная `a`[1]. Результат деления будет равен 0.666667. Замечу, что запись

```
c=(float) a/b;
```

тоже достигнет цели, потому что слово (**`float`**) относится не к результату целочисленного деления, а к переменной **`a`**. На первых порах трудно понять, что к чему относится, поэтому лучше поставить «лишние» скобки:

```
c=((float)a)/b;
```

Общее правило языка Си (приведение более «узкого» типа к более «широкому») справедливо и для целых переменных, но здесь больше неожиданностей — из-за того, например, что результат может не уместиться в переменной того же типа (например, сумма двух целых со знаком 32000 + 32000) или из-за того, что переменную одного типа нельзя без потерь привести к переменной другого типа.

Особенно внимательным нужно быть с переменными типа `char` и `short`, которые могут по-разному приводиться к типу `int`. Программа, показанная в листинге 3.5, присваивает значение `0xFF` переменным трех типов: `unsigned char` , `signed char` и `int`.

Перед выполнением программы все восемь бит как в `a`, так и в `b`, будут установлены в 1. Для переменной `signed` это соответствует числу –1, а для переменной `unsigned` — числу 255 (см. круговую диаграмму на рис. 3.1).

Листинг 3.5

```
#include <stdio.h>
main(){
unsigned char ch1=0xFF;
```

---

[1] Ни `a` ни `b` при этом не пострадают. В делении примут участие их дубли, приведенные к типу `float`.

```
signed   char ch2=0xFF;
int i;
i=ch1;/*unsigned char превращается в int*/
printf("%d\n",i); /* выведет 255     */
i=ch2; /* signed char превращается в int*/
printf("%d\n",i); /* выведет c1      */
printf("%u\n",i); /* выведет 65535   */
}
```

Первая функция `printf()` выведет на экран число 255, и в этом нет ничего удивительного, потому что константа `0xFF` — это восемь идущих подряд единичных бит, что и равно 255.

Вторая функция `printf()` напечатает уже число –1, и это тоже понятно. Ведь переменная `ch2`, значение которой присваивается переменной `i`, относится к типу `signed char`, а мы уже знаем (см. рис. 3.2), что в переменной со знаком все биты, установленные в 1, соответствуют числу –1. Но `printf()` в данном случае печатает переменную `i`, в которой 16 бит, а не восемь! Значит, когда переменной `int` присваивается значение переменной `char`, лишние биты заполняются единицами, если переменная `char` объявлена как `signed`, или нулями — если `char` — беззнаковая переменная!

После присваивания начальных значений наши переменные в двоичном представлении выглядят так:

```
11111111            /* ch1=255 */
11111111            /* ch2=-1  */
0000000011111111    /* i=ch1   */
1111111111111111    /* i=ch2   */
```

Последняя функция `printf("%u\n",i)` использует шаблон `%u` для печати беззнаковых целых. Число 65535, которое она печатает, соответствует шестнадцати единичным битам, что и требовалось доказать.

Строки и символы

Среди объектов, с которыми обычно работают программы, особое место занимают тексты, то есть последовательности букв, пробелов, знаков препинания и невидимых управляющих символов, которые обозначают конец строки, начало нового абзаца и т.п.

Одна из самых распространенных программ, о которой мы уже довольно много говорили, — это компилятор. Она анализирует тексты программ и превращает их в команды процессора. Другая, не менее распространенная, программа — текстовый редактор. С ее помощью пишутся программы, повести, романы. В работе программиста и писателя много общего.

Буквы, знаки препинания и управляющие символы называются в языке Си просто *символами*. Один символ кодируется восемью битами и может храниться в переменной `char`. Собственно, символы для компьютера — такие же числа. Но чтобы подчеркнуть их особую роль, символы часто записывают в виде букв, обрамленных одинарными кавычками:

```
unsigned char x;

x='a';
```

После выполнения этого фрагмента программы в переменной `x` окажется число, которым кодируется малая латинская буква `a`. Как правило, это число равно 97, но могут быть компьютеры, где эта буква кодируется иначе. Поэтому запись `x='a';` справедлива для разных компьютеров, она, как говорят, более *мобильна*.

Некоторые символы невидимы, и для их записи используется последовательность букв в кавычках, начинающаяся с наклонной черты. Самые распространенные — уже известный нам символ перевода строки `'\n'` и `'\r'` — возврат каретки.

Символы обычно следуют друг за другом, образуя слова, предложения, абзацы, главы, тома. Их удобно хранить в массивах типа `char`, поскольку любой символ, независимо от

того, записывается ли он с помощью буквы, или последовательности букв, перед которой ставится наклонная черта, может быть закодирован восемью битами. Слово «МАМА» можно хранить в массиве из четырех чисел:

```
char x[4];
x[0]= 'M';
x[1]= 'A';
x[2]= 'M';
x[3]= 'A';
```

Вывести его на экран можно с помощью функции `printf()`, в которой есть специальный шаблон для печати символов `%c`:

```
int i;
for(i=0;i<=3;i++)
    printf("%c",x[i]);
printf("\n");
```

Такой способ вывода, хоть и законен, но очень неудобен, потому что требует заранее знать число символов в массиве. Можно упростить вывод на экран, завершив слово или фразу специальным символом, который не может быть ни буквой, ни знаком препинания — ничем таким, что могло бы встретиться в *середине* слова (фразы). В языке Си этот символ называется `null`, записывается как `'\0'` и представляет собой настоящий ноль, то есть число, в котором все 8 бит равны нулю. Программа из листинга 3.6, подобно маленькому ребенку, произносит свое первое слово «МАМА».

Листинг 3.6

```
#include <stdio.h>
main(){
unsigned char x[5];
int i;
x[0]='M';
```

```
x[1]='A';
x[2]='M';
x[3]='A';
x[4]='\0';
i=0;
while(x[i] != '\0'){
     printf("%c",x[i]);
     i++;
}
printf("\n");
}
```

Здесь в массиве `x[]` уже пять символов (в конце слова добавлен `'\0'`), и вместо цикла `for()`, более уместного, когда число оборотов точно известно, применен цикл `while()`. В записи условия нам встретился новый значок `!=`, означающий «не равно», то есть, цикл будет выполняться до тех пор, пока `x[i]` отличается от `'\0'`.

Программа, показанная в листинге 3.6, хоть и стала лучше с появлением символа `'\0'`, все еще очень неудобна, потому что приходится вручную вводить в массив каждую букву. Чтобы поручить этот процесс компилятору, в языке Си есть специальная конструкция — *строка*[1]. Строка — это последовательность символов, обрамленных двойными кавычками:

```
"MAMA"
```

Строка состоит из символов, заключенных в двойные кавычки, и завершающего символа `'\0'`, который добавляет

[1] В некоторых книгах по языку Си строки называют *стрингами* (от английского слова string, т.е. «строка»), а символы — *литерами*.

компилятор. Чтобы загнать строку в массив, достаточно написать:

```
unsigned char[]="MAMA";
```

Встретив такую строчку, компилятор сам создаст массив из пяти символов — четырех букв и завершающего символа null '\0' (см. рис. 3.3.). Точно такой же массив мы уже создавали “вручную” (см. листинг 3.6).

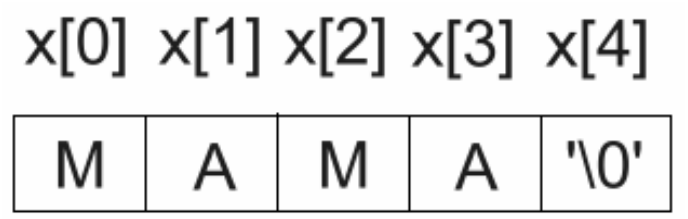

Рис. 3.3. Массив x, создаваемый по команде char x[] = "MAMA"

Программа, выводящая на экран слово «MAMA», будет теперь выглядеть так:

```
#include <stdio.h>
main(){
int i;
unsigned char x[]="MAMA";
i=0;
while(x[i]){
    printf("%c",x[i]);
    i++;
}
printf("\n");
}
```

Обратите внимание, теперь условие выполнения цикла записано как `while(x[i])`, а раньше (см. листинг 3.6) оно выглядело так: `while(x[i] != '\0')`. Дело в том, что результат проверки условия в языке Си — это единица, если условие выполняется и ноль — если нет. Но цикл `while()` устроен так, что он будет выполняться, если в скобках окажется любое ненулевое число. Массив `x[]`, как мы знаем, содержит символы, причем нулю равен только последний — `'\0'`. Поэтому `x[i]` будет отличен от нуля для всех букв нашего слова. Цикл `while(x[i]){...}` прекратит работу, когда `x[i]` равен нулю, то есть условия `while(x[i] != '\0')` и `while(x[i])` хоть и не идентичны (проверка первого даст четыре единицы и завершающий ноль, проверка второго — четыре положительных числа и тот же ноль), но приводят к одним и тем же результатам. Запись `while(x[i])` более короткая и больше соответствует духу языка Си.

Наконец, сделаем последний шаг в совершенствовании программы, говорящей свое первое слово «МАМА» — выбросим цикл `while()`, над улучшением которого так долго работали. Оказывается, в функции `printf()` есть специальный шаблон `%s` для печати строк. С его помощью программа станет совсем короткой:

```
#include <stdio.h>
main(){
unsigned char x[]="MAMA";
printf("%s\n",x);
}
```

Вывод на экран строки, помещенной в массив `x[]`, внешне выглядит так же, как и вывод обычного числа x: функции передается только имя массива. Но мы уже догадываемся, что за такой простотой стоят довольно сложные действия: нужно определить место в памяти, где начинается строка, затем проверять каждый последующий символ: не равен ли он нулю, и если не равен — показывать на экране. Если символ нулевой, значит, пришел конец строке, нужно вывести символ `'\n'` (то

есть переместиться в начало следующей строчки экрана) и завершить программу. Другими словами, `printf()` все равно делает примерно то же, что и цикл `while()`, с которым (перед тем как бросить) мы так долго возились в этом разделе.

УКАЗАТЕЛИ

— Вот он! — закричал Вий и уставил на него железный палец. И все, сколько ни было, кинулись на философа. Бездыханный, грянулся он на землю, и тут же вылетел дух из него от страха.

Н.В. Гоголь. «Вий»

Предыдущий пункт завершился выводом на экран строки `"MAMA"`. Оказалось, что для этого нужно передать функции `printf()` имя массива, хранящего строку. Чтобы функция `printf()` могла что-то сделать, она должна знать, где, в какой ячейке памяти начинается строка и, поскольку функции передается только имя массива, в нем и содержится адрес строки.

Этот адрес можно передать и другим, более прямым способом, через *указатель*. Указатель на строку, как и любая другая переменная, должен быть объявлен:

```
unsigned char *px="MAMA"
```

и вся программа, выводящая на экран слово «MAMA», будет выглядеть при этом так:

```
#include <stdio.h>
main(){
unsigned char *px="MAMA";
printf("%s\n",px);
}
```

Здесь `px` — это *указатель на строку* `"MAMA"`. Чтобы показать, что объявляется именно указатель, перед его именем ставится звездочка:

```
unsigned char *px="MAMA";
```

Когда компилятор встречает такую строку, он прежде всего выделяет пять байт, записывает туда 5 символов — четыре буквы и завершающий null '\0', затем помещает *адрес* первого байта строки в указатель px.

Указатели бывают не только на строки, но и на обычные переменные. Например, объявление

```
int *pa;
```

задает указатель на переменную int. Как и обычные переменные, указатель после объявления содержит «мусор», то есть указывает пальцем в небо. Чтобы его «настроить» на определенную переменную, нужно получить ее адрес. Для этого в языке Си есть специальный оператор &. В программе из листинга 3.7 сначала объявляются две переменные, i (типа int) и px (указатель на int):

Листинг 3.7

```
#include <stdio.h>
main(){
int i;
int *px; /* px с указатель на int  */
i=2;
px=&i; /*в px с адрес i            */
*px=3; /* i теперь равно 3!        */
printf("%d\n",i);
}
```

Затем переменной i присваивается значение 2. Далее идет строчка

```
px=&i;
```

которая засылает в указатель px адрес переменной i. Следующая строка

```
*px=3;
```

записывает число 3 по адресу, хранящемуся в `px`. Но в `px` хранится адрес переменной `i`, значит `i` теперь равна 3! Действительно, скомпилировав и запустив программу, мы увидим, что она выводит на экран число 3.

Звездочка`*`, стоящая перед указателем `px` называется *оператором раскрытия ссылки*. Получается, что если `px` — указатель на `int`, то `*px` — *переменная* типа `int`. Теперь понятно, почему так странно объявляется указатель. Строка

```
int  *px;
```

как бы говорит нам, что `*px` — это что-то типа `int`. В программе (см. листинг 3.7) проложена тайная тропа к переменной `i`. Вместо того чтобы честно написать `i=3` мы незаметно подкрались, выведав сначала адрес переменной `px=&i`, а затем написав по этому адресу число 3.

Похоже, от указателей исходит большая опасность, ведь они позволяют скрытно менять значение переменной. Может случиться и так, что указатель содержит неверный адрес и запись чего-то по этому адресу, испортит данные или, что еще хуже, саму программу. Все это так. Но если пользоваться указателями осторожно, они принесут больше пользы, чем вреда. Но об этом — в следующих главах.

Глава 4. Действия

Очередность

Нам уже знакомы арифметические вычисления, выполняемые с помощью операторов + (сложение), – (вычитание), * (умножение), / (деление), % (вычисление остатка при целочисленном делении). Но до сих пор все эти операторы встречались нам по отдельности. А что произойдет, когда они окажутся вместе? Каков, например, будет результат вычислений `x=5+2*2`? Если сначала выполнить сложение, то `x` будет равен (5+2)*2=14. Если же первым делом выполнить умножение, то `x` окажется равным 5+(2*2)=9.

Проверка показывает, что x будет равен 9, то есть в языке Си оператор умножения обладает *большим приоритетом*, чем оператор сложения. Правила языка таковы, что операторы * / % выполняются в первую очередь, их приоритет одинаков[1]. Операторы + и – выполняются во вторую очередь, их приоритет по отношению друг к другу тоже одинаков. Арифметические операторы с равным приоритетом выполняются слева направо, то есть при вычислении выражения `a + b + c` сначала к `a` прибавится `b`, а потом к сумме `a` и `b` прибавится `c`.

Если естественный приоритет нас не устраивает и по смыслу задачи сначала нужно сложить две переменные, а лишь потом умножить их сумму на другую переменную или число, применяются круглые скобки:

```
x=(a + b) * c;
```

Скобки лучше ставить даже там, где они не обязательны — это сделает вычисления более понятными. Нужно стараться избегать сложных формул. Лучше разбить формулу на

---

[1] В приложении А есть таблица приоритетов различных операторов языка Си.

несколько частей, чтобы можно было проконтролировать каждую часть вычислений и быстрее найти возможную ошибку.

Арифметические операторы, о которых мы только что говорили, называются *бинарными*, потому что связывают каким-либо действием *две* переменные. Но есть, как мы уже знаем, операторы, которые применяются к одной переменной. Это *унарные* операторы. Простейший унарный оператор — это минус, меняющий знак переменной:

```
x=ca;
```

Другой уже известный нам унарный оператор — два идущих подряд плюса `++` — увеличивает значение переменной на единицу. Этот оператор может стоять как *до* переменной (`++i`), так и *после* нее (`i++`). Если переменная, к которой применяется оператор `++`, стоит отдельно, как, например, в цикле `while()`:

```
while(условие) {
    …
    ++i;
}
```

то положение оператора `++` значения не имеет. Куда ни поставь его, он увеличит переменную на единицу. Вместо него можно просто написать `i=i+1` или `i+=1`. Но если переменная, к которой применяется оператор `++` (его еще называют оператором автоувеличения), не одинока, а, например, складывается с другой переменной или передается функции, положение оператора становится важным.

В программе, показанной в листинге 4.1, переменной `x` присваиваются значения переменных `i` и `j`, первоначально равных единице. К обеим переменным применяется оператор автоувеличения `++`, но в строчке `x=++i` он стоит слева от переменной и поэтому `i` *сначала* увеличивается на единицу, а уж потом ее значение (двойка) присваивается `x`. В строчке `x=j++` переменной `x` присваивается старое значение `j`, то

есть единица, а уж *потом* `j` увеличивается на единицу. В конце концов, как `i`, так и `j` становятся равными двойке.

Листинг 4.1

```
#include <stdio.h>
main(){
int i,j,x;
i=1;
j=1;
x=++i;  /*  x=2 */
printf("%d\n",x);
x=j++;  /*  x=1 */
printf("%d\n",x);
printf("%d %d\n",i,j); /*i=2 j=2 */
}
```

Те, кто любит и понимает Си, часто используют оператор `++`, потому что он отвечает духу языка, позволяя написать программу просто и кратко. Цикл `while()`, в котором мы уже выводили на экран слово «МАМА»

```
while(x[i] != '\0'){
     printf("%c",x[i]);
     i++;
}
```

можно записать так:

```
while(x[i])
      printf("%c",x[i++]);
```

что гораздо изящнее и проще. Положение оператора `++` правее переменной `i` (такое расположение называют *постфиксной*

записью), говорит о том, что `i` увеличивается *после того* как очередной символ `x[i]` выведен на экран.

Кроме оператора `++`, существует и оператор *автоуменьшения* `cc`, который делает все так же, но с точностью до наоборот: если оператор `++` увеличивает значение переменной на единицу, то `cc` уменьшает его.

Многие начинающие программисты считают особым шиком использовать операторы `++` и `cc`, не понимая опасностей, которые с ними связаны. Дело в том, что в языке Си не определено, какое из слагаемых вычисляется первым. Одни компиляторы могут при вычислении суммы `x=a + b` сначала вычислить `a`, затем `b`. Другие же, наоборот, вычислят сначала `b`, потом `a`. Поэтому сумма

```
i=1;
x=++i + i;
```

может оказаться равной четырем, трем и вообще чему угодно. Точно так же нельзя сказать, что выведет на экран `printf()` в следующем фрагменте:

```
int i = 7;
printf("%d\n", i++ * i++);
```

— результатом может быть 49, 56, 64 или что-то еще. Положение оператора `++` справа от переменной говорит о том, что увеличение на единицу произойдет *после*, но после чего? Компилятор может, например, умножить 7 на 7, а потом дважды увеличить i на единицу. *Значение выражений, где переменная, измененная операторами ++ или – – встречается дважды и более раз, непредсказуемо.*

Условности

Условная инструкция `if(условие)`, относится к числу тех *условностей*, пренебрегать которыми недопустимо. Малейшая ошибка — и программа отправится по ложному пути. Чтобы этого не произошло, нужно правильно записать условие в

круглых скобках, стоящих после `if`, где, наряду с арифметическими, можно встретить и *логические операции*.

От арифметической логическая операция отличается прежде всего тем, что ее результат — целое число, у которого только два значения: 0 (ложь) и 1 (истина). Вместо присваивания `i=1` можно записать `i=10 > 1`, потому что значение логической операции 10 > 1 истинно и, следовательно, равно единице. То, что результат логической операции представляет собой целое число, означает, что арифметические и логические операторы можно смешивать. Выглядит это довольно странно, но работает. Можно, например, присвоить переменной `i` значение 2 следующим образом:

```
i = (10>1) + (100>10);
```

Оба выражения в скобках истинны, то есть каждое равно единице, а 1+1, как известно, равно двум. Скобки в такой записи обязательны, иначе первым будет выполнено сложение, так как приоритет операции сложения выше, чем приоритет логической операции. Если скобок не поставить, то после того, как к 1 прибавится 100, выражение справа превратится в загадочную последовательность цифр и стрелок:

```
i = 10 > 101 > 10;
```

Чтобы ее расшифровать, нужно знать, что логические операции выполняются слева направо[1]. Значит, результат операции `10 > 101` (то есть 0) будет сравниваться с десятью:

```
i = 0 > 10;
```

В итоге `i` окажется равной нулю.

Кроме логического оператора > (больше) есть еще несколько операторов с равным приоритетом, выполняемых слева направо: >= (больше или равно), < (меньше), <= (меньше или равно).

---

[1] В Приложении А есть таблица приоритетов и порядок выполнения различных операторов языка Си.

Смысл следующего бинарного оператора == (равно) понятен. Результат операции `x==y`, где `x` и `y` — целочисленные переменные, равен единице, если x равен y, и нулю — в противном случае. Прямо противоположен ему оператор != (не равно). Операция `x != y` даст единицу, если `x` и `y` *не равны*, и ноль — если равны. Операторы `==` и `!=` имеют меньший приоритет, чем операторы `>`, `>=`, `<`, `<=`, поэтому в присваивании

```
x=100 > 10 == 10 > 1;
```

компилятор сначала займется выражениями 100 > 10 и 10 > 1 (оба выражения истинны, так что получатся две единицы), а затем сравнит две единицы и запишет в переменную `x` результат этого сравнения (тоже единицу).

У следующих двух логических операторов `&&` и `||` еще более низкий приоритет, потому что они имеют дело уже с результатами сравнений. Исследовать эти самые логические из всех разобранных логических операторов поможет программа, показанная в листинге 4.2.

Листинг 4.2.

```
#include <stdio.h>
main(){
int i,j;
for(i=0;i<2;i++)
     for(j=0;j<2;j++){
         printf("%d && %d = %d ",i,j,i&&j);
         printf("%d || %d = %d\n",i,j,i||j);
     }
}
```

В ней перебираются все возможные комбинации нуля и единицы (всего их четыре), выводятся на экран значения

каждой из двух переменных и соответствующее значение оператора.

Функция `printf("%d || %d = %d\n",i,j,i || j);` выводит на экран переменные `i`, `j` и результат операции `i || j`. То есть, аргументами, передаваемыми в функцию, могут быть не только переменные, но и выражения (переменные и операторы).

Результат работы программы (см. листинг 4.2)

```
0 && 0 = 0    0 || 0 = 0
0 && 1 = 0    0 || 1 = 1
1 && 0 = 0    1 || 0 = 1
1 && 1 = 1    1 || 1 = 1
```

показывает, что оператор `&&` (логическое И) равен единице лишь в том случае, когда оба значения (их называют операндами) истинны (равны 1), а в остальных случаях равен нулю; оператор `||` (логическое ИЛИ) равен нулю лишь в том случае, когда оба операнда нулевые, а в остальных трех случаях он равен единице.

Задача 4.1 Напишите простую программу, вроде той, что показана в листинге 4.2, где исследуются операторы `<`, `<=`, `>=`, `==`, `!=`.

Стоит еще еще раз напомнить, что *истинным в языке Си считается любое ненулевое значение переменной.* Поэтому значение `10 && c5` также будет равно 1, а `0 && 100000` — нулю.

И, наконец, последний логический оператор `!` преобразует любое значение в свою логическую противоположность. Это унарный оператор, переводящий истину в ложь и наоборот. Значений у него всего два — `0` и `1`. Если, скажем, `x=10`, то `!x` равно нулю, если же `x=0`, то `!x` равен единице.

Работа с указателями

В разделе «Указатели» главы 3 мы уже познакомились с операторами получения адреса `&` и раскрытия ссылки `*`. Первый позволяет направить указатель на переменную, а второй — прочитать или изменить ее.

Все это понятно. Не ясно пока, зачем это делать, ведь можно изменить значение переменной без всяких указателей, просто написав переменная=значение. К сожалению, указатели слишком сложны, чтобы сразу понять их красоту и силу. Мы будем знакомиться с ними постепенно и, в конце концов, узнаем и полюбим их.

Начнем с задачи суммирования всех элементов массива. В листинге 4.3 показана программа, которая использует для решения этой задачи указатели.

Листинг 4.3

```
#include <stdio.h>
main(){
int dig[10]={5,3,2,4,6,7,11,17,0,13};
int sum,i;
int *p;
p=&dig[0];
sum=0;
for(i=0;i<10;i++){
     sum += *p;
     p++;
}
printf("sum=%d\n",sum);
}
```

Строчка `p=&dig[0]` получает адрес нулевого элемента массива складываемых чисел. При входе в цикл `for()`

переменная `sum` равна нулю, а указатель `p` указывает на `dig[0]`. Значит, строка

```
sum += *p;
```

прибавит к переменной `sum` значение нулевого элемента массива, то есть `dig[0]`. Далее указатель `p` увеличивается инструкцией `p++`. Я нарочно не сказал «увеличивается на единицу», потому что смысл оператора `++` для указателей иной: указатель ведь не число, это *адрес,* и он увеличивается так, чтобы указывать на *следующий* элемент массива. Здесь, конечно, подходит любая арифметическая операция. Можно написать `p=p+1` — результат будет тот же.

Итак, после увеличения `p++` указатель переключается на следующий элемент массива и строка `sum += *p` прибавит к переменной `sum` число, хранящееся в `dig[1]`— его первом[1] элементе. Дальнейшее не должно вызвать затруднений: программа сложит все десять элементов массива и выведет на экран их сумму `sum`.

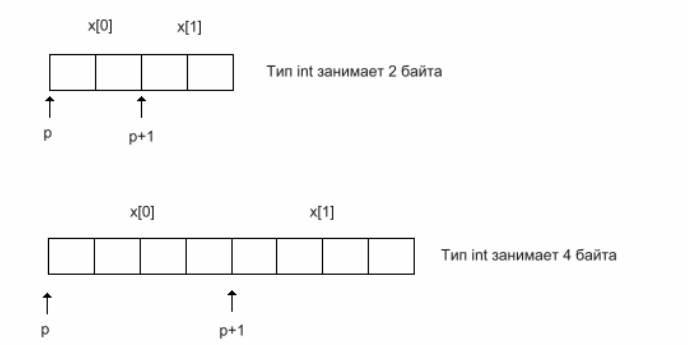

Рис. 4.1 Прибавление единицы к указателю

[1] Нелишне еще раз напомнить: нумерация элементов массива начинается с нуля. Поэтому первым идет элемент dig[0], а вторым — dig[1]

Как мы узнали, к указателям можно прибавлять целые числа. Прибавление единицы перенаправит указатель к следующему элементу того же типа. Если переменная `int` занимает два байта, то указатель переместится на два байта вперед, если `int` занимает 4 байта — на четыре (см. рис. 4.1)

Понятно, что прибавив к указателю двойку, мы переместим его на два элемента вперед, прибавив тройку — на три и т.д. Но если к указателю, поставленному на начало массива из десяти элементов, прибавить 10, он «перепрыгнет» через массив и будет указывать непонятно куда.

Естественно, к указателю можно не только прибавлять, но и вычитать из него числа. Если вычесть единицу, указатель переместится к *предыдущему* элементу массива, но необходимо (как и при сложении) следить, чтобы указатель не вышел за пределы массива.

Указатели, ссылающиеся на элементы одного и того же массива, можно сравнивать. Равные указатели ссылаются на один и тот же элемент массива. Меньший указатель ссылается на элемент, стоящий левее.

Еще указатели можно вычитать. Если, скажем, `p1` указывает на нулевой элемент массива, а `p2` — на четвертый, то их разность `p2-p1` равна четырем (рис. 4.2).

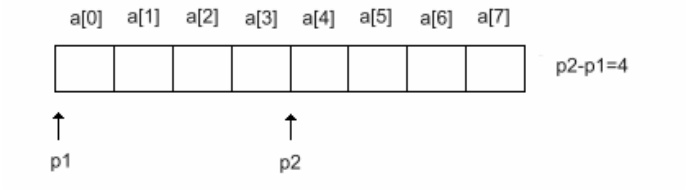

Рис. 4.2. Указатели можно и вычитать

Битовые операции

В прошлой главе говорилось о разных типах переменных, позволяющих экономить память компьютера и при этом соответствовать хранимым данным. Но есть переменные всего

с двумя состояниями, для хранения которых достаточно одного бита. Когда таких переменных много, жалко тратить на каждую из них по байту. Но если упаковать в один байт несколько таких переменных, возникает другая трудность: процессоры не поддерживают доступ к битам. Минимальный размер ячейки памяти — как правило, байт.

Преодолеть эту трудность и помогают специальные логические операции (о них мы уже говорили в этой главе), но примененные уже не к парам переменных, а *к каждой паре бит в этих переменных*.

Пусть, например, нам захотелось узнать, чему равен 5-й (если считать справа налево — от самого младшего бита) бит в переменной `char`. Чтобы решить эту задачу, изготовим сначала специальную битовую маску — вспомогательную переменную типа `unsigned char`, в которой все биты, кроме пятого, равны нулю. Выглядит она так:

```
00010000
```

и записывается в шестнадцатеричном коде (см. раздел «Анатомия типов» в главе 3) как `0x10`. Программа, проверяющая пятый бит, показана в листинге 4.4

Листинг 4.4.

```
#include <stdio.h>
main(){
unsigned char x,mask;
mask=0x10;
x=17;
if(x & mask)
    printf("=1");
else
    printf("=0");
}
```

Центральная часть программы — условная инструкция `if()`, в которой появилась еще одна часть, идущая после слова `else`. Если условие в скобках истинно, то выполняется инструкция `printf("=1")`, если же оно ложно, то выполняется инструкция, стоящая после слова `else`, то есть, `printf("=0")`. Значение выражения `x & mask` — целое число, каждый бит которого получается в результате логической операции `&&` над соответствующими битами переменных `x` и `mask`.

Чтобы, например, получить младший битрезультата операции `x & mask`, нужно взять младший бит `x` (он равен единице) и младший бит `mask`, равный нулю. Результат логической операции `1 && 0` равен нулю. Значит, младший бит результата тоже будет нулем.

Как мы уже знаем, логическое И равно единице, если истинны обе переменные. Но раз в переменной `mask` все биты, кроме пятого, равны нулю, то в результате от нуля может отличаться только пятый бит. В нашем случае `x=17`, в двоичном коде это `00010001`. Значит, пятые биты и в `mask`, и `x` равны единице, и в двоичном коде результат операции `x & mask` будет равен `00010000` (см. рис. 4.3)

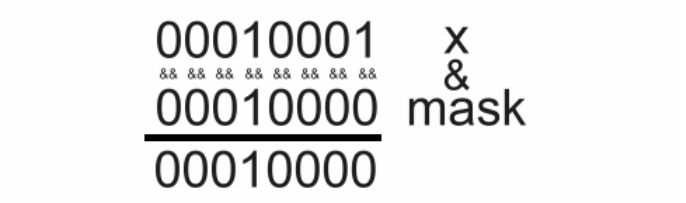

Рис. 4.3. Побитовое И (&)

Так как Си воспринимает любое ненулевое значение как истинное (а `x & mask` в нашем случае не равно нулю), то

выполнится первая ветка инструкции `if()`, и программа выведет на экран `r=1€`. Если бы результат был равен нулю (в случае, когда пятый бит в `x` равен нулю), то выполнялась бы ветка `else`, и программа вывела бы на экран «=0».

Кроме побитовой операции И, есть еще операция ИЛИ, обозначаемая оператором `|`. Нулевой бит результата побитовой операции ИЛИ получается только если оба бита равны нулю. В остальных случаях результат равен единице (рис. 4.4).

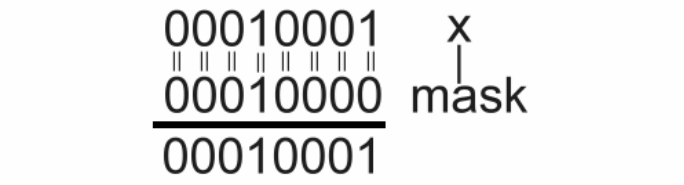

Рис. 4.4. Побитовое ИЛИ (|)

Следующая побитовая операция — *исключающее или* – выделяет в переменных *отличающиеся* биты. Если соответствующие биты различны, получится единица, если одинаковы — ноль. Операция исключающее ИЛИ выполняется оператором ^. Результат применения этого оператора к двум целочисленным переменным `x` и `mask` показан на рис. 4.5.

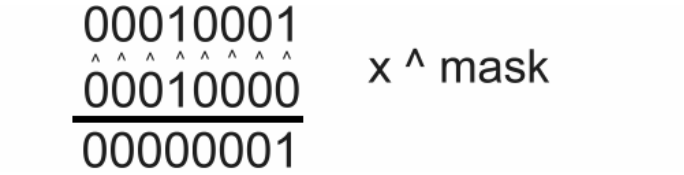

Рис. 4.5. Побитовое «исключающее или»

Задача 4.2 Пусть `a` и `в` – целочисленные переменные. Что делают инструкции `a ^= b; b ^= a; a ^= b;` Почему?

Побитовые операторы, которые мы рассматривали до сих пор, были бинарными, то есть связывали две переменные. Кроме

них, существуют и унарные операторы сдвига >> , << и отрицания ~.

Операторы >> и << сдвигают биты переменной вправо (>>) и влево (<<). При сдвиге влево нулевой бит становится первым, первый — вторым и т.д. Биты, освобождающиеся справа, заполняются нулями, а те, что дошли до левого края, сваливаются и пропадают. В результате сдвига влево на 3 позиции переменной `mask` (записывается как `x = mask << 3`) получится число `x`, равное в двоичном представлении `10000000`. Если сдвинуть `mask` влево на 4 позиции, получится ноль.

При сдвиге вправо для беззнаковой (unsigned) переменной освободившиеся слева биты заполняются нулями, а для переменной со знаком (signed) они могут (в зависимости от компилятора) заполняться как нулями, так и единицами.

Задача 4.3 Выясните, как ведет себя компилятор Turbo C при сдвиге вправо переменных со знаком.

Задача 4.4 Для чего некоторые компиляторы замещают единицами биты, освобождающиеся при сдвиге вправо переменных со знаком?

Задача 4.5 Докажите, используя позиционное представление двоичных чисел, что сдвиг беззнакового числа на один бит вправо равносилен делению на 2.

Последняя рассматриваемая нами битовая операция (ее выполняет оператор побитового отрицания ~) превращает каждый бит переменной в свою противоположность. Если в переменной `mask` только один бит равен единице, то в переменной `x=~mask` только один бит будет равен нулю (тот самый, что был единицей).

Завершим этот раздел тем, ради чего он и был написан — предупреждением об опасности спутать логические и побитовые операторы.

В самом деле, легко вместо && написать &. Компилятор, скорее всего, молча выполнит то, что указано. Но программа при этом

может работать неверно. Например, `4 && 3` равно 1, но `4 & 3` равно нулю.

Ошибка, возникшая от случайной замены логического оператора побитовым, может затаиться при отладке программы и обнаружить себя в самый неожиданный момент. Например, рассмотрим цикл, написанный для нахождения числа `x` в массиве `tab[]`:

`while()`:

```
i = 0;
while (i < tabsize && tab[i] != x)
i++;
```

Всего в массиве `tabsize` чисел, что соответствует изменению индекса от 0 (не забывайте, что нумерация в массивах начинается с нуля!) до `tabsize-1`. Цикл прекращается, либо когда переменная **`i`** (номер элемента в массиве, или индекс) принимает значение **`tabsize`** (то есть индекс выходит за пределы массива), либо когда `tab[i]==x` (то есть в массиве `tab[]` найдено число `x`). Если цикл «прокрутился» до конца (при этом `i` будет равно `tabsize`), то число в массиве не найдено.

Все только что сказанное верно и цикл выполнит порученную ему задачу. Но давайте посмотрим, что будет, если поменять в условии `&&` на `&`:

```
i < tabsize & tab[i] != x
```

Теперь вместо логического оператора `&&` используется `побитовый` оператор `&`, и только благодаря счастливой случайности программа будет работать. Дело в том, что операторы сравнения обладают большим приоритетом, чем побитовый оператор `&` или логический `&&`. При проверке условия будут сначала вычислены выражения `i<tabsize` и `tab[i] != x`. А мы уже знаем, что они равны либо нулю, либо единице. Это значит, что оператор побитового И будет применен к числам, у которых ненулевым может быть только

один, младший бит[1]. А для таких чисел (убедитесь в этом сами) операторы && и & дают один и тот же результат!

Значит, цикл while() выполнит свою задачу даже при замене && на &. Но если один из операндов, связанных оператором &, будет отличен от нуля или единицы, цикл перестанет работать (во всяком случае, правильно).

Функции

Функции, в отличие от операторов, сами не производят каких-либо действий. Они только собирают в одном месте переменные и операторы, что позволяет разбить большую сложную задачу на несколько меньших.

Мы уже немного знакомы со стандартными функциями языка Си, такими как printf(). Теперь пришло время научиться самим создавать функции. И начнем с примитивного, смешного примера — функции, складывающей два целых числа (см. листинг 4.5)

Листинг 4.5.

```
#include <stdio.h>
int add(int, int);
main(){
int i,j,sum;
i=2;
j=3;
sum=add(i,j);
printf("%d\n",sum);
}
```

[1] потому что ноль в двоичном представлении — это (для байтов) 00000000, а единица — 00000001

```
   int add(int a,int b){
   return a+b;
}
```

В самом начале программы функция объявляется как «что-то типа int»

```
int add(int , int );
```

Это подтверждает и строчка `sum=add(i,j)`, в которой значение `add(i,j)` присваивается переменной `sum`. Отличие функции от переменной в том, что функция зависит от аргументов. В нашем случае ей передаются два числа: i и j. Разумно предположить, что функция что-то делает с ними, и в результате этой обработки получается значение типа int, которое присваивается переменной sum.

Внутреннее устройство функции показано в нижней части листинга. Строчка `int add(int a,int b)` — это *заголовок* функции, в котором показаны ее *параметры* и их тип. В теле функции между фигурными скобками показано, что функция делает с параметрами. В нашем случае им присваиваются значения *аргументов* `i` и `j`[1] (параметр `a` становится равным `i`, а параметр `b` —`j`). Затем оба параметра складываются, сумма возвращается во внешний мир инструкцией `return a+b` (`return y` это по-английски

---

[1] То, что стоит в заголовке функции между открывающей и закрывающей скобками называется *параметрами;* В заголовке int add(int a, int b) a и b — параметры. Переменные, передаваемые функции, называются *аргументами*. В нашем случае i и j — аргументы

«возврат») и присваивается переменной sum. Функция printf() в теле основной программы выведет на экран сумму чисел 2 и 3, то есть 5.

Функция add(), с которой мы только что познакомились, обладает, как и любая другая функция, одним замечательным свойством: она никак не зависит от основной программы. Параметры функции a и b никак не связаны с переменными основной программы. Запись функции как бы говорит: «мне на все наплевать, давайте мне две целочисленных переменных, и я возвращу их сумму».

Переменные, пришедшие в функцию из внешнего мира, попадают как бы в виртуальную реальность: функция может использовать их для вычислений, но изменить пришедшие переменные она не в силах. В листинге 4.6 показана программа, которая вызывает функцию change(). Объявление функции void change(int), которое еще называют *прототипом*, начинается словечком «void» (пустая операция), которое говорит компилятору, что функция ничего не возвращает в основную программу. Естественно, в функции, объявленной как void, нет инструкции return.

Листинг 4.6.

```
#include <stdio.h>
void change(int);
main(){
int i;
i=2;
change(i);
printf("вне= %d\n",i);
}
void change(int i){
i+=3;
```

```
printf("внутри= %d\n",i);
}
```

Сама функция принимает извне единственный параметр `i` и увеличивает его на 3. И хоть параметр функции назван так же, как и переменная в основной программе, в результате ее работы получим:

```
внутри= 5
вне= 2
```

Понять это можно так: *в функцию передается не сама переменная, а ее копия, которая существует только внутри функции и,* выполнив свою задачу, исчезает после возврата в основную программу. Функция не в силах изменить внешнюю переменную, потому что не имеет к ней доступа.

Точно так же и вызывающая программа не имеет доступа к переменным, объявленным внутри функции. Функция и программа общаются друг с другом на ощупь, они не видят друг друга, и это хорошо, потому что снижает вероятность ошибки и облегчает ее поиск.

Начинающие программисты часто пишут программу одним сплошным куском. Пока она умещается на экране компьютера, это удобно. Но стоит программе превысить, скажем, 100 строк, и становится трудно охватить взглядом все переменные. Значит, возрастает вероятность того, что изменение переменной в одной части программы может самым неожиданным образом повлиять на другую ее часть.

Чтобы этого не произошло, программу лучше разбить на отдельные модули — функции, которые не должны (в идеале) зависеть друг от друга. Программисты стремятся написать программу так, чтобы каждая функция выполняла какой-то определенный кусок работы. Идеальную функцию можно, как кирпичик, использовать в других программах. Примеры таких функций всегда перед нами — это прежде всего стандартные функции языка Си (`printf()`, например).

И самая стандартная из всех — функция `main()`, которая должна быть в каждой программе. Как и всякая функция,

`main()` возвращает какое-то значение, и мы отступали до сих пор от стандарта языка, не указывая его тип левее «`main`». Стандарт говорит нам, что функция `main()` возвращает значение `int`, поэтому правильнее будет писать

```
int main(){
…
return 0;
}
```

Функция `main()` — главная в программе. И она возвращает значение тому, кто главнее, то есть запустившей ее процедуре операционной системы. Далее во всех программах мы будем писать `return 0;` не задумываясь о судьбе возвращенного значения — просто потому, что это правильно и одобрено стандартом.

Функции с длинными руками

Только что мы видели, как функция `change()` (см. листинг 4.6 в предыдущем разделе), силится «схватить» переменную i, но — руки коротки! Все, чем располагает функция — лишь *копия* передаваемой переменной — сама переменная, живущая в основной программе, ей недоступна.

Но давайте подумаем, что будет, если передать функции не переменную, а *указатель* на нее. Поскольку функция получает копию переменной, она не может повлиять на сам указатель. Но она может записать все, что угодно туда, куда он направлен.

В листинге 4.7 показана программа, которая, как и в предыдущем пункте, пытается изменить переменную `i`.

Листинг 4.7.

```
#include <stdio.h>
void change(int *);
int main(){
```

```
int i;
i=2;
change(&i);
printf("вне= %d\n",i);
return 0;
}
void change(int *a){
*a += 3;
}
```

На этот раз успешно. Объявление (прототип) функции `void change(int *)` показывает, что она принимает указатель на переменную `int`. Указатель (вернее, его копия) передается функции в строчке `change(&i)`. Внутри функции копия указателя называется `a`, и теперь раскрытие ссылки `*a` позволяет как угодно изменить переменную, *на которую ссылается указатель a*. Строчка `*a += 3` прибавляет к переменной `i` основной программы тройку. Теперь после вызова функции `change(&i)` функция `printf()` выводит на экран пятерку, а не двойку.

Переданный указатель похож на веревочку, которая связывает две функции (вызвавшую и вызванную). Дерни за веревочку, — откроется дверка к переменной, и вызванная функция сможет ее изменить. Впрочем, она может изменить и соседнюю переменную, если увеличит указатель на единицу. Здесь таится большая опасность, потому что функция ничего не знает о внешних переменных, в частности, ей не известно, указывает ли `(a+1)` на что-то действительно существующее. Нечаянно изменив указатель и записав что-то по новому адресу, она может испортить данные или всю программу.

Как мы уже поняли, передавать указатели опасно, но иногда без этого не обойтись. Например, в функциях, которые вводят данные с клавиатуры. Одна из таких функций — `scanf()` — делает работу, противоположную работе функции `printf()`.

Если `printf()` выводит на экране переменные, то `scanf()` вводит их с клавиатуры и отображает на экране.

В листинге 4.8 показана программа, которая складывает два вводимых с клавиатуры числа.

Листинг 4.8

```
#include <stdio.h>
int main(){
int a,b;
scanf("%d%d", &a,&b);
printf("\n%d\n", a+b);
return 0;
}
```

Собственно вводом занимается функция `scanf("%d%d", &a,&b)`, очень похожая на `printf()`. В ней тоже есть спецификация формата `"%d%d"`, говорящая о том, что будут введены два целых числа. Но, в отличие от `printf()`, ей передаются не сами переменные, которые функция не в состоянии изменить, а *указатели* на них.

Чтобы получить сумму чисел, нужно после запуска программы ввести с клавиатуры первое число, нажать Enter , затем ввести второе число, и после нажатия Enter функция `printf()` покажет на экране сумму. Можно ввести числа и на одной строке, для чего печатается одно число, пробел, второе число и нажимается Enter.

Как мы уже поняли, указатель в руках функции становится опасным: с его помощью она может дотянуться не только до переменной, на которую он указывает, но и до чего угодно. «Укоротить руки» функции способно словечко `const` (от английского константа, постоянная величина).

Если функцию `change()`, показанную в листинге 4.10, записать как

```
void change(const int *a){
```

```
        *a += 3;
}
```

компилятор выдаст сообщение об ошибке: Cannot modify a const object in function change. Это значит, что объект, на который ссылается указатель, постоянен и не может быть изменен.

Конечно, в функции `change()`, созданной именно для того, чтобы *менять* переменную, словечко `const` использовать нельзя. Но бывают функции, которым нужно только *читать* переменную, зная указатель на нее. Здесь слово `const` уместно и делает программу надежней и безопасней.

Рекурсия или «раз, два, три»

Не следует надолго уходить в себя, так как близкие нуждаются в общении с вами.

Из гороскопа.

Начнем этот раздел с простой задачки: вывести на экран три числа: 1, 2, 3. Есть много способов ее решения. Один из самых очевидных показан в листинге 4.9:

Листинг 4.9

```
#include <stdio.h>
int main()
{
int i;
for(i=1;i<4;i++)
        printf("%d\n",i);
return 0;
}
```

Эта программа, успешно решает одну простую задачу. Приспособить ее к чему-то еще (кроме, быть может, вывода на экран многих идущих подряд чисел) невозможно — слишком

уж она буквальна и прямолинейна. Чтобы будить мысль и куда-то вести, ей, очевидно, недостает той доли безумия, которая есть в программе, показанной в листинге 4.10.

Листинг 4.10

```
#include <stdio.h>
void CntTo3(int);
void CntTo2(int);
void CntTo1(int);
int main()
{
int n;
CntTo3(3);
return 0;
}
void CntTo3(int p)
{
    CntTo2(p-1);
    printf("%d\n",p);
}
void CntTo2(int p)
{
    CntTo1(p-1);
    printf("%d\n",p);
}
void CntTo1(int p)
{
    printf("%d\n",p);
```

```
}
```

Эта программа тоже выводит на экран три идущих подряд числа, но делает это совсем по-другому. Если посмотреть устройство функции `CntTo3(p)`, решающей задачу вывода чисел 1, 2, 3, то окажется, что она разбивает задачу на две части:

1. Вывести числа 1, 2 (функция `CntTo2()`)
2. Вывести число 3.

В свою очередь, функция `CntTo2()` использует такое же разбиение задачи:

1. Вывести число 1 (функция `CntTo1()`)
2. Вывести число 2.

И только функции `CntTo1()` уже некуда отступать. Она выводит на экран единицу и возвращает управление функции `CntTo2()`, которая в свою очередь показывает двойку (ведь именно таково значение переданного ей аргумента `p`) и возвращает управление функции `CntTo3()`, которая выводит тройку — значение переданного *ей* аргумента `p`, после чего `CntTo3()` передает управление вызвавшей ее функции `main()`, и программа благополучно завершается.

```
#include <stdio.h>

int CntTo3(int p)
        {
    int sum;

        sum = 0;
sum = p + CntTo2(p - 1);
        }
int CntTo2(int p)
        {
    int sum;

        sum = 0;
sum = p + CntTo1(p - 1);
        }
int CntTo1(int p)
        {
    int sum;

        sum = p;
        }

    int main()
        {
        int n;
printf("%d", CntTo3(3));
        return 0;
        }
```

Задача 4.6 Напишите программу, которая складывает три числа: 1+2+3 — примерно такую, как в листинге 4.10.

Теперь можно немножко обобщить нашу задачу. Предположим, необходимо вывести на экран `n` подряд идущих чисел от `1` до `n`. Эту задачу можно разбить на две:

1. Вывести числа от `1` до `n-1`
2. Вывести n

Точно так же задача вывода чисел от `1` до `n-1` разбивается на две:

1. Вывести числа от `1` до `n-2`
2. Вывести `n-1`

и так далее, вплоть до единицы.

Такой способ решения задачи называется *рекурсией*. По сути, рекурсия — частный случай стратегии «разделяй и властвуй». Если от задачи можно «отколоть» кусочек, то нужно это сделать, потому что оставшаяся задача будет проще.

Но как решать рекурсивные задачи на компьютере? Приведенный способ (см. листинг 4.10) годится для вывода трех чисел, но для вывода сотни он слишком громоздок и попросту глуп.

Но давайте все-таки представим, как выглядела бы программа, показанная в листинге 4.10, если бы ее задачей было вывести не три подряд идущих числа, а сотню. Очевидно, такая программа принципиально мало отличалась бы от той, что выводит три числа. Только вместо трех функций в ней было бы сто!

Но вот что интересно: 99 из этих ста функций выполняли бы *абсолютно одинаковые действия*:

```
void CntTo100(int p)
{
    CntTo99(p-1);
    printf("%d\n",p);
}
void CntTo99(int p)
{
    CntTo98(p-1);
    printf("%d\n",p);
}
…
void CntTo2(int p)
{
```

```
    CntTo1(p-1);
    printf("%d\n",p);
}
```

и только последняя, оказавшись «крайней» просто выводила бы на экран единицу!

То, что все эти 99 функций по сути, отличаются только названием, наводит на мысль: а нельзя ли записать это в виде одной единственной функции, которая 99 раз *обращается сама к себе*?

И такая возможность действительно есть, причем у любой функции, даже у `main()`. Функции, которые вызывают сами себя, так же как задачи, для решения которых они созданы, называются *рекурсивными*. Глядя на листинг 4.10, легко написать программу, использующую рекурсивную функцию для вывода идущих подряд цифр. Выглядеть она может так, как показано в листинге 4.11.

Листинг 4.11

```
#include <stdio.h>
void CntTo(int);
int main()
{
int n;
CntTo(3);
return 0;
}
void CntTo(int n)
{
    if(n > 0){
        CntTo(n-1);
        printf("%d\n",n);
```

```
        }
    }
```

Используемая здесь функция `CntTo()`может вывести на экран любую последовательность чисел, но для простоты мы по-прежнему будем выводить «1,2,3».

При первом обращении к функции `CntTo()`значение аргумента равно 3, и так как выражение `(n > 0)` истинно, `CntTo()` вызовется еще раз, но уже с аргументом `n-1`, равным двойке. Далее, раз двойка больше нуля, то `CntTo()` вызовется в третий раз, уже с единичным аргументом. Единица тоже больше нуля, и `CntTo()` будет вызвана в четвертый раз. На этот раз `n` равно `0`, условие `(n>0)` не выполнится, и следующего вызова `CntTo()` не будет. Вместо вызова, произойдет *возврат*. Но куда? Чтобы понять это, полезно еще раз посмотреть цепочку функций `CntTo3`, `CntTo2`, `CntTo1` в листинге 4.10., которая показывает нам, что же на самом деле творится при рекурсивном вызове.

Происходит странное: вызывается функция, которая называется так же, как и вызывающая, но на самом деле она *другая*. У нее свои переменные, независимые от переменных вызвавшей функции. Переменная, объявленная в вызывающей функции, никак не связана с одноименной переменной в вызванной функции. И когда управление передается вызвавшей функции, мы имеем дело со значением переменной именно в этой функции.

В нашем случае после четвертого вызова, когда n было равно нулю, функция возвращается к инструкции, следующей непосредственно за вызовом, то есть к `printf("%d\n",n)` Причем, `n` в данном случае — не какое-то абстрактное число, а значение параметра `n` *в третьей копии* `CntTo()`, то есть единица. Показав на экране «1», третья копия `CntTo()` вернется к вызвавшей ее второй копии, опять в то место, которое следует непосредственно за вызовом, то есть снова к `printf("%d\n",n)`, но на этот раз n будет равно двум. Наконец, при третьем возврате будет выведена тройка.

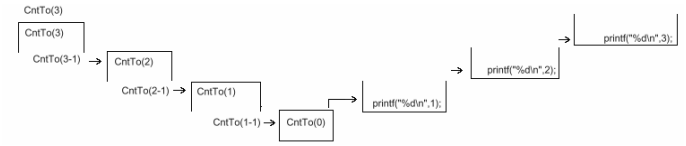

Рис. 4.6. Закат и восход рекурсивной функции

На рис. 4.6 показано, как функция `CntTo()` «углубляется в себя». Если бы не условие `if(n >0){}`, функция ушла бы в себя и не вернулась. Поскольку вызов каждой копии функции требует памяти для хранения переменных, то после исчерпания памяти, программа должна была бы аварийно завершиться. Инструкция `if()` разрешает функции вызвать себя только четыре раза. С каждым разом функция уходит все глубже в себя, она «закатывается». Когда `n` становится равным нулю, начинается «восход». Функция возвращается во все свои более ранние копии, пока не высветятся все три числа. Затем она взойдет еще на одну ступеньку — и окажется в функции `main()`.

Завершим этот раздел еще одним примером рекурсивной функции, на этот раз — складывающей все числа от единицы до ста. Задача сложения идущих подряд чисел может быть решена в два этапа: сумма чисел от `1` до `n` равна `n` плюс сумма от `1` до `n-1`. Эта последняя сумма равна `n-1` плюс сумма чисел от `1` до `n-2` — и так до самой последней суммы от единицы до единицы. Программа, которая все это делает, показана в листинге 4.12.

Листинг 4.12.

```
#include <stdio.h>
int sum(int);
int main(){
int i;
printf("%d\n",sum(100));
return 0;
```

```
}
int sum(int n){
if(n == 1)
     return 1;
else
     return sum(n-1) + n;
}
```

```
#include <stdio.h>

int sum(int n)
{

if (n <= 0)
return 0;
else if (n > 0)
return sum(n-1) + n;
}
int main()
{
int n;

n = 100;
printf("сумма %d подряд идущих чисел равна = %d", n, sum(n));
return 0;
}
```

В этой программе функция `sum()` вызывает себя до тех пор, пока аргумент не станет равен единице. Тогда выполнится условие `n==1` и функция возвратит единицу *той своей копии, в которой n равно двум*. Значит, «вверх» пойдет тройка (`return 1+2`); Тройка возвратится в ту копию функции, где `n=3`, значит число 6 (`return 3+3`) будет возвращено туда, где `n=4` и т.д., до возврата в функцию `main()`. Функция `sum()` как бы спускается в глубокую шахту, расставляя на этажах цифры. На верхнем будет цифра 100, дальше, соответственно, 99, 98 и т.д. В самом низу — единица.

Расставив числа, функция начнет движение вверх, на ходу подбирая цифры и складывая их. С нижнего этажа она возьмет единицу и прибавит к двойке, находящейся на втором этаже. Чуть выше будут найдены и прибавлены тройка, четверка и т.д. Когда сумма будет вычислена, мы окажемся на поверхности, в функции `main()`.

В заключение приведем еще одну версию программы, суммирующей числа от 1 до 100, в которой условная инструкция `if()Цelse` заменена *условным выражением*:

```
#include <stdio.h>
int sum(int);
int main(){
int i;
printf("%d\n",sum(100));
return 0;
```

```
}
int sum(int n){
return (n == 1) ? 1 : sum(n-1) + n;
}
```

Конструкция `(n == 1) ? 1 : sum(n-1) + n` состоит из условия, которое помещается левее вопросительного знака и двух выражений, разделенных двоеточием:

```
условие ? выр1 : выр2
```

Если `условие` выполняется, вычисляется `выр1`, если нет — `выр2`. Например, с помощью условного выражения вычисление максимума двух чисел можно записать так:

```
max = (x1 > x2) ? x1 : x2;
```

По сравнению с конструкцией, которую мы рассматривали раньше

```
if(x1 > x2)
      max=x1;
else
      max=x2;
```

условное выражение более компактно, и хорошо смотрится при записи в одну строчку.

Глава 5. Функции, указатели, массивы

Функции и массивы

Мы уже научились передавать функции переменные (в том случае, когда необходимо только читать их) и указатели (когда нужно менять переменные, на которые настроен указатель). Но как передать функции массив? Передать ей все элементы массива или указатели на все элементы?

Ни то ни другое. Функции достаточно передать указатель на нулевой элемент массива и число элементов в нем. Тогда она сможет менять указатель в безопасных пределах и получит полный доступ к любому элементу массива.

Посмотрим, например, как может выглядеть функция, которая определяет максимальный элемент массива. Разумно спроектировать ее так, чтобы она принимала указатель на нулевой элемент массива и число элементов в нем, а возвращала максимальное значение. Прототип функции будет при этом таким:

```
int maxi(int *a, int n).
```

В листинге 5.1 показана программа, которая использует функцию `maxi()` для нахождения максимального элемента массива:

Листинг 5.1

```
#include <stdio.h>
int maxi(int *, int);
int main(){
int dig[10]={5,3,2,4,6,7,11,17,0,13};
int m;
m=maxi(&dig[0],10);
```

```
printf("%d\n",m);
return 0;
}
int maxi(int *a,int n){
int i,max;
max=*a;
for(i=1;i<n;i++)
   if(*(a+i) > max)
      max=*(a+i);
return max;
}
```

В самом начале программы показан прототип функции, `int maxi(int *, int)`, сообщающий о том, что функция принимает указатель на `int` и целое `int`, возвращает также целое.

Возвращаемое функцией значение присваивается переменной `m: m=maxi(&dig[0],10)`. Как видим, функции передается адрес нулевого элемента массива `&dig[0]` и число элементов в нем (`10`). Попав внутрь функции, эти аргументы превращаются в указатель `a` и целочисленную переменную `n`. Далее переменной `max` присваивается значение нулевого элемента массива: `max=*a`. Затем в цикле перебираются все остальные элементы с номерами от 1 до 9. В строке `if(*(a+i) > max)` сравнивается текущее значение переменной `max` со значением элемента `i` (напомним, что согласно правилам работы с указателями, `a+i` *указывает* на i-й элемент массива, а раскрытие ссылки `*(a+i)` дает *значение* этого элемента). Если текущий элемент массива больше `max`, то он становится новым значением максимума: max=текущее значение, то есть `max=*(a+i)`. В конце концов, значение `max` возвращается во внешний мир и выводится на экран.

Очень поучительно сравнить программу, показанную в листинге 5.1 с программой из раздела «Массивы» второй главы (листинг 2.4), также вычисляющей максимальное значение в массиве, но без использования функций. Программа, из листинга 2.4, как одноразовый стаканчик, годится для решения единственной задачи: нахождения максимального элемента массива `dig[]`, состоящего из десяти элементов. А программа, показанная в листинге 5.1 (вернее, функция `maxi()`), уже способна вычислить максимальный элемент в *любом* целочисленном массиве!

Задача 5.1 Подумайте, как можно сконструировать функцию, которая возвращает не только максимальное значение, но и номер наибольшего элемента в массиве.

Массивы и указатели

Выводя на экран содержимое массива (см. раздел «Строки и символы» в главе 3), мы уже поняли, что функции `printf()` можно вместо указателя на начало строки передать само имя строки. Передача имени вместо указателя, оказывается, возможна не только для строк, но и для любых массивов и любых функций, принимающих указатели. Так, в программе из листинга 5.1 можно было бы вместо

```
m=maxi(&dig[0],10)
```

написать просто `m=maxi(dig,10)`.

Это значит, что компилятор рассматривает имя массива как *адрес его нулевого элемента*. И он поступает с именем массива так, как будто это указатель! Встретив какой-то элемент массива, например `arr[i]`, компилятор преобразует его по правилам работы с указателями: `*(arr + i)`. То есть, `arr` — это *как бы указатель*, `i` — целочисленная переменная. Сумма `arr+i` указывает на i-й элемент массива, а оператор раскрытия ссылки `*` дает значение самого элемента. На рис. 5.1 показан массив `x`, в котором хранятся три целых числа — 13, 28 и 5, и некоторые комбинации имен элементов, указателей и связанных с ними операторов `&` и `*`.

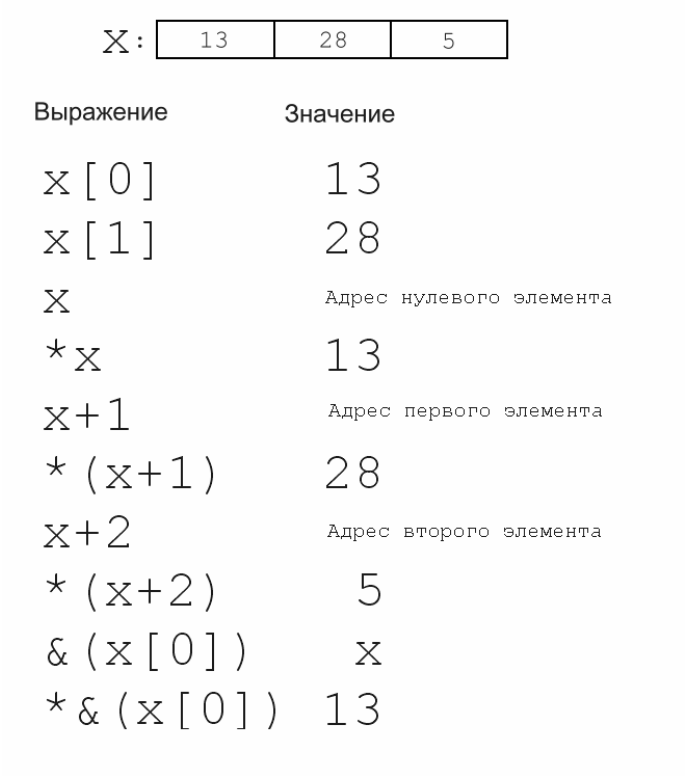

Рис. 5.1. Действия с именем массива и его элементами

Самая важная строчка на рисунке — предпоследняя. Она показывает, что адрес нулевого элемента массива `&(x[0])` равен имени массива `x`.

Преобразование `x[0]` в `*x`, показанное на рис. 5.1, может навести на мысль, что имя массива — *и есть* указатель на его нулевой элемент. Но это не так. Отличие имени массива от указателя в том, что имя *не может быть изменено*. Пусть имеется массив `x[i]`. Как мы теперь знаем, его нулевой элемент равен `*x`, первый `*(x+1)` и т.д. Но если бы `x` был настоящим указателем, его значение можно было бы

увеличить на единицу `x=x+1` и тогда `x` указывал бы на первый элемент массива. Ничего подобного с именем массива сделать нельзя. Оно всегда указывает в одно и то же место — на свой нулевой элемент.

Различие между указателями и массивами легче понять, если сравнить два объявления строки, с которыми мы встретились в главе 3 (разделы «Строки и символы» и «Указатели» главы 3):

```
char a[]="MAMA";

char *p="MAMA"; ,
```

Первая строка объявляет массив `a[]`, в котором 5 элементов: четыре буквы и завершающий символ `'\0'`. Адрес нулевого элемента определяется во время компиляции, и в программе нет такой переменной, где бы он хранился. Когда компилятор видит в строке `printf("%s",a)` имя массива a, то подставляет вместо него адрес начала строки.

Во втором случае адрес начала строки хранится в *переменной* p (указателе). Эту строку можно вывести на экран так же, как и первую, передав функции `printf()` значение указателя, хранящееся в переменной `p`:

```
printf("%s",p).
```

Но в указатель `p`, как и в любую переменную, можно записать новое значение и тогда `p` будет указывать на что-то другое.

Указатели и массивы

Если значение i-го элемента массива `x[i]` равно `*(x + i)`, то логично предположить, что всякий фрагмент программы `*(p + i)`, где `p` — указатель, а `i` — целочисленная переменная, можно записать как `p[i]`. И это, к чести языка Си, действительно так, потому что компилятор, встретив в программе запись `p[i]` преобразует ее к виду `*(p+i)` независимо от того, что такое `p` — массив или указатель.

Значит, в функции `maxi()`, (см. листинг 5.1) можно указатель `a` в выражении `*(a+i)` заменить на `a[i]` и переписать всю функцию следующим образом:

```
int maxi(int *a,int n){
int i,max;
max=a[0];
for(i=1;i<n;i++)
     if(a[i] > max)
         max=a[i];
return max;
}
```

Писать `*(a+i)` или же `a[i]`— дело вкуса, хотя те, кто программировал на Бейсике, предпочтут, скорее всего, второй способ.

Завершим этот раздел еще одним вариантом программы, суммирующей все элементы массива (см. листинг 5.2). От предыдущей версии (см. листинг 4.3) ее отличает прежде всего строка `int *p=dig`, передающая адрес нулевого элемента массива «настоящему» указателю `p`. Этот указатель `p` (в отличие от имени массива) можно менять так, чтобы он указывал последовательно на все элементы массива.

Листинг 5.2

```
#include <stdio.h>
main(){
int dig[10]={5,3,2,4,6,7,11,17,0,13};
int sum=0,i;
int *p=dig;
for(i=0;i<10;i++){
     sum += *p++;
printf("sum=%d\n",sum);
}
```

Второе отличие — в строке `sum += *p++` , совмещающей раскрытие ссылки и увеличение указателя.Такая запись выглядит устрашающе, но для программистов, хорошо знакомых с языком Си, она проста и понятна. В сущности, запись `sum +=*p++` означает, что к `sum` прибавляется значение `*p`, а затем `p` увеличивается на единицу и указывает на следующий элемент массива.

Динамические массивы

Предположим, что в переменной `p` хранится указатель (скажем, на `int`), причем, известно, что не только `p`, но и `p+1`, `p+2`, `p+3`, …. `p+k-1` указывают на области памяти, откуда программа может читать текущие значения переменных и куда может записывать их новые значения.

Можно сказать, что в этом случае программе доступны `k` *идущих подряд* безымянных переменных, с каждой из которых нас связывает только указатель. К переменной, на которую указывает `p+1`, можно обратиться как `*(p+1)`, а к переменной, на которую указывает `p+k-1` — как `*(p+k-1)`.

Но что такое эти `k` переменных, занимающих сплошной кусок памяти? Очевидно, это *массив* и если `p` — указатель на его начало, то любой элемент массива можно записать как `p[i]`, где `i` меняется от `0` до `k-1`.

Массивы в виде сплошного куска памяти с указателем на его начало, играют весьма важную роль в языке Си, и для их создания предусмотрена специальная функция `malloc()`, принимающая число байт, занимаемых массивом, и возвращающая указатель на его нулевой элемент. Массив из десяти переменных типа `int` можно выделить с помощью `malloc()` (m от memory — память, alloc — выделение) следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
```

```
int *p,i;
p=malloc(10*sizeof(int));
for(i=0;i<10;i++)
    printf("%d\n",p[i]);
return 0;
}
```

Как мы уже знаем, каждая функция в языке Си должна быть описана. Описание функции `malloc()` находится в файле `stdlib.h`, который подключается к тексту программы строчкой `#include <stdlib.h>`. Здесь мы впервые сталкиваемся с тем, что к программе можно подключить два (или более) заголовочных файла.

Еще раз обратим внимание на то, что возвращенный `malloc()` указатель можно использовать как имя массива[1], что и делает функция `printf()`, показывающая каждый его элемент `p[i]`[2]. Но (в отличие от настоящего массива) этот указатель можно менять.

Выражение `10*sizeof(int)`, указывает функции `malloc()`, сколько нужно выделить байтов памяти для хранения десяти элементов массива. Оно равно произведению числа элементов (10) на размер одного элемента в байтах: `sizeof(int)`. Если программа компилируется в Turbo C, то `sizeof(int)` равен двум. Можно было бы написать `p=malloc(20)`, но делать так было бы неразумно, потому что при переходе к другому компилятору, где переменная `int` занимает 4, а то и 8 байт, `malloc(20)` выделит не 10, а только 5 (или даже две) переменных типа `int`, и программа, скорее всего, работать не будет.

---

[1] Потому что компилятор, согласно правилам языка С преобразует p[i] в *(p+i)

[2] В каждом элементе массива, созданного с помощью `malloc()`, содержится «мусор»

Откажется она работать и когда malloc() не сможет выделить требуемое количество памяти. Память под 10 переменных типа int, скорее всего, найдется, но если переменных миллионы, ее может просто не хватить. В этом случае malloc() возвратит специальный нулевой указатель NULL, о котором известно, что он не может в принципе указывать на какую-то область памяти. Это и скажет нам, что память выделить не удалось. Для надежной работы программы нужно всегда проверять, не равен ли значению NULL указатель, возвращенный malloc(). Программа с такой проверкой может выглядеть так:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
int *p,i;
p=malloc(10*sizeof(int));
if(p == NULL){
    printf("Недостаточно памяти!\n");
    return 1;
}
for(i=0;i<10;i++)
    printf("%d\n",p[i]);
return 0;
}
```

Проверку на NULL и выделение памяти можно совместить в одной строчке:

```
if(NULL == (p = malloc(10*sizeof(int)))){
    printf("Недостаточно памяти!\n");
    return 1;
}
```

Выражение `NULL == (p = malloc(10*sizeof(int)))` вычисляется справа налево: сначала указателю `p` присваивается значение `malloc(20*sizeof(int))`, затем проверяется, не равен ли указатель значению `NULL`. Если равен, то печатается сообщение «Недостаточно памяти!» и программа завершается, если нет, то выполняется дальше.

Память, выделенная с помощью `malloc()`, может быть освобождена функцией `free()`(от английского — освободить):

```
if(NULL == (p = malloc(10*sizeof(int)))){
    printf("Недостаточно памяти!\n");
    return 1;
}
...
/* работа с массивом  p[]    */
free(p);
return 0;
```

Побывав в лапах функции `free()`, указатель теряет свои полезные свойства. Раньше он был направлен на свободную область памяти, но после `free()` показывает пальцем в небо. Попытка записать `p[i] = rчто-тоЄ` после освобождения памяти закончится, скорее всего, крахом программы, и это — одна из самых распространенных ошибок при программировании на Си.

Копирование строк

Если работа с `malloc()` и `free()` так опасна, то, может быть, вообще не стоит пользоваться этими функциями, а лучше задавать массивы привычным способом: `int p[1000000]`?

К сожалению, память, занятую таким массивом, нельзя освободить, кроме того, нельзя изменить и размер массива в процессе выполнения программы. Поэтому говорят, что так

задается «статический массив». Но далеко не всегда известен точный размер массива, и чтобы программа работала, нужно задавать его «с запасом», что некрасиво и расточительно.

В отличие от статического массива, задание динамического массива с помощью функции `malloc()` позволяет выделить ровно столько памяти, сколько нужно, а функция `free()` освобождает память сразу же, как только нужда в ней отпадет.

Для иллюстрации сказанного напишем программу, которая копирует строки (см. листинг 5.3).

Листинг 5.3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(){
int i;
char *p1="Используйте malloc()";
char *p2;
if(NULL == (p2 = malloc(strlen(p1)+1))){
    printf("Недостаточно памяти!\n");
    return 1;
}
for(i=0;i<=strlen(p1);i++)
    p2[i]=p1[i];

printf("%s\n",p2);
free(p2);
return 0;
}
```

Строка «Используйте malloc()» помещена в область, на которую указывает p1. Наша задача — скопировать ее в другую область, на которую указывает p2. Строчка char *p2 выделяет память только *для указателя*, но не для самой строки! Первоначально p2 содержит «мусор». Чтобы p2 указывал на область памяти, где может поместиться копия строки, его надо инициализировать функцией malloc():

```
p2=malloc(strlen(p1)+1);
```

Функция strlen(), описанная в заголовочном файле string.h, принимает указатель на строку и возвращает число символов в ней, при этом завершающий символ '\0' не считается. Значит, malloc() должна выделить для копии строки strlen(p1) + 1 байт.

После выделения памяти можно копировать строку, что и делается в цикле

```
for(i=0;i<=strlen(p1);i++)
    p2[i]=p1[i];
```

Пользуясь тем, что p1 и p2 — указатели, этот цикл можно переписать так:

```
for(i=0;i<=strlen(p1);i++)
    *(p2+i)=*(p1+i);
```

После копирования строки программа выводит ее на экран с помощью функции printf("%s\n",p2), освобождает память, занятую копией строки: free(p2), возвращает 0 и на этом заканчивает работу.

Программа, показанная в листинге 5.3, копирует строки самостоятельно — в цикле for(). Но обычно для этого используется стандартная функция strcpy(). Копирование строки p1 в p2 (для краткости говорят «строка p1» вместо «строка, на начало которой указывает p1») при использовании стандартной функции будет выглядеть так:

```
strcpy(p2,p1);
```

Очень важно понимать, что стандартные функции языка Си, работающие со строками, никак не заботятся о выделении памяти, и программа

```
#include <stdio.h>
#include <string.h>
int main(){
char *p1="Используйте malloc()";
char *p2;
strcpy(p2,p1);
printf("%s\n",p2);
return 0;
}
```

работать не будет, потому что `p2` содержит «мусор» и, следовательно, не указывает на непрерывную область памяти, куда может поместиться копия строки. О стандартной функции языка Си, как о ребенке, приходится заботиться: передать ей нужные параметры, выделить память и только после этого она сделает свою часть работы — хорошо и быстро.

В заключении заметим, что заводить специальный указатель `p1` для строки `char *p1=""` вовсе не обязательно. Функции `strcpy()`, как и любой другой функции, можно передать саму строку: `strcpy(p2, "Используйте malloc()")`. Встретив строку в списке параметров функции или при объявлении указателя, `char *p1="..."`, компилятор помещает символы в некую область памяти, а функции или указателю передает адрес начала этой области.

Задача 5.2 Напишите собственные версии функций `strlen()` и `strcpy()`.

Глава 6 Файлы

Падение железного занавеса

До сих пор наши программы могли только принимать символы с клавиатуры и высвечивать их на экране монитора. Они как бы находились в заточении и могли лишь перестукиваться с соседом по камере, в то время как огромный и прекрасный внешний мир оставался недоступен. Настало время вышибить тюремную дверь и соединиться с миром файлов, откуда программа берет необходимую информацию и куда возвращает результаты своей работы.

Файлами, как вы знаете, называется то, что очень быстро забивает жесткий диск любого размера. У каждого файла есть имя, обычно состоящее из букв, разделенных точками, например: sort.c , dog.jpg и т.д. Буквы, стоящие правее последней точки, называются *расширением* и по ним часто можно узнать, информация какого вида хранится в файле. Так, расширение .jpg обычно бывает у файлов, хранящих изображения, а расширение .c говорит о том, что это исходный текст программы на Си.

И наш первый опыт работы с файлами как раз и будет связан с этими исходными текстами. Дело в том, что при отладке программ программистам необходимо многократно просматривать исходные тексты, и чем больше строк удается увидеть, тем лучше. Для просмотра подходят редакторы Блокнот (Notepad) и улучшенный Notepad — Ecopad32. В них можно выбрать более мелкий шрифт и читать столько строк текста, сколько удастся разглядеть. Но, к сожалению, текст в этих редакторах прижимается к левому краю экрана, что затрудняет чтение, а отодвинуть его вправо нет возможности. Поэтому была бы полезна программа, которая читает исходный текст на Си, сдвигает его вправо и записывает в другой файл. Чтобы написать ее, нужно научиться работать с файлами, что совсем не сложно.

Программа на языке Си общается с файлом с помощью специальной переменной — *указателя файла*, которая объявляется как

```
FILE *in;
```

То есть, формально создается указатель на тип `FILE`, которому нужно присвоить определенное значение функцией `fopen()`:

```
in=fopen("имя", "режим");
```

где «режим» определяет характер работы с файлом, а «имя» должно должно быть именем реального файла, иначе функция `fopen()` возвратит `NULL`. Соединение программы с файлом может выглядеть так:

```
FILE *in;
in = fopen("имя", "rb");
if (in == NULL){
     printf("Ошибка %s\n","имя");
     return 1;
}
```

Режим `"rb"` означает, что файл открывается только для чтения (`r`) (в целях безопасности) и рассматривается как бинарный (`b`), то есть воспринимается «как есть» со всем своим содержимым[1]. После успешного открытия файла программа получает в свое распоряжение переменную `in`, которая передается как параметр различным функциям.

В этом разделе нас будут интересовать две функции, работающие с файлами: функция, читающая строку из файла и функция, записывающая строку в файл.

---

[1] Кроме бинарного есть еще странный текстовый режим, в котором программа без нашего спроса будет менять некоторые символы. Им мы пользоваться не будем.

Но прежде чем переходить к знакомству с ними, нужно, пожалуй, сделать небольшое отступление и рассказать о том, что текстовые файлы, к которым относятся и тексты программ, состоят из отдельных строк. Каждая строка заканчивается специальными символами, которые показывают программе просмотра (такой как Блокнот), что следующие символы нужно показывать с новой строки. В разных системах символы окончания строки разные, но в DOS и Windows строку завершают два невидимых байта `0D0A` (в шестнадцатеричном представлении).

После знакомства со строками, можно перейти к стандартным функциям чтения строки из файла `fgets()` и записи строки в файл `fputs()`.

Первая из них имеет три параметра:

```
fgets(char * buf, int n, FILE *f).
```

Первый параметр (`buf`) указывает на массив символов, в котором хранится прочитанная строка, во втором параметре (n) хранится максимальное число прочитанных символов, в третьем (`f`) — файловый указатель. Функция `fgets()` использует внешнюю память, на которую указывает `buf`. Эта память, задаваемая массивом или выделяемая функцией `malloc()`, не безгранична. Поэтому в функции `fgets()` и предусмотрен второй параметр `n`, ограничивающий длину читаемой строки. Функция прочитает `n-1` символов, поставит в конце нулевой символ `'\0'` и перейдет к следующей строке. Значит, некоторые слишком длинные строки могут быть обрезаны[1]. После успешного чтения каждой строки `fgets()` возвращает указатель на буфер, где эта строка хранится (в нашем случае это `buf`), если же достигнут конец файла или возникла ошибка при чтении, `fgets()` возвращает `NULL`.

Функция `fputs()` устроена проще:

---

[1] Естественно, символ `'\0'` дописывается в конец каждой строки, независимо от того, прочиталась ли она полностью или оказалась обрезанной.

`fputs(char *buf, FILE *f).`

В ней всего два параметра. Функция `fputs()` записывает строку из буфера `buf` в файл `f`, причем перед записью символ `'\0'`, прибавленный `fgets()`, удаляется.

Теперь нам почти хватает знаний, чтобы понять программу, показанную в листинге 6.1[1]

Листинг 6.1

```
#include <stdio.h>
#define BSIZE 200
int main(){
char buf[BSIZE];
FILE *in, *out;
in=fopen("l21.c","rb");
out=fopen("z.c","wb");
buf[0]='\t';
while(fgets(buf+1,BSIZE-1,in) != NULL)
    fputs(buf,out);
fclose(in);
fclose(out);
return 0;
}
```

Строка `#define BSIZE 200` определяет *символическую константу* `BSIZE`. Значок `#` роднит эту строку с первой, `#include...`, уже хорошо нам знакомой. Обе строки обрабатываются еще до компиляции так называемым

[1] Для лучшего понимания сути программы, не делается проверка in == NULL. В реальной программе такая проверка нужна

препроцессором. Встретив строку `#define` препроцессор заменяет всюду в тексте программы «BSIZE» на «200». Значит, компилятор увидит строку `char buf[200]`, а не `char buf[BSIZE]`. В языке Си часто используются символические константы, так как это позволяет легко менять параметры, от которых зависит программа.

Строка `FILE *in, *out` объявляет два файловых указателя: `in` и `out`. Первый связывается с исходным файлом, а второй — с файлом, куда записывается результат.

Строка `in=fopen("l21.c","rb")` открывает файл под именем l21.c (файл l21.c нужно предварительно создать в каком-нибудь редакторе, например Блокнот. Разумно поместить в этот файл исходный текст программы из листинга 2.1) и связывает с ним указатель `in`. Вторая строка создает новый файл z.c. Если файл z.c уже существует, его содержимое теряется, и после выполнения строки `out=fopen("z.c","wb")` на его месте создается файл с тем же именем, но совершенно пустой.

Перед чтением строки из файла в нулевой элемент массива `buf` засылается символ табуляции `buf[0]='\t'`. Далее в цикле читается строка файла `fgets(buf+1,BSIZE-1,in)`. Обратите внимание, функции `fgets()` передается `buf+1` — указатель на *первый* элемент буфера, потому что в нулевом уже находится символ табуляции. Из-за того, что один элемент буфера всегда занят, функции передается число, на единицу меньшее размера буфера.

Ну вот, наконец-то мы добрались до главного цикла `while()`, в котором читается очередная строка, проверяется значение, возвращаемое функцией `fgets()`, и если оно не равно `NULL`, конец файла еще не достигнут, и функция `fputs()` переписывает строку вместе с символом табуляции в файл z.c.

Казалось бы, все сделано. Но после того, как все строки обработаны, работа еще не закончена. Файлы остаются соединенными с программой и могут быть случайно изменены. Перед завершением программы необходимо отсоединиться от файлов или, как говорят, *закрыть* их. Делается это функцией

`fclose()`, которой надо передать указательна закрываемый файл.

Попробуйте взять файл l21.c с исходным текстом, запустить программу, текст которой показан в листинге 6.1, и посмотрите, что окажется в файле z.c. Помните, что программа и файл l21.c должны находиться в одной папке.

Массивы указателей

Программа, с которой мы познакомились в предыдущем разделе, хотя и выполняет стоящую перед ней задачу, но очень неудобна в работе, потому что имя каждого нового файла приходится включать в исходный текст и снова его компилировать. А хочется, чтобы программа, увидев в командной строке что-то вроде

Программа.exe имя_файла.c имя_результата.c

сама открыла файл имя_файла.c, читала бы каждую строчку, предваряла символом табуляции и записывала результат в файл имя_результата.c.

Такая возможность конечно же есть, но чтобы ей воспользоваться, придется познакомиться поближе с функцией `main()`, которая умеет читать командную строку, то есть параметры, введенные при вызове программы.

Пусть наша программа сдвига текста вправо называется rshift.exe и вызывается как

rshift.exe имя_файла.c имя_результата.c .

Тогда слова имя_файла.c, имя_результата.c, разделенные пробелами, называются *аргументами* командной строки. В нашем случае у программы два аргумента — два файловых имени. Но просто запустить программу с именами файлов недостаточно. Нужно прочитать имена и правильно ими воспользоваться.

Вот для чтения этих аргументов как раз и создана функция `main()`. Ее прототип выглядит так:

```
int main(int argc, char *argv[]).
```

Параметр `argc` на единицу больше числа аргументов, переданных программе. Это «на единицу больше» возникает из-за того, что один аргумент (имя программы) передается ей всегда. Значит, `argc` не может быть меньше единицы.

Второй параметр функции `main` — `char *argv[]` — требует более подробных объяснений, потому что мы с ним до сих пор не встречались. Нам уже хорошо знакомы массивы символов, которые объявляются как `char argv[]`, нам также понятны указатели на `char`, которые объявляются как `char *argv`. Но что такое `char *argv[]`? Легко догадаться, что это *массив, элементами которого служат указатели*, то есть *массив указателей*. Запуская программу с двумя аргументами в командной строке

rshift.exe имя_файла.c имя_результата.c,

мы получаем значение `argc`, равное трем и массив трех указателей `argv[0]`, `argv[1]`, `argv[2]`, первый из которых направлен на имя программы, а два других — на параметры командной строки (см. рис. 6.1)

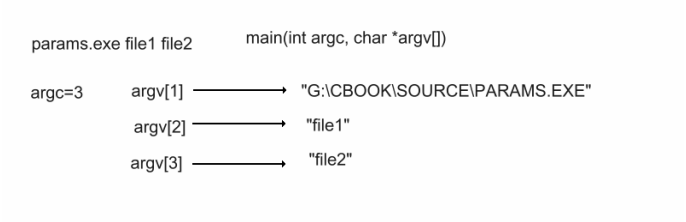

Рис. 6.1 Так хранятся аргументы командной строки

Чтобы лучше представлять себе, как обрабатываются аргументы командной строки, напишем простенькую программу (см. листинг 6.2), которая выводит на экран все аргументы командной строки, число которых равно `argc`.

Листинг 6.2

```
#include <stdio.h>
int main(int argc, char *argv[]){
```

```
int i;
for(i=0;i<argc;i++)
    printf("%s\n",argv[i]);
return 0;
}
```

Если, например, программа запускалась с двумя параметрами

имя_программы.exe file1 file2 ,

то `argc` равен 3 и функции `main()` станут доступны три указателя на строки: `argv[0]`, `argv[1]` и `argv[2]`. Первый `argv[0]` укажет на полный путь к программе[1], второй `argv[1]` — на первый аргумент, то есть file1, третий `argv[2]` — на file2.

Если программа находится в директории G:\CBOOK\SOURCE\, то после запуска с параметрами file1 и file2

имя_программы.exe file1 file2

она выведет на экран три строки — полный путь к программе и два параметра:

G:\CBOOK\SOURCE\имя_программы.EXE

file1

file2

Теперь можно вернуться к задаче, с которой начиналась эта глава, — сдвигу исходного текста программы вправо. Ясно, что имена файлов (исходного и получившегося в результате сдвига) можно передать в командной строке программы, а их чтение может выглядеть так, как показано в листинге 6.3.

---

[1] Некоторые компиляторы покажут только имя программы, а некоторые — не покажут ничего

Листинг 6.3.

```
#include <stdio.h>
#define BSIZE 200
int main(int argc, char *argv[]){
char buf[BSIZE];
FILE *in, *out;
if (argc <3) {
    printf("Слишком мало параметров\n");
    return 1;
}
in=fopen(argv[1],"rb");
if(in==NULL){
    printf("Не открывается файл %s\n",
    argv[1]);
    return 1;
}
out=fopen(argv[2],"wb");
buf[0]='\t';
while(fgets(buf+1,BSIZE-1,in) != NULL)
       fputs(buf,out);
fclose(in);
fclose(out);
return 0;
}
```

В этой программе приняты определенные меры предосторожности. Во-первых, проверяется число параметров. Если `argc` меньше трех, программа не сможет работать, ведь ей нужны имена *двух* файлов. Поэтому при `argc < 3`

выводится сообщение «Слишком мало параметров», и работа завершается.

Во-вторых, проверяется, удачно ли отрыт файл, на имя которого указывает `argv[1]`. Если написать в командной строке имя несуществующего файла, то функция `fopen(argv[1],"rb")` возвратит `NULL`, потому что нельзя прочитать файл, которого нет. В этом случае программа также не будет работать.

Что касается второго файла, то программа будет работать в любом случае: если файла с указанным именем не существует, она создаст его. Если же файл уже есть, программа его уничтожит и на его месте создаст новый.

Чтобы не потерять нужный файл, необходимо каждый раз проверять, существует ли файл с таким именем, и если да — спрашивать, можно ли его уничтожить. С такой задачей справится следующий фрагмент программы:

```
int notexist, ans;
notexist=0;
if((out=fopen(argv[2],"rb")) == NULL){
     notexist=1;
     fclose(out);
}
if(notexist){
     out=fopen(argv[2],"wb");
}
else{
     printf("Удалить %s Y/N?\n",argv[2]);
     ans=getchar();
     if(ans == 'Y' || ans == 'y')
        out=fopen(argv[2],"wb");
     else return 1;
```

```
}
```

Чтобы выяснить, существует ли файл, мы сначала пытаемся открыть его для чтения. Если это удается, то `(fopen(argv[2], "rb")` не возвращает NULL. Значит, файл, на имя которого указывает `argv[2]`, существует.

Состояние файла запоминается в переменной `notexist` и файл закрывается (`fclose(out)`). Если файла с таким именем нет, `notexist` принимает значение 1, выполняется условие `if(notexist)`, и просто создается файл с указанным именем (параметр `"wb"`).

Если файл существует, `notexist=0`, и выполняется ветка `else{}` условной инструкции, в которой спрашивается, можно ли удалить файл. В ответ нужно ввести символ, который (после нажатия Enter) запишется в переменную `ans`. Если `ans` равна `'Y'` или `'y'` (от английского yes — да), файл будет уничтожен. Если же файл изменить нельзя, то программа завершит работу (`return 1`).

В рассмотренном отрывке программы нам встретилась новая функция `getchar()`, возвращающая значение символа, введенного с клавиатуры (или, как говорят, стандартного устройства ввода). У функции `getchar()` нет параметров, она лишь возвращает значение `int`. Казалось бы, функция, читающая с клавиатуры символ, должна возвращать значение `char`, но тогда непонятно, как закончить ввод. Либо он будет бесконечным, либо один из символов придется считать признаком конца ввода. Но в значении типа `int`, возвращаемом `getchar()`, уместится число, *отличное* от любого символа. В языке Си признак конца ввода называется `EOF`, и с помощью `getchar()` можно вводить с клавиатуры длинные последовательности символов и показывать их на экране, как в программе, показанной в листинге 6.4

Листинг 6.4

```
#include <stdio.h>

int main() {
```

```
int c;
while ((c=getchar()) != EOF) {
    putchar(c);
}
return 0;
}
```

Символ вводится с клавиатуры, и после нажатия клавиши Enter его значение присваивается переменной c. Далее переменная сравнивается с `EOF` (символ `EOF` получается, если нажать клавишу Ctrl *и не отпуская* ее, нажать Z), и если равенства нет, символ выводится на экран функцией `putchar(c)`. Если же значение, возвращенное `getchar()`, равно `EOF`, то программа заканчивает работу.

Присваивание `c=getchar()` помещено в скобки, потому что оператор `!=` имеет больший приоритет, чем присваивание `=`. В отсутствии скобок сначала вычислялось бы выражение `getchar() != EOF`, которое ложно (равно нулю) лишь когда `getchar()` возвращает `EOF`. Во всех остальных случаях это выражение истинно, следовательно `c` будет равно единице, и программа в ответ на любой символ будет печатать две смешные рожицы (убедитесь в этом сами).

Задача 6.1 Напишите программу, которая сдвигала бы текст влево, то есть удаляла бы все символы табуляции, вставленные программой, которую мы разобрали в этом разделе

Указатели на указатели

Программа, выводящая на экран аргументы командной строки (см. листинг 6.2) может быть написана немного по-другому:

Листинг 6.5

```
#include <stdio.h>
int main(int argc, char *argv[]){
char *p;
```

```
while((p=*argv++) != NULL))
    printf("%s\n",p);
return 0;
}
```

Хотя, на первый взгляд, она стала просто неузнаваемой и напоминает смесь букв, костей, звездочек и спичек.

Но давайте разберемся. Если объявление `char *argv[]` задает массив *указателей* на `char`, то `argv` – *указатель на нулевой элемент* этого массива. Получается, что `argv` — *указатель на указатель*. Применяя к `argv` раскрытие ссылки, получим `*argv`, что равно нулевому элементу массива, то есть `argv[0]`. Первым элементом массива, записанным с помощью оператора раскрытия ссылки, будет `*(argv+1)`. И так далее вплоть до `argv[argc]`, который, согласно стандарту языка, равен нулевому указателю, т.е. `NULL`.

В цикле `while((p=*argv++)...)`(см. листинг 6.5) указателю `p` присваивается значение `*argv`, а затем `argv` увеличивается на единицу[1]. Значит, в цикле `while()` перебираются все указатели массива, вплоть до `argv[argc]`. В этом случае `p` станет равным `NULL`, и цикл завершится. Заметим, что проверка `(p=*argv++) != NULL)` на самом деле не нужна, потому что действительное значение `NULL` — обычный ноль, то есть число, все биты которого нулевые. Обозначение `NULL` вводится для того, чтобы явно показать, что в работе участвуют указатели, а не числа. Поэтому условие в цикле может быть записано просто как `while(p=*argv++)`.

Раз уж в этом разделе зашла речь о массивах указателей и указателе на указатель, вспомним, что до сих пор параметры функции `main()` записывались как `main(int argc, char`

---

[1] Совмещение раскрытия ссылки и увеличения указателя уже встречалось нам в разделе «Указатели и массивы» главы 5

`*argv[])`. Но теперь мы уже знаем, что вместо массива функции передается указатель на его нулевой элемент. Значит, в списке параметров вместо `char a[]` (так обозначается массив) логично записать `char *a` (указатель). Первый вариант более понятен тем, кто программировал на Бейсике или Фортране, а второй (`char *`) лучше соответствует духу языка Си, потому что функция действительно получает указатель, а не сам массив.

Если вместо массива передается указатель, то что передается вместо массива указателей? Очевидно, указатель на указатель (`argv`, как выяснилось, и есть указатель на указатель). Значит, список параметров функции `main()` можно записать и так:

```
main(argc, char **argv).
```

То есть, указатель на указатель объявляется в языке Си с помощью двух звездочек. Объявление `int **a;` говорит компилятору, что `**a` — это что-то типа `int`.

Чтобы лучше понять двойное раскрытие ссылки (`**`), вернемся еще раз к массиву указателей `*argv[]`. Имя `argv` указывает, как и в любом массиве, на его нулевой элемент. Значит, `*argv` — *и есть* нулевой элемент массива, то есть `argv[0]`. Но `argv[0]` — это *указатель на первый параметр командной строки*, и к нему тоже можно применить оператор раскрытия ссылки `*(argv[0])`. В результате получится, что `*(argv[0])` равно `*(*argv)` или `**argv`.

Задача 6.2 Если параметры командной строки выглядят так же, как на рис. 6.1, то чему равен `**argv`, `**(argv+1)`, `**(argv+2)`?

Как обычно, указатель нельзя путать с массивом. Простое объявление `int **a` выделяет память *только для одного указателя*, а массив `*argv[]`, с которым мы недавно познакомились, содержит несколько указателей на строки, да и сами строки (аргументы командной строки), тоже можно считать частью этой структуры данных.

Файлы — не массивы!

Программа, созданная в разделе «Массивы указателей» прочитала имена двух файлов из командной строки, сдвинула исходный файл вправо и записала результат в выходной файл. Получилось это не очень красиво: пришлось заботиться о том, чтобы не уничтожить нужный файл, и если имя выходного файла было выбрано по ошибке, программа прекращала работу.

Возникает мысль: а нельзя ли записать результат работы в тот же файл, не прибегая к созданию дополнительного выходного файла? Но тогда последовательность действий меняется: нужно создать временный файл, записать туда сдвинутый текст и затем скопировать временный файл в исходный.

Но — спросите вы — какая польза от временного файла, ведь его имя тоже может совпасть с именем уже существующего? К счастью, в языке Си есть стандартная функция `tmpfile()`, которая создает никому не мешающий временный файл с уникальным именем.

Теперь посмотрим на реализацию этой идеи (листинг 6.6)

Листинг 6.6

```
#include <stdio.h>
#define BSIZE 200
int main(int argc, char *argv[]){
char buf[BSIZE];
int ch;
FILE *in, *tmp;
if (argc < 2) {
    printf("Слишком мало параметров\n");
    return 1;
}
in=fopen(argv[1],"rb");
```

```
if(in==NULL){
    printf("Не открывается файл %s\n",
    argv[1]);
    return 1;
}
tmp = tmpfile();
buf[0]='\t';
while(fgets(buf+1,BSIZE-1,in) != NULL)
    fputs(buf,tmp);
fclose(in);
in = fopen(argv[1],"wb");
fseek(tmp,0l,SEEK_SET);
while ((ch=fgetc(tmp)) != EOF){
    fputc(ch,in);
}
fclose(in);
fclose(tmp);
return 0;
}
```

Начало программы, показанной в листинге 6.6, хорошо нам знакомо. Единственное отличие от предыдущей версии в том, что теперь проверяется условие `argc < 2`, потому что в командной строке нужно ввести имя только одного, исходного файла.

После его открытия `in=fopen(argv[1],"rb")` создается временный файл `tmp = tmpfile()`. Далее идет знакомый фрагмент программы, где читается строчка в исходном файле, и переписывается (уже с символом табуляции) во временный файл.

Далее исходный файл закрывается (`fclose(in)`) и создается вновь: `in = fopen(argv[1],"wb")`. Теперь это пустой файл, с тем же, правда, именем, где должны оказаться результаты работы.

Следующая строка

```
fseek(tmp,0l,SEEK_SET);
```

до сих пор нам не встречалась, но она крайне важна, потому что проливает свет на природу файлов. Дело в том, что *файлы — не массивы*. Массив — типичная структура данных *с произвольным доступом*. Это значит, что доступен каждый его элемент. Мы можем написать `a=dig[0]` или `a=dig[10]`. Файлы — совсем другое дело. Чтобы прочитать десятую строку текста, хранимого в файле, необходимо перед этим прочитать предыдущие девять строк. Файл похож на кассету: чтобы услышать десятую песню, необходимо прослушать девять предыдущих или…перемотать ленту.

Вот таким перемотчиком файлов и служит функция `fseek()`. Можно представить себе, что с каждым открытым файлом связан специальный флажок, которым помечается место, доступное в данный момент для чтения/записи. При открытии файла флажок расположен в самом его начале. Прочитали одну строку — и флажок переместился к началу следующей строки. Когда прочитаны все строки, флажок оказывается в конце файла и при попытке чтения функция `fgets()` выдает `NULL`.

То же самое происходит и при записи строк в файл: новая строка пишется туда, где в данный момент находится флажок. Записали строчку, и флажок переместился к ее концу.

Значит, после записи сдвинутого текста флажок переместится *в конец* файла tmp. Если теперь мы вздумаем читать временный файл, то флажок, установленный в его конце, переместится дальше, где просто ничего нет!

Значит, перед тем как копировать временный файл в исходный, *необходимо переместить флажок в начало*, именно это и делает функция `fseek(tmp,0L,SEEK_SET)`. Ее первый аргумент `tmp` — указатель файла, `0L` — константа

(длинное целое), равная смещению флажка, а SEEK_SET — константа, показывающая, что смещение отсчитывается *от начала* файла[1]. Аргументы функции fseek() таковы, что временный файл «будет перемотан», и флажок окажется в самом его начале.

Теперь можно читать файл tmp и переписывать его содержимое в заново созданный входной файл. Программа, показанная в листинге 6.5, использует для копирования файлов другие функции — fgetc() и fputc(), очень похожие на getchar() и putchar() (см. листинг 6.4). И та и другая пара функций читает/пишет по символу при каждом обращении к ним. Разница в том, что fgetc()/fputc() читают/пишут из файла в файл, а не с клавиатуры на экран. Можно применить и копирование строк fgets()/fputs().

Открытие файла

В предыдущем разделе исходный файл сначала был прочитан, потом закрыт, затем открыт вновь, чтобы записать туда результаты работы. Это делалось потому, что исходный файл первоначально был открыт *только для чтения* (параметр "rb"), а нам необходима была *запись*.

Возникает вопрос: нельзя ли как-нибудь так открыть файл, чтобы его можно было читать, а затем записать туда сдвинутый текст. Чтобы понять, возможно ли это, изучим разнообразные режимы открытия файлов в языке Си, показанные в таблице 6.1

К каждому режиму, показанному в табл. 6.1, можно дописать букву b, которая велит воспринимать файл таким, каков он есть. Без буквы b файлы считаются текстовыми, а это значит, что при чтении и записи будут меняться служебные символы в

---

[1] Есть еще две константы: SEEK_CUR задает смещение флажка с текущей его позиции, а SEEK_END — с конца файла. Само смещение может быть и отрицательным, например fseek(tmp,-3L, SEEK_CUR) — переместить флажок назад на три байта от текущей позиции.

конце строки (обычно это символ возврата каретки, обозначаемый CR, и символ перевода строки, обозначаемый LF).

Таблица 6.1. Различные режимы открытия файлов

Файл	r	w	a	r+	w+	a+
Должен существовать	x			x		
Уничтожается		x			x	
Можно читать	x			x	x	x
Можно писать		x	x	x	x	x
Можно дописывать в конец			x			x

Нас пока интересует открытие файла в бинарном режиме (с буковкой «b»), причем необходимо, чтобы можно было открыть существующий файл и для чтения, и для записи (чтобы поместить результат работы в тот же файл.). Как видно из табл. 6.1, всем этим условиям отвечает режим `"r+"`. Добавив еще букву `b`, получим режим открытия файла: `in=fopen(argv[1],"r+b")`.

Теперь не нужно закрывать входной файл, нужно только переместить флажки к началу входного и временного файла, а затем читать символы из файла tmp и записывать их «поверх» уже существующего файла. Программа, которая все это делает, показана в листинге 6.7

Листинг 6.7

```
#include <stdio.h>
#define BSIZE 200
int main(int argc, char *argv[]){
char buf[BSIZE];
int act_read;
```

```
FILE *in, *tmp;
if (argc <2) {
    printf("Слишком мало параметров\n");
    return 1;
}
in=fopen(argv[1],"r+b");
if(in==NULL){
    printf("Не открывается файл %s\n",
    argv[1]);
    return 1;
}
tmp = tmpfile();
buf[0]='\t';
while(fgets(buf+1,BSIZE-1,in) != NULL)
    fputs(buf,tmp);
rewind(in);
rewind(tmp);
do{
    act_read=fread(buf,1,BSIZE,tmp);
    fwrite(buf,1,act_read,in);
} while (act_read == BSIZE);
fclose(in);
fclose(tmp);
return 0;
}
```

В ней есть несколько новшеств. Во-первых, входной файл открывается для чтения и записи `in=fopen(argv[1], "r+b")`. Во вторых, для «перемотки» файла используется функция `rewind(in)` которая смотрится гораздо проще, а делает то же самое, что и `fseek(tmp,0l,SEEK_SET)` — перемещает флажок в начало файла.

Далее, в программе используется другой тип цикла: `do {} while(условие)`. Раньше нам был известен цикл `while(){}`, который выполнял инструкции, заключенные в фигурные скобки, до тех пор пока выполнялось `условие`. Иными словами, проверка условия проводилась *до* входа в цикл, и могло случиться так, что цикл не выполнится ни разу.

С циклом `do {} while(условие)` такого произойти не может, потому что условие проверяется *после* выполнения инструкций в фигурных скобках. Значит, цикл этого типа выполнится, по крайней мере, один раз.

В нашем случае цикл

```
do{
     act_read=fread(buf,1,BSIZE,tmp);
     fwrite(buf,1,act_read,in);
} while (act_read == BSIZE);
```

копирует временный файл в исходный.

Функция `fread()`, с которой мы тоже встречаемся впервые, читает `BSIZE` байт из временного файла и записывает их в буферный массив `buf`. Точнее, `BZIZE` байт она прочтет в том случае, если они будут в файле. Последняя порция данных (в конце файла) может быть меньшего размера. Поэтому `fread()` возвращает число реально прочитанных байт, которое запоминается в переменной `act_read`. Естественно, функция `fwrite(buf,1,act_read,in)` пишет из буфера `buf` во входной файл `in` ровно столько байт, сколько сумела прочитать `fread()`. Ясно, что обе функции будут все время читать/писать по `BSIZE` байт, пока не будет достигнут самый конец файла, вот здесь `fread()`в первый и последний раз

может прочитать меньшее число байт (оно может быть равно и нулю, если размер файла кратен BSIZE), и это будет служить признаком окончания работы.

Представим себе, что BSIZE равно двумстам, а в файле ровно 450 байт. При входе в цикл функция `fread(buf,1,BSIZE, tmp)` прочитает 200 байт, которые окажутся в массиве `buf`, а переменная `act_read` станет равной двумстам. Далее функция `fwrite(buf,1,act_read,in)` возьмет `act_read` байт из массива `buf` и запишет их во входной файл. Теперь мы подошли к проверке условия `act_read == BSIZE`. Оно (так как `act_read` равно `BSIZE)` выполняется, и мы переходим к следующему обороту цикла, который пройдет также, как и первый — 200 байт будут прочитаны и записаны.

Но к третьему обороту останется всего 50 непрочитанных байт, и значение, возвращенное `fread()` и присвоенное переменной `act_read`, будет равно 50. Точно такое же число байт будет записано в файл, и когда в третий раз настанет очередь проверки `act_read == BSIZE`, условие не выполнится, и цикл прекратит работу.

В заключение скажем, что функция `fread()` (и, соответственно, `fwrite()`) может задавать чтение (запись) не только двухсот байт, а, скажем, 200 пар двухбайтовых чисел. Для этого второй аргумент функции должен быть равен двойке:

```
fread(buf,2,BSIZE,tmp);
```

Это удобно, когда из файла читаются не символы, а, например, целые числа. Чтение 200 переменных типа `int` можно задать так:

```
fread(buf,sizeof(int),200,tmp);
```

Эта запись выглядит более понятно, и кроме того, она еще и более универсальна, так как размер целочисленной переменной может меняться в зависимости от компьютера или компилятора.

Глава 7. Строки

Считалочка

И тут в мой разум грянул блеск с высот, неся свершенье всех его усилий

Данте «Божественная комедия. Рай»

Строки, то есть последовательности букв (символов) очень часто встречаются в задачах, решаемых компьютером, ведь тексты книг, статей, да и сами программы состоят из отдельных строк. Поэтому многое в языке Си создано для работы с ними. В этой главе мы попробуем работать со строками и попутно изучим некоторые новые для нас конструкции языка.

Начнем с задачи сортировки последовательности слов, из которых состоит детская считалочка: *на златом крыльце сидели: царь, царевич, король, королевич, сапожник, портной. Кто ты будешь такой*? Эта задача, несмотря на шутливую формулировку, крайне важна, например, при создании словарей, где отдельные статьи должны быть расставлены в алфавитном порядке.

Прежде чем приступать к ее решению, договоримся, что сортируемые слова хранятся в файле — по одному на каждой строчке:

```
На
златом
крыльце
…
кто
ты
будешь
```

```
Такой
```

Хранить строки в файле очень удобно: файл легко редактируется и вмещает сколько угодно слов, но чтобы отладить программу, достаточно и тех четырнадцати, из которых состоит считалочка.

Было бы неразумно сразу писать программу сортировки целиком, как это часто делают неопытные программисты. Набросав программу из 100-200 строк, они начинают в ней путаться и не могут справиться со множеством ошибок в разных ее частях. Создание программы можно сравнить с прокладкой шахты: после каждых нескольких метров работа прекращается, и ставятся подпорки. И только надежно обеспечив тыл, шахтеры продолжают движение дальше.

Примерно так же поступим и мы, создав сначала программу чтения и хранения слов из файла. Но сначала нужно решить стратегическую задачу: как будут храниться сортируемые слова? Целые числа, которые мы сортировали в разделе «Массивы» главы 2, можно было менять местами, но со словами так не получится, так как они отличаются друг от друга размером: «царевич» не уместится там, где стоял «царь».

Одно из возможных решений — хранение слов в массивах одинакового размера — некрасиво и неэффективно. Во-первых, неизвестен максимальный размер слова, и придется сначала прочитать все слова, определить максимальную длину, а уж потом выделять память и читать их еще раз. Во-вторых, при размещении слов в массивах одинакового размера приходится расходовать лишнюю память. В третьих, переписывание с места на место длинных слов сильно загружает процессор и программа (в случае сортировки большого числа слов) будет работать медленно. Поэтому хочется придумать что-то более изящное.

Задача 7.1 *Подумайте, как организовать хранение слов, чтобы к ним был легкий доступ и чтобы удобно было менять их местами*

И тут (смотрите эпиграф) вспоминается, что размер указателя *постоянен* и не зависит от длины строки, на который он указывает. Это значит, что *можно менять местами*

указатели, а соответствующие слова пусть стоят на месте! То есть, сортировать не сами строки, а лишь *указатели* на них. А потом просто брать по очереди указатели и печатать соответствующие слова.

Теперь план размещения данных в памяти компьютера становится очевиден: заводим массив указателей на `char` (их число равно числу слов). Каждый указатель направлен на отдельное слово (строку). Первый — на слово «на», второй — на слово «златом», … последний — на слово «такой» (рис. 7.1)

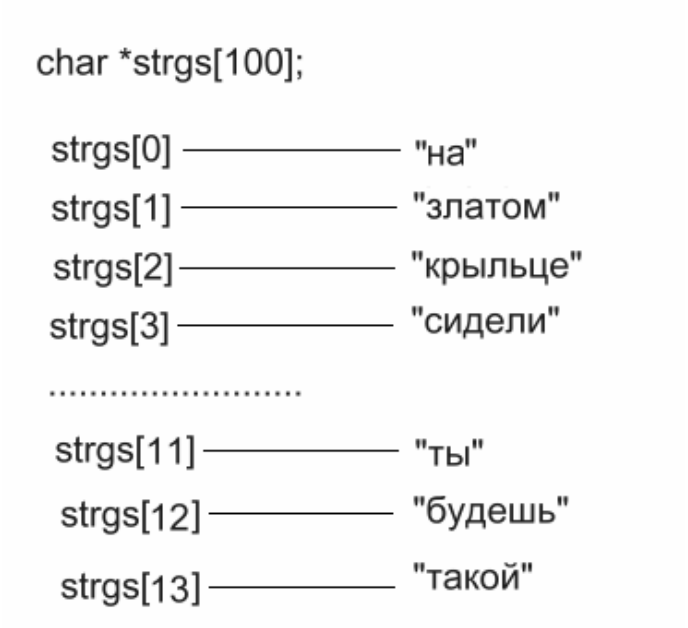

Рис. 7.1. Правильное хранение строк

Программа, читающая слова считалочки и затем печатающая их на экране, показана в листинге 7.1.

Листинг 7.1.

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

```
#include <stdlib.h>
#include <string.h>
#define BSIZE 200
int main(int argc, char *argv[]){
char buf[BSIZE];
char *strgs[100];
int i, nofl;
FILE *in;
if (argc <2) {
    printf("Слишком мало параметров\n");
    return 1;
}
in=fopen(argv[1],"r+b");
if(in==NULL){
    printf("Не открывается файл %s\n",
    argv[1]);
    return 1;
}
i=0;
while (fgets(buf,BSIZE-1,in) != NULL){
    strgs[i]=malloc(strlen(buf)+1);
    strcpy(strgs[i],buf);
    i++;
}
nofl=i;
/*
сюда вставим саму сортировку
```

```
*/
for(i=0;i < nofl;i++)
    printf("%s", strgs[i]);
return 0;
}
```

Как и в прежних наших программах, здесь выделяется буфер размером `BSIZE` для чтения одной строки из файла. Кроме того, объявляется массив указателей на `char` (см. раздел «Массивы указателей» главы 6) `char *strgs[100]` — всего 100 указателей, но для считалочки достаточно первых четырнадцати. Далее в программе открывается файл, в котором записаны слова, с этим файлом связан указатель `in`. Самая важная часть программы занимает всего несколько строк:

```
i=0;
while (fgets(buf,BSIZE-1,in) != NULL){
    strgs[i]=malloc(strlen(buf)+1);
    strcpy(strgs[i],buf);
    i++;
}
nofl=i;
```

Перед входом в цикл `while()` счетчик прочитанных слов равен нулю (`i=0`). Функция `fgets()`, с которой мы познакомились в разделе «Падение железного занавеса» главы 6, читает очередное слово из файла, записывает его в буфер `buf` и дописывает в конце `'\0'`. Если значение, возвращенное `fgets()`, не равно `NULL`, выполняются инструкции, помещенные внутрь цикла. Первая,

```
strgs[i]=malloc(strlen(buf)+1);
```

выделяет память под очередное слово, указатель на которое хранится в элементе массива `strgs[i]`. Число выделяемых байт равно `strlen(buf)+1` , потому что `strlen(buf)`

вычисляет длину строки, хранящейся в буфере, без учета завершающего нуля `'\0'`. После выделения памяти функция `strcpy(strgs[i],buf)` переписывает строку из массива `buf` в то место памяти, на которое указывает `strgs[i]`. Завершает оборот цикла увеличение счетчика прочитанных слов `i++`.

Наконец, последний цикл в программе

```
for(i=0;i < nofl;i++)
    printf("%s", strgs[i]);
```

отображает на экране все введенные слова. Здесь `strgs[i]` — указатель на очередное слово, а `"%s"` — задает вывод строки символов.

По сути, мы построили некую рамку, в которую должна быть вставлена сама сортировка (это место показано в листинге 7.1). Пока же программа просто читает, считает и выводит на экран записанные в файле слова.

В заключение этого раздела стоит обратить внимание на три заголовочных файла в начале программы. В первом находится описание `printf()`, во втором — `malloc()`, а в третьем — — описание функций `strlen()` и `strcpy()`.

Сортировка строк

Научившись вводить и правильно хранить слова, можно перейти к следующей задаче, без которой немыслима сортировка: сравнению строк. Эта задача не так проста, как сравнение целых чисел, ведь даже малые дети знают, что две шоколадки больше, чем одна, но те же дети немало удивятся, узнав, что `"лампочка"` больше чем `"апельсин "`.

Строки можно сравнивать по-разному, например, считать *большей* более длинную строку. Но для задачи расстановки в алфавитном порядке необходимо другое определение: *большей* из двух строк будем считать строку, у которой первая отличающаяся буква (если смотреть слева) стоит *позже* в алфавитном списке: "лампочка" *больше* чем "апельсин",

потому что первая отличающаяся буква «л» стоит в алфавите *позже*, чем «а».

Именно такому определению соответствует библиотечная функция `strcmp(char *str1, char *str2)`, принимающая указатели на начала строк и возвращающая 0, если строки равны, положительную величину, если `str1` больше `str2`[1] и отрицательную — если `str1` меньше `str2`.

Теперь все готово к тому, чтобы полностью написать программу сортировки слов. В сущности, программа будет почти такой же, как и для сортировки целых чисел (см. листинг 2.6), но вместо сравнения целых чисел `if(dig[j] > dig[mm]` нужно будет сравнивать строки `if(strcmp(strgs[j],strgs[mm])>0)`, да еще поменять типы некоторых переменных. Например, `tmp` должна быть не целой, а указателем на `char`. Фрагмент, который должен быть вставлен между чтением и печатью слов (см. листинг 7.1) будет таким:

```
nofl=i;
char *tmp;
int j,mm;
for(i=nofl-1;i>=1;i--){
    mm=0;
    for(j=1;j<=i;j++)
        if(strcmp(strgs[j],strgs[mm]) > 0)
            mm=j;
    tmp=strgs[i];
    strgs[i]=strgs[mm];
    strgs[mm]=tmp;
```

[1] Когда говорится, что `str1` больше `str2`, имеется в виду, что строка, на которую указывает `str1`, больше строки, на которую указывает `str2`.

```
    }
```

Здесь почти все повторяет сортировку целых чисел (см. листинг 2.6). Сначала находится самая большая строка, указатель на которую меняется местом с указателем, хранящимся в конце массива. Далее ищется наибольшая строка из оставшихся, и указатель на нее меняется с указателем, стоящем в массиве на предпоследнем месте. Так продолжается до тех пор, пока все указатели не будут расставлены в порядке возрастания соответствующих строк. После сортировки все слова выводятся на экран и получается нечто любопытное:

```
будешь
златом
королевич
король
крыльце
кто
Ц
```

(конец списка посмотрите сами).

Обратите внимание: переменные `tmp`, `j`, `mm` объявляются непосредственно перед сортировкой. В языке Си объявить переменные можно в любом месте программы, но использовать их можно только *после* объявления.

Указатель на функцию

Программа сортировки слов, представленная в предыдущем разделе, очень несовершенна. И то, что она работает — еще не повод прекращать работу над ней самой.

Взглянув еще раз на исходный текст (см. листинг 7.1), видим, что программа состоит из трех малозависимых частей: ввода данных, собственно сортировки и вывода на экран отсортированной последовательности слов. Спрашивается, какая часть программы наиболее важна и универсальна?

Конечно же, сама сортировка! Значит, нужно обособить эту часть, написав соответствующую функцию, чтобы можно было использовать ее в других программах.

Опытные программисты, пришли бы к такому выводу сразу. Они наверняка начали бы с упрощенной схемы:

```
ВВОД
СОРТИРОВКА
ВЫВОД
```

Потом они продумали бы, как хранить введенные строки. Тогда стало бы ясно, что для сортировки необходима функция, принимающая массив указателей и число строк. Возможно, разработка программы началась бы именно с функции, выполняющей сортировку.

Так сделали бы опытные программисты, а нам, пока мы только учимся, лучше поступить наоборот: посмотреть на уже работающую программу и понять, какой будет функция сортировки.

Очевидно, она должна принимать массив указателей и число слов. Возвращать же ей нечего, после себя `strsort()` оставляет отсортированный массив указателей. Выглядеть она может так:

```
void strsort(char *s[],int N){
char *tmp;
int i,j,mm;
for(i=N-1;i>=1;i--){
     mm=0;
     for(j=1;j<=i;j++)
         if(strcmp(s[j],s[mm]) > 0)
             mm=j;
     tmp=s[i];
     s[i]=s[mm];
```

```
        s[mm]=tmp;
    }
    }
```

Обратите внимание, теперь некоторые переменные (`mm`, `j`, `tmp`), которые раньше приходилось объявлять в основной программе, перешли внутрь функции, что повышает надежность программы в целом.

Функция, которую мы только что написали, имеет, по крайней мере, один недостаток: она использует единственный способ сравнения строк — `strcmp()`, хотя ясно, что таких способов может быть множество. Можно учитывать при сравнении разницу между большими и малыми буквами, как это делает `strcmp()`, а можно не учитывать. Можно сортировать строки не в алфавитном порядке, а как-то иначе, например, в порядке возрастания длины и т.д.

Чтобы функция сортировки была по-настоящему универсальной, у нее, помимо числа строк и массива указателей на них, должен быть еще один параметр, который позволит задать любую функцию сравнения. Что это за параметр, легко понять, посмотрев, как вызывается функция в программе, прошедшей через компилятор и ставшей последовательностью команд процессора.

Функции для процессора — те же двоичные коды, и с виду они мало отличаются от строк или массивов. Так же, как и массивы, они занимают последовательные ячейки памяти, где хранятся двоичные числа. Разница в том, что эту последовательность ячеек процессор воспринимает не как идущие подряд числа или буквы, а как последовательность *команд*.

Чтобы выполнилась та или иная функция, нужна подготовительная работа: перед ее вызовом в условленное место записываются аргументы функции и номер ячейки, куда вернется процессор после ее выполнения. Далее процессору передают номер ячейки, с которого начинается функция, и он послушно исполняет записанную в ней команду. Команды в первых ячейках функции достают из условленного места

памяти ее аргументы. Инструкции, из которых состоит функция, выполняются одна за другой, до самого конца. Последняя инструкция передает процессору номер ячейки, куда он должен вернуться. Обычно это ячейка, которая находится в памяти сразу за той, где была команда перейти к функции. Немного отвлекшись, процессор продолжает работу как ни в чем не бывало (рис. 7.2).

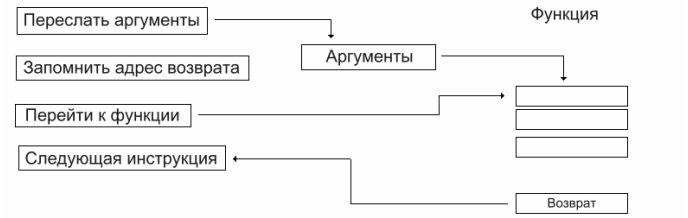

Рис. 7.2. Вызов функции и возврат из нее (очень схематично)

Давайте теперь представим, как могла бы быть устроена функция, которой можно передать другую функцию в качестве параметра. Очевидно, функции можно передать *не саму функцию, а адрес ее начала* вместе со списком и типами параметров. Адрес нужен процессору, чтобы правильно перейти к функции, а список параметров — чтобы правильно передать ей аргументы.

Но что такое адрес начала функции? Очевидно, это *указатель на нее*. И нам осталось только догадаться, как может выглядеть такой указатель. Объявление `int *a` задает, как мы знаем, указатель на `int`. Объявление `int *f()` задает функцию, возвращающую указатель на `int`. Как же задать указатель на функцию? Создатели языка Си пришли к такой записи:

```
int (*f)(); /* указатель на функцию, возвращающую int */
```

Указателю на функцию, как и любому указателю, нужно перед использованием, придать разумное значение. Как это сделать, показывает листинг 7.2

Листинг 7.2.

```
#include <stdio.h>
int sum(int,int);
int main(){
      int i=2,j=3,k;
      int (*f)(int,int);
      printf("%d\n",sum(i,j));
      f=&sum;
      printf("%d\n",(*f)(i,j));
      return 0;
}
int sum(int f,int s){
      return (f+s);
}
```

Здесь `sum()` — функция, возвращающая сумму двух целых чисел, а `f` — *указатель* на функцию с двумя целочисленными параметрами, возвращающую значение `int`.

Объявление указателя на функцию

```
int (*f)(int, int)
```

только выделяет участок памяти для него. Первоначально там содержится «мусор». Разумное значение указателю на функцию, как и любому другому указателю, придает оператор `&`. После присваивания `f=&sum` указатель `f` равен адресу, с которого начинается функция `sum()`. Теперь, чтобы пользоваться им как функцией, к нему нужно применить оператор раскрытия ссылки `*`: `(*f)(i,j)` возвращает точно такое же значение, что и `sum(i,j)`, то есть пятерку.Наконец, после длинного отступления, мы готовы создать функцию сортировки, принимающую функцию сравнения строк (вернее, указатель на нее) в качестве параметра. Выглядеть она может так:

```
void strsort(char *s[],int N,
int (*cmp)(char *,char *)){
char *tmp;
int i,j,mm;
for(i=N-1;i>=1;i--){
     mm=0;
     for(j=1;j<=i;j++)
         if((*cmp)(s[j],s[mm]) > 0)
             mm=j;
         tmp=s[i];
         s[i]=s[mm];
         s[mm]=tmp;
}
}
```

Сравнивая ее с предыдущей версией, видим, что появился третий параметр — указатель на функцию `cmp` с двумя параметрами (оба — указатели на `char`). Функция, на которую указывает `cmp`, возвращает целое значение `int`. Внутри функции сортировки, там, где стояла раньше функция сравнения

```
if(strcmp(s[j],s[mm]) > 0)
```

стоит указатель на функцию, с оператором раскрытия ссылки:

```
if((*cmp)(s[j],s[mm]) > 0)
```

Пользоваться новой функцией довольно просто. Во-первых, в начале программы ее нужно объявить:

```
void strsort(char *[],int,
int (*cmp)(char *,char *));
```

Далее, нужно написать собственную функцию сравнения строк. Если, скажем, хочется расставить строки в порядке возрастания длины, то функция сравнения будет такой:

```
int cmplen(char *s1, char *s2){
    return (strlen(s1) - strlen(s2));
}
```

Она возвращает разницу длин двух строк, которая может быть больше, меньше или равна нулю. Функция сортировки будет в этом случае вызываться так:

```
strsort(strgs,i,&cmplen);
```

Здесь `strgs` — массив указателей на строки, `i` — число строк, а `&cmplen` — адрес нашей функции сравнения.

Задача 7.2 Напишите программу, которая читает строки из файла и выводит их на экран в порядке возрастания длины. Можете использовать разные подходы, в том числе указатели на функции.

Имена функций и указатели

Этот раздел содержит правдивые, но вредные сведения, способные окончательно запутать читателя. В зависимости от упорства и храбрости ему предлагаются три варианта действий: либо вообще его не читать, либо пропустить при первом чтении, либо сразу перейти ко второму чтению, минуя первое.

В предыдущем разделе мы познакомились с указателями на функцию, которые объявляются и используются так же, как и любые другие указатели. Объявляя указатель на `int` — `int *p`, мы выделяем для него память, в которой первоначально содержится «мусор». Перед использованием указатель должен содержать адрес существующей переменной или свободной области памяти. Разумное значение указателю придает оператор `&`: `p=&i` (в `p` засылается адрес переменной `i`). Оператор раскрытия ссылки `*` открывает еще один доступ к переменной. Присвоить ей значение можно двояко: просто

записав `i=число` или воспользовавшись указателем: `*p=число`.

Аналогично, объявление указателя на функцию `int (*f)()` выделяет всего лишь участок памяти, содержащий «мусор». С помощью оператора `&` указателю присваивается разумное значение: `f=&fun`, где `fun` — имя существующей функции. После этого указателем можно пользоваться для обращения к функции, применяя к нему оператор раскрытия ссылки: `fun()` эквивалентно `(*f)()`. Казалось бы, указатели используются одинаково для переменных любого типа. Но давайте еще раз вспомним про массивы. Как уже говорилось в разделе «Массивы и указатели» главы 5, имя массива — это адрес его нулевого элемента, который можно получить обычным способом, применяя оператор `&`: `p=&arr[0]` (указатель `p` равен адресу нулевого элемента массива) или просто `p=arr`. То, что имя массива в Си равно адресу его нулевого элемента, стало возможно потому, что с именем массива не связано какое-то значение, как в обычной целочисленной переменной.

Если `pl` — указатель на `char`, а `l` — переменная типа `char`, то нельзя написать `p=l`, потому что тип того, что стоит слева, не совпадает с типом того, что стоит справа. Но если `lm` — имя *массива* символов, а `p` — указатель на char, наши руки развязаны, и можно написать `p=lm`, потому что `lm` — это *только имя*, с ним непосредственно не связано значение элемента массива, а значит, можно ради удобства считать, что имя массива — это адрес его нулевого элемента.

То, что справедливо для массивов, должно быть справедливо и для функций. Ведь массивы и функции похожи: и в тех и в других имя отделено от содержания, и так же как имя массива эквивалентно адресу нулевого элемента, так и *имя функции равно адресу ее начала*. Это значит, что оператор `&` не обязателен, когда указателю на функцию присваивается значение, и если `f` — указатель на функцию, а `fun` — сама функция, то можно просто писать `f=fun`! Более того, по аналогии с массивами не нужен и оператор раскрытия ссылки. Если `f` — указатель на функцию, то можно воспользоваться им, как простым именем функции: `f()`.

Программу из листинга 7.2 можно переписать так:

```
#include <stdio.h>
int sum(int,int);
     int main(){
     int i=2,j=3,k;
     int (*f)(int,int);
     printf("%d\n",sum(i,j));
     f=sum;
     printf("%d\n",f(i,j));
 return 0;
}
```

Как видим, имя функции и указатель на нее практически равноправны. Разница между ними в том, что имя функции нельзя отделить от нее самой, оно всегда связано с постоянным адресом. Указатель же более свободен, и его можно направить куда угодно.

Qsort

Функции, которые мы использовали для сортировки строк в предыдущих разделах, были написаны для того, чтобы изучить новые способы хранения данных и новые конструкции языка (в частности, указатели на функции).

А для самой сортировки разумнее поискать подходящую функцию в стандартной библиотеке языка Си, ведь тот, кто пишет все функции сам, подобен человеку, который катается на старом велосипеде, в то время как новенький «Мерседес» стоит без дела в его собственном гараже.

Программисты часто изобретают собственные функции, потому что им лень разбираться с чужими. Но в результате затрачивается больше времени, а программа получается хуже. Стандартными функциями стоит еще заниматься потому, что они открывают много нового в самом языке Си. Особенно это

справедливо в отношении функции `qsort()`, беглый взгляд на прототип которой

```
void qsort(void* base, unsigned n, unsigned
size, int (*cmp)(const void*, const void*));
```

показывает, что она принимает какой-то указатель `base`, два значения `unsigned` : `n` и `size` и указатель на функцию `int (*cmp)(...)`. Общий смысл этих параметров понятен: `base` — указатель на начало сортируемого массива, `n` — число сортируемых элементов, `size` — размер элемента, а `int (*cmp)(const void*, const void*)` `y` указатель на функцию сравнения, которая показывает, какой из двух сортируемых элементов больше.

Чтобы продвинуться дальше, необходимо понять, что означает `void * base`. До сих пор слово `void` встречалось нам лишь в прототипах функций `void fun()`, которые ничего не возвращают вызвавшей программе. Там все было понятно, ведь «void» в переводе с английского означает «пустота». Но в применении к указателям это говорит не о пустоте, а об универсальности (а слова `void *base`, стоящие первыми в списке, означают, что `base` — указатель на `void`). Указатель на `void` — это *обобщенный* указатель, о котором нельзя сказать, на что он указывает, до тех пор, пока его не приведут к какому-то конкретному типу. К указателю `base`, объявленному как `void *base` нельзя прибавить число, потому что не известен размер переменной, на которую он ссылается.

Почему же первый параметр в функции `qsort()` объявлен как указатель на `void`? Потому, что это позволяет принять *любой* указатель. Если бы в списке параметров стояло `int *base`, функция `qsort()` могла бы обрабатывать только целочисленные данные. Но если стоит указатель на `void`, функция готова сортировать *любые* переменные (если, конечно указать ей их размер `size`). Естественно, функция должна еще знать число сортируемых элементов `n`.

Нам осталось разобраться с последним параметром функции — указателем на функцию, принимающую два указателя на `const void` и возвращающую значение `int`:

```
int (*cmp)(const void*, const void*).
```

Функция, на которую указывает `cmp`, сравнивает два сортируемых элемента. Ее нужно писать самостоятельно, потому что, как мы видели, сравнивать можно по-разному. В разделе «Указатель на функцию» мы уже знакомились с двумя функциями сравнения: одна нужна для сортировки строк в алфавитном порядке, другая — в порядке возрастания длины. Функция `qsort()` вызывает, когда это нужно, функцию `cmp`[1], передавая ей *указатели* на сравниваемые элементы. Функцию `qsort()` не интересуют внутренности функции сравнения `comp()`. Нужно только, чтобы `comp()` возвращала положительное значение, если первый аргумент больше второго, нуль — если они равны и отрицательное значение, если второй аргумент больше первого.

Указатели на `void` сообщают нам, что функция `cmp()` может сравнивать элементы любого размера, а слово `const` говорит о том, что с помощью передаваемого указателя нельзя изменить *сам* элемент. И это разумно, потому что функция `cmp()` не должна *менять* передаваемые ей элементы массива. Она должна только *читать* их и решать, какой из них больше. Чуть подробнее о слове `const` сказано в разделе «Функции с длинными руками» главы 4.

Ну вот, теперь мы, наконец, может разобрать пример применения функции qsort(). В листинге 7.3 показана программа, сортирующая целые числа, хранящиеся в массиве `x[]`.

Листинг 7.3.

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

---

[1] Вместо «функция, на которую указывает cmp», будем говорить «функция cmp» ведь указатель на функцию и имя функции, как следует из предыдущего раздела, эквивалентны.

```
#include <stdlib.h>
int cmp(const void *,const void *);
int main(){
int i;
int x[10]= {5,3,2,4,6,7,11,17,0,13};
qsort(x,10,sizeof(int),cmp);
for(i=0;i<10;i++)
    printf("%d\n",x[i]);
return 0;
}
int cmp(const void *a, const void *b){
int n1,n2;
n1=*(int *)a;
n2=*(int *)b;
if(n1 < n2)          return -1;
else    if(n1 == n2) return 0;
        else         return 1;
}
```

Большую часть этой программы занимает подготовка массива сортируемых чисел и вывод на экран результата. Все это делается ради строки

```
qsort(x,10,sizeof(int),cmp) ,
```

где вызывается функция сортировки, которой передается имя массива (`x`) количество сортируемых чисел (10), число байт, занимаемых каждым числом (`sizeof(int)`) и указатель на функцию сравнения `cmp()`. Ее прототип `int cmp(const void *,const void *)` всегда одинаков, но внутренности должны отличаться в зависимости от того, что сравнивается. В нашем случае сравниваются целые числа, поэтому указатель на `void` должен быть приведен к указателю на `int` (`int`

`*)`, а чтобы получить само число, нужен оператор раскрытия ссылки: `n1=*(int *)a`. Вычислив оба числа, функция сравнения возвращает –1, если первое число меньше второго, 0, если они равны, и 1, если первое число меньше. Пожалуй, здесь впервые нам встретились вложенные логические инструкции, смысл которых понятен: если условие `(n1 < n2)` не выполняется, то возможны два варианта: если `n1==n2`, возвращается ноль, в противном случае — единица.

Задача 7.3 Функция сравнения

```
int cmp(const void *a, const void *b){
    return *(int *)a - *(int *)b;
}
```

казалось бы, выполняет ту же задачу, что и функция `cmp()` в листинге 7.3. Тем не менее, она ненадежна и опасна. Почему?

Задача 7.4 Напишите программу, которая сортирует целые числа, хранящиеся в файле. Каждое число занимает одну строку и состоит из символов `'0',...'9'`. Перед отрицательными числами стоит знак `'-'`. Подсказка: используйте функцию `atoi()`, которая преобразует строки в числа.

Завершим этот раздел примером сортировки строк с помощью `qsort()`. В первых разделах этой главы мы поняли, что строки могут быть как угодно разбросаны в памяти компьютера, если только указатели на них собрать в одном массиве. Сортировка строк сведется в этом случае к сортировке указателей на них. Значит, вызов функции `qsort()` в данном случае будет выглядеть так:

```
qsort(strgs,n,sizeof(char*),pstrcmp),
```

Здесь `strgs` — массив указателей на `char`, `n` — число строк, `sizeof(char*)` — размер указателя, а `pstrcmp` — функция сравнения строк.

Если строки сортируются в алфавитном порядке, то хотелось бы не изобретать «Мерседес», а использовать стандартную функцию `strcmp()`, о которой мы уже говорили в разделе

«Сортировка строк» этой главы. Но вот беда, функция `strcmp()` принимает два указателя на `char`, а функция `qsort()` посылает функции сравнения два указателя на сравниваемые элементы, то есть в нашем случае — *указатели на указатель на* `char`. Значит, для совместной работы `qsort()` и `strcmp()` необходим шлюз, который «уравняет давление» между ними.

Как мы знаем, функция сравнения, совместимая с `qsort()`, должна принимать два указателя на `void`:

```
int pstrcmp(const void *p1, const void *p2){}
```

Эти указатели нужно преобразовать внутри функции к тому типу указателей, которые передаются ей на самом деле, то есть к типу «указатель на указатель на `char`»: `(char **) p1`. После этого нужно раскрыть ссылку, потому что функции `strcmp()` требуются *указатели* на сравниваемые строки. Окончательный вид функции сравнения будет таким:

```
int pstrcmp(const void *p1,const void *p2){
   return strcmp(*(char **)p1, *(char **)p2);
}
```

Задача 7.5 Напишите программу, которая читает строки из файла и сортирует их с помощью `qsort()`.

Иголка, сено и лыко в строку

В этой книге мы довольно мало говорим о стандартных функциях — таких, например, как `printf()` или `scanf()`. Делается это потому, что гораздо важнее понять *душу* языка — указатели, устройство функций, рекурсию, чем затвердить наизусть названия функций, типы принимаемых и возвращаемых ими значений.

Сведения об этих функциях обычно содержатся в специальных help-файлах, поставляемых вместе с компиляторами, и я еще не видел программистов, которые бы этими справочными файлами не пользовались. Наоборот, значительную часть времени программист тратит на поиск нужных функций и на

чтение документации. Образование — вовсе не сумма знаний, а скорее, умение учиться дальше.

Но все-таки, есть функции, о которых стоит рассказать даже в такой небольшой книжке, как эта. Одна из них — `strstr()` — позволяет искать одну строку в другой. Возможность поиска фрагмента текста — едва ли не самое большое преимущество компьютеров перед пишущими машинками и гусиными перьями. Поиск возможен в любом, даже самом примитивном текстовом редакторе. И вот теперь мы так далеко продвинулись в изучении языка Си, что можем организовать поиск сами.

Программа из листинга 7.4, ищет «иголку», затерянную в «сене», или точнее — строку `str2` в строке `str1`.

Листинг 7.4.

```
#include <string.h>
#include <stdio.h>
char *str1=
"сеносеносеносеносеноиголкасеносеносено";
char *str2 = "иголка";
char *substr;

main(){
substr = strstr(str1,str2);
if (substr != NULL)
    printf("%s",substr);
}
```

Результат поиска — указатель на то место в `str1`, откуда начинается `str2` (в случае, если строка найдена) и `NULL` — если нет. Если поиск удачен, программа выводит на экран строку `str1` с того места, где в ней найдена `str2`. В нашем случае на экране появится:

```
иголкасеносеносено
```

Задача 7.6. Подумайте, как можно реализовать поиск по-другому, и напишите собственный вариант функции `strstr()`.

Еще одна задача, о которой стоит поговорить в этом разделе: вставить один фрагмент текста в другой. На первый взгляд, эта задача очень проста. Представим себе, что в строку `"всякое в строку"` нужно вставить `"лыко"`, причем так, чтобы в результате получилось `"всякое лыко в строку"`. Возможное решение задачи содержит листинг 7.5:

Листинг 7.5.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main(){
char *s1 = "всякое в строку";
char *s2 = "лыко";
char *s3,   *cptr;
s3 = calloc(strlen(s1)+strlen(s2)+3,1);
cptr = strtok(s1," ");
strcpy(s3,cptr); /* s3="всякое" */
strcat(s3," ");  /* s3="всякое " */
strcat(s3,s2);   /* s3="всякое лыко" */
strcat(s3," ");  /* s3="всякое лыко " */
while(cptr = strtok(NULL," ")){
    strcat(s3,cptr);
    strcat(s3," ");
}
printf("%s\n",s3);
```

```
}
```

Первая трудность, с которой приходится сталкиваться при решении этой задачи — невозможность записать результат в исходную строку `s1`, потому что там просто нет места для новых символов. Поэтому приходится выделять дополнительную область памяти с помощью функции

```
s3 = calloc(strlen(s1)+strlen(s2)+3,1);
```

Как и функция `malloc()`, `calloc()` возвращает указатель на некую область памяти, но теперь там содержится не «мусор», как в случае `malloc()`, а нули. В случае, когда память выделяется для хранения строк, функция `calloc()` удобна, так как после выделения памяти сразу же получается пустая строка, в которой есть только символы `'\0'`. `Calloc()` (в отличие от `malloc()`) имеет два параметра: число выделяемых «единиц» и размер каждой единицы. В нашем случае выделяются байты, так что второй параметр равен единице. Первый параметр задается с учетом того, чтобы было куда поместить завершающий строку ноль '\0' и два дополнительных пробела (первый — перед словом «лыко», второй — в конце строки).

После выделения памяти можно приступить к вставке «лыка» в строку. Для этого строка «всякое в строку» разбирается на части, разделенные пробелами, и затем частями же переписывается в строку `s3`. После переписывания кусочка «всякое» вставляется «лыко».

Для выделения фрагментов строки используется хитрая функция `strtok()`, которую нужно использовать по-разному в зависимости от того, первый раз она применяется к строке или нет.

Первый фрагмент строки выделяется так:

```
cptr = strtok(s1," ");
```

здесь `cptr` — указатель на первый фрагмент (по сути, он указывает на начало строки `s1`). У функции `strtok()` два параметра: первый — указатель на строку, второй — вспомогательная строка, где указывается символ, который

`strtok()` будет считать разделителем[1]. В нашем случае разделитель один — это пробел.

После выделения первого фрагмента, он переписывается в строку результата функцией `strcpy(s3,cptr)`, которая просто копирует строку `cptr` в строку `s3`. Далее к результирующей строке прибавляется пробел: `strcat(s3," ")`. Функция `strcat()`, которая для этого используется, отличается от `strcpy()` тем, что не уничтожает исходную строку, а дописывает одну строку в конец другой. В нашем случае в конец строки `s3` дописывается пробел.

Далее наша программа добавляет в результирующую строку "лыко" (`strcat(s3,s2)`), затем еще пробел и переходит к разбору остальной строки `s1`. Делается это в цикле `while()`:

```
while(cptr = strtok(NULL," ")){
     strcat(s3,cptr);
     strcat(s3," ");
}
```

Как видим, второе и последующие обращения к `strtok()` выглядят по-другому. Вместо разбираемой строки первым аргументом становится `NULL`. Возвращаемый функцией указатель на очередной элемент строки засылается в `cptr`. Этот элемент присоединяется к результату функцией `strcat(s3,cptr)`. В конце разбора строки (когда не останется фрагментов) функция `strtok()` возвратит `NULL` и работа цикла `while()` прекратится, потому что `NULL` — это на самом деле обычное целое число, равное нулю, а это воспринимается циклом, как невыполненное условие. Цикл `while()` завершится, и строка `s3`, в которой содержится результат работы, будет выведена на экран.

---

[1] Можно указывать сразу несколько разделителей, например,

`" ,."`

Задача 7.7. Кроме символов `/*…*/` комментарии в программах на Си могут обозначаться двумя идущими подряд косыми чертами `//`:

```
strcat(s3," "); // добавить пробел
```

Комментарием будет любой текст справа от значка `//` и до конца строки. К сожалению, такой комментарий довольно поздно попал в стандарт языка Си, и компилятор Turbo C его не понимает. Чтобы с помощью Turbo C компилировать программы с новыми комментариями, необходимо преобразовать их в старые `/*…*/`. Напишите программу, которая это делает.

Глава 8. Основные типы на сборочном конвейере

Перечисления

Язык Си ни за что не стал бы таким популярным, не будь в нем возможности создания новых типов переменных на основе базовых. Базовые типы: `int`, `float`, указатели на переменные разных типов и на функции — всего лишь детали, из которых, как на конвейере, можно собрать очень сложные и причудливые переменные нового типа.

Простейший новый тип — переменные, принимающие только определенные целые значения, которые задаются конструкцией `enum`. Строки

```
enum day {sun, mon, tues, weds, thur, fri,
sat } d1, d2;
```

объявляют две переменные нового типа `day`: `d1` и `d2`, которые могут принимать целые значения от 0 до 6. У каждого целочисленного значения есть свое имя. Так, `sun` (от слова Sunday — воскресенье) соответствует нулю, `mon` (от слова Monday — понедельник) — единице, `tues` (Tuesday — вторник) — двойке, и так до последнего значения `sat` (Saturday — суббота), то есть шести.

Вообще говоря, переменные `d1` и `d2` могут принимать и любые другие значения. Компилятор, скорее всего, не воспримет как ошибку присваивание `d2=100`. Но имена могут быть только у значений, перечисленных в фигурных скобках, следующих за словом `enum`.

Чтобы лучше освоиться с новым типом переменных и заодно познакомиться с некоторыми другими конструкциями языка, напишем программу, которая определяет время года по номеру месяца (см. листинг 8.1).

Листинг 8.1.

```
#include <stdio.h>
int main() {
enum months {JAN = 1, FEB, MAR, APR, MAY,
JUN, JUL, AUG, SEP, OCT, NOV, DEC};
enum months m2;
scanf("%d", &m2);
switch (m2){
    case DEC: case JAN: case FEB:
        printf("ЗИМА\n");
        break;
    case MAR: case APR: case MAY:
        printf("ВЕСНА\n");
        break;
    case JUN: case JUL: case AUG:
        printf("ЛЕТО\n");
        break;
    case SEP: case OCT: case NOV:
        printf("ОСЕНЬ\n");
        break;
    default:
        printf("Ошибка ввода\n");
        break;
}
return 0;
}
```

Новый тип переменной `months` задают в этой программе строки

```
enum months {JAN = 1, FEB, MAR, APR, MAY,
JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

По умолчанию первое имя в фигурных скобках после `enum` имеет нулевое значение. Его можно изменить: `JAN = 1`, и все остальные значения будут соответствовать первому. `FEB` будет равно двум, `MAR` — трем, а последнее значение `DEC` — двенадцати[1].

После задания значений для переменной нового типа, в программе объявляется сама переменная: `enum months m2`, которая принимает значения, перечисленные в `enum{}`. Далее, эта новая переменная читается функцией `scanf("%d", &m2)`. Формат `"%d"` показывает, что `m2` — обычная целочисленная переменная, некоторые значения которой имеют имена. Например, значению 1 соответствует имя `JAN`.

Прочитав переменную `m2`, нужно определить, какому времени года она соответствует. Это делает инструкция `switch(){}`, которую называют *переключателем*. В круглых скобках стоит сама анализируемая переменная `m2` (а может стоять любое целочисленное выражение), дальше перечисляются различные значения, предваренные словом `case`, и действия, которые выполняются, когда выражение в круглых скобках (в нашем случае это анализируемая переменная `m2`) равно одному из значений. Например, строки

```
case DEC: case JAN: case FEB:
        printf("ЗИМА\n");
        break;
```

означают, что в случае, когда `m2` равно `DEC`, `JAN` или `FEB` (то есть 12, 1 или 2), на экране появляется слово «ЗИМА». Слово `break` после вызова `printf()`, передает управление

[1] Вообще, изменить можно любое значение и тогда следующее будет на единицу больше. Если, скажем, написать MAR=7, то APR будет равно 8, MAY — 9 и т.д.

инструкции, следующей непосредственно за переключателем `switch(){}`. Чтобы лучше понять роль инструкций `break`, представим себе[1], что все они исключены из программы, показанной в листинге 8.1. Если с клавиатуры вводится число 1, соответствующее январю, то программа выведет на экран:

```
ЗИМА
ВЕСНА
ЛЕТО
ОСЕНЬ
Ошибка ввода
```

Произойдет это потому, что после выполнения нужной ветки `case:` программа «провалится» на нижние ветви и выполнит все инструкции, которые там встретит.

Нам осталось познакомиться с последней составляющей переключателя — словом `default` и инструкциями, которые за ним следуют. По функции `printf("Ошибка ввода\n")` можно догадаться, что `default:` обозначает начало «мусоросборника» — того места в переключателе, куда попадают значения, не подходящие ни под один из ранее перечисленных случаев[2].

Мы уже говорили о том, что переменные, задаваемые словом `enum{}`, могут принимать любые целые значения, а не только перечисленые в фигурных скобках. Это значит, что можно ввести не только числа от 1 до 12 («легальные» значения переменной, каждое из которых имеет имя), но и какие-то другие, например, 13. Для отлавливания таких случаев неверного ввода и нужен пункт `default:` переключателя.

---

[1] а лучше взять листинг, «закомментировать» в нем инструкции break, заново скомпилировать и посмотреть, что будет

[2] на самом деле, раздел default может быть в любом месте переключателя, но лучше поместить его в конце — так понятней.

В этом разделе мы впервые познакомились с инструкцией `break`, которая выбрасывает программу за пределы переключателя `switch()`. Оказывается, `break` можно использовать и в любом из циклов `while()`, `for()` и `do {} while()`. Встретив `break`, программа покидает цикл и переходит к инструкции, следующей непосредственно за ним. Посмотрим, например, как можно использовать `break` в цикле, подсчитывающем длину строки (листинг 8.2)

Листинг 8.2

```
#include <stdio.h>
int main(){
int i,len;
char x[]="MAMA";
i=0;
len=0;
while(1){
     if(!x[i++]) break;
     len++;
}
printf("%d\n",len);
return 0;
}
```

В условии `if(!x[i++])`, применен оператор логического отрицания `!`, который переводит любое ненулевое значение в ноль, а ноль — в единицу (см. раздел «Условности» главы 4). Когда сработает инструкция `break` (а это случится лишь при достижении конца строки, где находится завершающий нулевой символ `'\0'`), цикл `while()` прервется, программа перейдет к функции `printf("%d\n",len)`, покажет на экране длину строки и завершит работу. Замечу, что условие в

цикле `while(1)` всегда выполняется и, не будь инструкции `break`, программа работала бы вечно.

К `break` очень близка инструкция `continue`, разница лишь в том, что `continue` направляет программу к границе цикла, а не за его пределы. В программе из листинга 8.3 подсчитывается число цифр в строке.

Листинг 8.3

```
#include <stdio.h>
#include <string.h>
int main(){
int i,nofd;
char x[]="123abcd4";
i=0;
nofd=0;
for(i=0;i<=strlen(x);i++){
    if(x[i] > '9' || x[i] < '0')
        continue;
    nofd++;
}
printf("%d\n",nofd);
return 0;
}
```

Узнать цифру можно по числу, которым она кодируется. Это число получается автоматически в языке Си, если написать нужную цифру в одинарных кавычках. Так, например, число, которым кодируется единица, равно `'1'`. Можно, конечно, сразу написать вместо `'1'` число 48, потому что именно им обычно кодируется символ «1». Но запись ‘1’ более универсальна и годится для компьютеров, где символ «1» кодируется другим числом. Отличить цифру от буквы или

иного знака очень просто: числа, которыми кодируются цифры, идут, как правило, подряд. Значит, если символ в строке не меньше `'0'` и не больше `'9'` — это цифра. В программе перебираются все символы строки (всего их `strlen(x)`) и если символ — цифра, увеличивается на единицу счетчик `nofd`. Если же символ — не цифра, срабатывает инструкция `continue`, программа обходит увеличение счетчика, и начинается новый оборот цикла.

Двухмерные массивы и указатели на…

До сих пор нам встречались только одномерные массивы с одним индексом (например, `x[i]`), в которых хранятся строки текста, числа или указатели. Одномерные массивы хороши для хранения «одномерных» же данных, например, последовательности среднесуточных температур за месяц. Но для хранения картинок, у которых, как известно, два измерения — ширина и высота, больше подходят *двухмерные массивы*, в которых есть строки и столбцы (рис. 8.1)

Рис. 8.1. «Дедка» за «репкой», «бабка» за «дедкой»…

Для доступа к элементу двухмерного массива нужны два индекса: первый указывает строку, второй — на столбец. Как обычно в Си, нумерация строк и столбцов начинается с нуля, поэтому буква `'П'`, стоящая на рис. 8.1 в нулевой строке и втором столбце, будет элементом `a[0][2]` двухмерного массива `a[][]`.

Двухмерный массив, как и любой другой, должен быть объявлен. Например, массив, показанный на рис. 8.1, содержит 15 символов и объявляется как

```
char a[3][5];
```

Но память компьютера, как мы знаем, одномерна, в ней нет ничего кроме идущих подряд ячеек памяти, поэтому компилятор вынужден будет «вытянуть» двухмерный массив в линейку по *строкам*: сначала в памяти разместится первая строка, потом — вторая, за ней третья: «дедка» за «репкой», «бабка» за «дедкой».

Так как двухмерный массив — не более чем удобная форма записи, можно попробовать заменить его одномерным с такими же элементами, например таким, как массив `s[]`, показанный на рисунке 8.1. Элемент одномерного массива `s[i*ncols + j]`соответствует элементу двухмерного массива, расположенному в строке `i` и столбце `j` (`ncols` — число столбцов в соответствующем двухмерном массиве). Буква `'П'`, стоящая в нулевой строке и втором столбце массива `a[][]`(то есть элемент `a[0][2]`), будет найдена в массиве `s[]` на 0*6+2, то есть втором месте.

Как видим, для нормальной работы с одномерным массивом, в котором расположены двухмерные данные, необходимо знать не только строку и столбец, где помещается нужный элемент, но и *общее число столбцов* двухмерного массива.

Если иметь дело только с двухмерным массивом, то для доступа к элементу число столбцов знать не нужно. Так, для доступа к элементу, стоящему, скажем, во второй строке и втором столбце, нужно просто написать `a[2][2]`.

В листинге 8.4 показана программа, создающая таблицу, состоящую из трех строк и трех столбцов, в которой диагональные элементы (те, в которых номер столбца равен номеру строки) равны единице, все же остальные — нулю. Такая таблица называется в математике единичной *матрицей*. Матрица может храниться в двухмерном массиве `a[3][3]` с тремя столбцами и тремя строками, а может и в одномерном массиве `s[9]`.

Чтобы присвоить элементам массива значения, перебираются все возможные номера строк и столбцов (это делают два вложенных цикла `for()`)[1]. Для каждой пары индексов `i` и `j` условное выражение `a[i][j] = (i == j) ? 1 : 0` решает, чему будет равен элемент массива — нулю или единице.

Листинг 8.4

```
#include <stdio.h>
#define NCOLS 3
int main(){
char a[3][3];
char s[9];
int i,j;
for(i=0;i<3;i++)
for(j=0;j<3;j++){
    a[i][j] = (i == j) ? 1 : 0;
    s[i*NCOLS + j] = (i == j) ? 1 : 0;
}
for(i=0;i<3;i++)
for(j=0;j<3;j++)
    printf("%6d%c",
    a[i][j],(j == 2)? '\n':' ');
printf("\n");
```

[1] Возможно, не всем понятно, какие действия совершаются во внешнем цикле `for(i=0;i<3;i++)`. Поскольку «тело» цикла не обозначено фигурными скобками, в него попадает только идущая следом инструкция — сам внутренний цикл `for(j=0;j<3;j++){}`

```
for(i=0;i<3;i++)
for(j=0;j<3;j++)
    printf("%6d%c",
    s[i*NCOLS+j],(j == 2)?'\n':' ');
}
```

Условные выражения применяются и при выводе на экран обеих получившихся матриц (двухмерной и одномерной). Здесь все значения строк и столбцов так же перебираются двумя вложенными циклами. Для вывода на экран отдельных строк матрицы используется функция

```
printf("%6d%c",a[i][j],(j == 2)? '\n':' ');
```

Формат `%6d` задает вывод целого числа, занимающего шесть позиций, формат `%c` предназначен для вывода одного символа. Сам символ определяется условным выражением `(j == 2)? '\n':' '`, смысл которого прост: символ почти всегда равен пробелу, и лишь когда j == 2, то есть напечатана последняя строка матрицы, он равен `'\n'`. То есть, перед выводом на экран следующей матрицы, хранящейся в одномерном массиве, программа пропускает строку.

Как видно из листинга 8.4, двухмерный массив чуточку удобнее одномерного, но и эти преимущества почти теряются, если его нужно передать функции. Нам известно, что сами массивы функции не передаются. Передается лишь указатель на первый элемент. Но в случае двухмерных массивов этого мало, приходится указывать еще число столбцов, то есть, по сути, *длину строки (*напомню, что двухмерные массивы располагаются в памяти *построчно:* сначала первая строка, за ней вторая и т.д.). Зная начало массива и длину строки, функция без труда (так же, как в одномерном массиве, заменяющем двухмерный) сможет найти элемент, стоящий в любой строке и любом столбце.

В листинге 8.5 показана программа, которая использует для создания единичной матрицы специальную функцию `set1()`.

Листинг 8.5.

```
#include <stdio.h>
#define NCOLS 3
void set1(char [][3], int);
int main(){
char  a[3][3];
int i,j;
set1(a,3);
for(i=0;i<3;i++)
for(j=0;j<3;j++)
    printf("%6d%c",
    a[i][j],(j == 2)? '\n':' ');
return 0;
}
void set1(char matr[3][3],int dim){
int i,j;
for(i=0;i<dim;i++)
for(j=0;j<dim;j++)
    matr[i][j] = (i == j) ? 1 : 0;
}
```

В списке параметров функции явно указаны размеры массива `matr[3][3]`. Впрочем, функции, как мы уже поняли, нужно знать только размер строки (то есть, число столбцов). Значит, заголовок функции может быть и таким:

```
void set1(char matr[][3],int dim)
```

Запись `matr[][3]` или `matr[3][3]`в списке параметров функции более-менее понятна и естественна, но тем, кто пытлив, хочется понять, *что же на самом деле* передается функции вместо двухмерного массива.

Мы хорошо знаем, что вместо одномерного массива функции передается указатель на его первый элемент. Точно так же, не передается функции и двухмерный массив. Вместо него передается *указатель на массив,* число элементов которого равно длине строки двухмерного массива. Например, функция, работающая с массивом `char a[3][5]`, получает *указатель на массив* из пяти элементов типа char, который объявляется так:

```
char (*ap)[5];
```

Чтобы понять такую запись, достаточно вспомнить указатель на функцию (раздел «Указатель на функцию» главы 7), который объявляется как

```
char (*f) (); /*указатель на функцию, возвращающую значение char) */
```

Скобки `(*ap)` в объявлении обязательны, ведь `char *ap[5]` — это *массив указателей на char* — совсем не то, что нам нужно.

Указатель на массив — довольно странный указатель. Мы привыкли к тому, что оператор раскрытия ссылки превращает указатель в то, на что он указывает. Если, скажем, `ip` — указатель на `int`, то `*ip` — само значение типа `int`. Но пусть теперь `ia` — указатель *на массив*. Чем тогда должно быть `*ia`? Массивом? Очевидно, нет. Ведь язык Си не работает непосредственно с массивами. Имя массива не выступает как нечто целое, а лишь как *адрес его начала*. Поэтому раскрытие ссылки, примененное к указателю на массив, даст снова указатель, но уже на *нулевой элемент* массива. Еще одно раскрытие ссылки — и перед нами конкретное значение элемента массива.

Теперь представим себе двухмерный массив, который, как и положено, расположен строчка за строчкой в памяти компьютера. Если `ia` — указатель на первую строку двухмерного массива, то через такой указатель возможен доступ к любому элементу массива. Ведь первое раскрытие ссылки для такого указателя переместит нас к началу выбранной строки. Если `ip` указывает на нулевую строку

массива, то `*(ip+2)` указывает на начало второй строки, а еще одно раскрытие ссылки `*(*(ip +2)+3)` даст значение третьего элемента второй строки, то есть значение из второй строки и третьего *столбца*. Раскрытие ссылки можно, как мы знаем, записать с помощью индексов:

```
*(*(ip +2)+3) == *(ip[2] + 3) == ip[2][3].
```

То есть, указателем на массив, если придать ему правильное значение, можно пользоваться, как *именем двухмерного массива,* и наоборот — имя двухмерного массива *есть указатель на его нулевую строку*[1].

Листинг 8.6

```
#include <stdio.h>
int main(){
char a[3][5];
char (*ap)[5];
ap=a;
a[0][2]='П';
printf("%c\n",ap[0][2]);
return 0;
}
```

Короткая программа (см. листинг 8.6) показывает, что можно объявить двухмерный массив `a[3][5]` и указатель на массив из пяти элементов `(*ap)[5]`. Только что объявленный указатель содержит «мусор». Но если настроить его на начало двухмерного массива, приравняв `ap` и `a` (`ap=a`), то указатель становится *другим именем* массива. В программе из листинга 8.6 элементу массива `a[0][2]` сначала присваивается значение `a[0][2]='П'`, а затем на экран

[1] Нумерация строк и столбцов начинается с нуля!

выводится элемент массива с тем же значением, но с *другим* именем

```
printf("%c\n", ap[0][2]),
```

То, что имя двухмерного массива оказалось указателем на массив, наводит на мысль, что двухмерный массив в языке Си — не что иное как *массив массивов*. Действительно, имя одномерного массива указывает на его первый элемент. То есть одномерный массив — это массив *элементов* типа `char`, `int` и т.д. Раз имя двухмерного массива само указывает на массив, то двухмерный массив *состоит из массивов*. А раз так, параметр функции, получающей имя двухмерного массива (см. листинг 8.5) может быть *указателем на массив*. Функция `set1()` из листинга 8.5, может быть записана как

```
void set1(char *(matr)[3],int dim){
int i,j;
for(i=0;i<dim;i++)
for(j=0;j<dim;j++)
     matr[i][j] = (i == j) ? 1 : 0;
}
```

Задача 8.1. Если двухмерный массив — массив *массивов*, то что тогда такое трехмерный массив a[][][]? Как будет выглядеть список параметров функции, принимающей трехмерный массив?

Хранение и переработка двухмерных массивов

Тем, кто внимательно прочитал предыдущий раздел, должны быть хорошо видны недостатки двухмерных массивов: невозможность поменять их размеры и неудобства при передаче их функциям. Особенно неприятен последний недостаток. Ведь получается, что для обработки массива с тремя столбцами нужно писать одну функцию, а для массива с пятью — другую!

К счастью, язык Си — достаточно гибкий, и позволяет создавать массивы таким образом, что в функциях, работающих с ними, не нужно явно указывать число столбцов. Тип данных, на основе которого строится такой массив, нам уже знаком. Это *массив указателей* (см. раздел «Массивы указателей» главы 6).

Листинг 8.7

```
#include <stdio.h>
#include <stdlib.h>
void set(int **);
#define NROW 3
#define NCOL 5
int main(){
int *arr[NROW], i;
for (i=0;i<NROW;i++)
    arr[i]=malloc(NCOL * sizeof(int));
set(arr);
printf("%c\n", arr[2][3]);
for (i=0;i<NROW;i++)
    free(arr[i]);
return 0;
}
void set(int **a){
    a[2][3]='П';
}
```

В программе из листинга 8.7, «остовом» будущего двухмерного массива служит массив указателей `*arr[NROW]`, где NROW — число строк. Цикл

```
for (i=0;i<NROW;i++)
```

```
arr[i]=malloc(NCOL * sizeof(int));
```

выделяет `NROW` участков памяти по `NCOL` переменных типа `int` в каждом[1]. То есть, в создаваемом двухмерном массиве будет `NROW` строк и `NCOL` столбцов. Заметим, что получившаяся структура будет внутри устроена совсем не так, как двухмерный массив, о котором рассказывалось в предыдущем разделе. «Настоящий» двухмерный массив располагался непрерывно в памяти компьютера, строка за строкой. Строки двухмерного массива, «растущего» из массива указателей, разбросаны в беспорядке, потому что участок памяти, выделяемый при следующем обращении к `malloc()`, может начинаться совсем в другом месте (рис. 8.2)

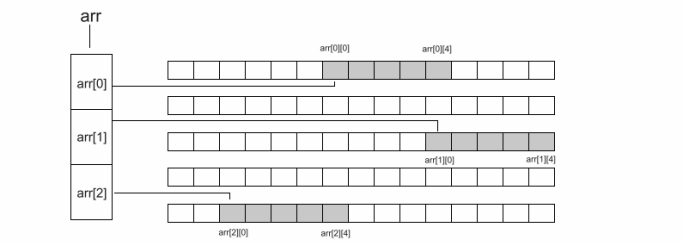

Рис. 8.2. Двухмерный массив, растущий из массива указателей, разбросан по памяти

Но, несмотря на иное внутреннее устройство, созданной структурой данных можно пользоваться как двухмерным массивом. Действительно, если `arr[]` — массив указателей, каждый из которых направлен на свободный участок памяти длиной NCOL*(размер переменной), то `arr[0]` указывает на начало строки под номером 0, arr[1] — на начало первой строки и т.д. Чтобы получить значение третьего элемента второй строки, нужно взять указатель на начало второй строки, то есть `arr[2]`, прибавить к нему тройку и раскрыть ссылку оператором *: `*(arr[2] + 3)`. На языке квадратных скобок

[1] Естественно, в реальной программе каждый возвращенный malloc() указатель нужно проверять — не равен ли он NULL.

это выглядит как `arr[2][3]`. То есть, `arr[][]` смотрится типичным двухмерным массивом, хотя само имя `arr` уже не указатель на массив (как это было с «настоящим» двухмерным массивом), а *указатель на указатель*. Действительно, если имя обычного массива, состоящего из чисел или букв, есть указатель на его нулевой элемент, то чем может быть имя массива, каждый элемент которого — указатель? Только указателем на указатель. Поэтому функция `set()` из листинга 8.7 и принимает указатель на указатель на `int (int **a)`, что позволяет ей работать с двухмерными массивами любых размеров[1].

У массива, который мы только что научились создавать «на лету», наряду с достоинствами, есть определенный недостаток: в нем фиксированное число строк, равное числу элементов в массиве указателей `arr[]`. Этот недостаток преодолевается играючи, ведь функция `malloc()`, конечно же, может выделять память не только под *переменные* типа `int`, но и под указатели!

Программа, где двухмерный массив создается «на лету», будет отличаться от показанной в листинге 8.7 несколькими строчками. Во-первых, двухмерный массив будет расти не из массива указателей, а из указателя на указатель `int **a`, который получает нужное значение от функции `malloc()`[1]:

```
arr=malloc(NROW * sizeof(int *));
```

Еще одно отличие: в конце программы, после освобождения памяти, занимаемой строками

```
for (i=0;i<NROW;i++)
```

---

[1] Эта функция, конечно же, не сможет работать с «настоящими» двухмерными массивами, о которых говорилось в предыдущем разделе. Ей нужен двухмерный массив, растущий из массива указателей

[1] Обратите внимание, в `malloc()` стоит `sizeof(int *)` — размер *указателя* на `int`. Размеры переменной типа `int` и указателя на `int` могут отличаться.

```
free(arr[i]);
```

нужно еще будет освободить память, занимаемую указателями: `free(arr);`

Задача 8.2. Напишите программу, в которой двухмерный массив создается полностью динамически: указатель на указатель — массив указателей — сами строки.

Задача 8.3. Подумайте, как динамически создать непрерывный двухмерный массив, чьи строки не разбросаны по памяти, а идут вплотную.

Записи

Обед, как видно, не составлял у Ноздрева главного в жизни; блюда не играли большой роли: кое-что и пригорело, кое-что и вовсе не сварилось. Видно, что повар руководствовался более каким-то вдохновеньем и клал первое, что попадалось под руку: стоял ли возле него перец — он сыпал перец, капуста ли попалась — совал капусту, пичкал молоко, ветчину, горох — словом, катай-валяй, было бы горячо, а вкус какой-нибудь, верно, выдет.

Н.В.Гоголь, «Мертвые души»

До сих пор нам встречались (поодиночке или в массивах) разнообразные, но однородные переменные. Но жизнь собирает их в группы и кучки, и часто бывает так, что один объект обладает многими свойствами, которые в одной переменной не выразишь.

Возьмем, к примеру, обед у Ноздрева. Хоть он и не составлял главного в жизни, но ведь какие-то продукты к нему закупались? И не один горох, а еще и капуста и ветчина и перец…Значит, с обедом связано сразу несколько переменных. Причем, обед — вещь довольно регулярная, и Ноздрев, надо полагать, обедал каждый день, если, конечно, не уезжал на ярмарку. Значит, обедам Ноздрева можно сопоставить целый ряд *записей* (каждый день, в зависимости от вдохновения

повара, запись будет иной): сколько было съедено гороха, ветчины и т.д.

Такие записи в языке Си создаются с помощью слова `struct`. Например, запись об обеде Ноздрева может выглядеть так:

```
struct dinner {
float pepper;      /* перец   */
float cabbage;     /* капуста */
float peas;        /* горох   */
float ham;         /* ветчина */
};
```

Запись `struct dinner{}` создает лишь новый *тип*. *Переменные* этого типа объявляются по тому же принципу, что и перечисления: сначала идет слово `struct` , затем название записи (в нашем случае `dinner`) и далее — имя самой переменной:

```
struct dinner dinner_with_Chichikov
```

или даже массив записей, в котором можно хранить сведения о съеденном за год:

```
struct dinner dinners[365];
```

Но как присвоить значение конкретной записи? Ответ очевиден: никак. Ведь запись хранит разнородные элементы, горох имеет мало общего с капустой, и поэтому присвоить значение можно только отдельному *полю* записи. Предположим, на обеде Ноздрева съедено 5 граммов перца, 3 килограмма капусты, полкило гороха и килограмм ветчины. Записывается это следующим образом:

```
dinner_with_Chichikov.pepper  =0.005;
dinner_with_Chichikov.cabbage =3.0;
dinner_with_Chichikov.peas    =0.5;
dinner_with_Chichikov.ham     =1.0;
```

Для массива записей все выглядит чуть посложнее. Предположим, что обед состоялся в 150-й день года. Тогда количество съеденного перца запишется так:

```
dinners[150].pepper=0.005;
```

Листинг 8.8 показывает, как объявляется запись об обеде и как ее полям присваиваются начальные значения.

Листинг 8.8.

```
#include <stdio.h>
int main(){
struct dinner{
     float pepper;/* перец   */
     float cabbage;   /* капуста */
     float peas;  /* горох   */
     float ham;   /* ветчина */
}    d={0.005,3.0,0.5,1.0};
printf("%f\n",d.peas);
return 0;
}
```

Как видим, можно совместить объявление типа переменной и самой переменной. Строки

```
struct dinner {
…
}
d={…};
```

не только объявляют запись **d** типа **dinner**, но и присваивают начальные значения ее полям (эти значения разделяются запятыми внутри фигурных скобок). Функция `printf("%f\n",d.peas)` выводит на экран третье поле записи, и можно убедиться (скомпилировав и запустив

программу), что d.peas равно 0.5. Заметим, что записывать начальные значения полей внутри фигурных скобок можно только при объявлении записи. Значения полей в теле программы устанавливаются другими способами. Один из возможных — простое копирование. В листинге 8.9 показана программа, где создаются две записи dinner: d1 и d2. Полю d1.ham присваивается значение 1.0, и запись d1 целиком копируется в d2[1]. А дальше функция printf() выводит на экран поле d2.ham, равное, как и положено, 1.0.

Листинг 8.9.

```
#include <stdio.h>
int main(){
struct dinner{
float pepper; /* перец   */
float cabbage;/* капуста */
float peas;   /* горох   */
float ham; /* ветчина */
};
struct dinner d1,d2;
d1.ham=1.0;
d2=d1;
printf("%f\n",d2.ham);
return 0;
}
```

[1] Остальные поля записи d1, как и d2, содержат «мусор»

Записи и функции

Ноздревский повар (см. предыдущий раздел), больше доверял вдохновению, чем рецептам. Поэтому предугадать, сколько продуктов он истратит на конкретный обед, практически невозможно. Но можно попробовать предсказать, сколько их могло было быть съедено за год, если научиться моделировать его вдохновение.

Предположим, что вес продукта, выбираемого поваром, случайным образом колеблется в жестких пределах — от нуля до некоторого максимального значения. Если речь идет о перце, то повар с равной вероятностью может выбрать любое количество от нуля, до, скажем, 100г. Тогда его поведение можно описать с помощью функции `rand()`, равномерно выдающей случайные целые числа в диапазоне от нуля до `RAND_MAX`[1]. Функция, моделирующая поведение повара, показана в листинге 8.10.

Листинг 8.10.

```
void cook(struct dinner d[]){
int i;
for(i=0;i<365;i++){
     d[i].pepper = 0.1 *
     ((double) rand())/RAND_MAX;
     d[i].cabbage = 2.0 *
     (double) rand())/RAND_MAX;
     d[i].peas =  1.0 *
     ((double)rand())/RAND_MAX;
     d[i].ham=    0.5 *
     ((double) rand())/RAND_MAX;
```

[1] Это значение может отличаться в разных компиляторах. Для TURBO C `RAND_MAX` равно 32767

```
}
}
```

Единственный параметр `struct dinner d[]` функции `void cook` показывает, что функция принимает *массив записей* типа `dinner`. Внутри функции перебираются 365 элементов массива (каждый элемент — отдельная запись). Действия повара имитируются случайной функцией rand(). Количество капусты, съеденной в i-й день года, равно

```
d[i].cabbage=2.0*((double)rand())/RAND_MAX,
```

то есть случайному числу между нулем и двойкой. Действительно, отношение `(double)rand()/RAND_MAX` меняется от 0 до 1.0 (потому что сама `rand()` выдает случайные числа от `0` до `RAND_MAX`). Значит, `2*((double)…)` меняется от нуля до двойки.

Научившись имитировать действия повара, можно переходить к подсчету съеденного за год. Для этого напишем функцию, которая принимает тот же массив записей, суммирует съеденное и возвращает запись того же типа функции `main()` (см. листинг 8.11).

Листинг 8.11.

```
struct dinner sum(struct dinner d[]){
int i;
struct dinner s={0,0,0,0};
for(i=0;i<365;i++){
    s.pepper     += d[i].pepper;
    s.cabbage    += d[i].cabbage;
    s.peas       += d[i].peas;
    s.ham        += d[i].ham;
}
return s;
```

```
}
```

В этой функции объявляется вспомогательная запись `struct dinner s`, поля которой будут содержать сумму съеденных за год гороха, перца и т.д. Перед суммированием все поля записи должны быть равны нулю, поэтому при объявлении `s` в фигурных скобках указываются их начальные значения:

```
struct dinner s={0,0,0,0};
```

Записи по всем 365 дням суммируются в цикле `for(){}`. После его завершения сумма съеденного за год окажется в записи `s`, которая и будет возвращена функции `main()`. Вся программа (без самих функций `sum()` и `cook()`) показана в листинге 8.12.

Листинг 8.12.

```
#include <stdio.h>
#include <stdlib.h>
struct dinner{
float pepper; /* перец   */
float cabbage;/* капуста */
float peas;   /* горох   */
float ham; /* ветчина */
};
void cook(struct dinner[]);
struct dinner sum(struct dinner[]);
int main(){
int i;
struct dinner d[365], tot;
cook(d);
tot=sum(d);
printf("tot.pepper=%f\n",tot.pepper);
```

```
printf("tot.cabbage=%f\n",tot.cabbage);
printf("tot.peas=%f\n",tot.peas);
printf("tot.ham=%f\n",tot.ham);
return 0;
}
```

После включения необходимых заголовков (<stdio.h> нужен для функции printf(), а <stdlib.h> — для функции rand()) в программе объявляется запись struct dinner{}, чтобы другие функции о ней знали. Далее объявляются сами функции. Одна (void cook(struct dinner[])) ничего не возвращает, другая (struct dinner sum(struct dinner[])) возвращает запись типа dinner. В main() сначала вызывается функция cook() с аргументом d (массивом записей), затем с тем же аргументом вызывается sum(). Возвращенная функцией sum(d) сводка съеденного за год, хранится в записи tot, отдельные поля которой выводятся на экран функциями printf().

Результаты работы программы из листинга 8.12, будут зависеть от конкретной реализации rand(). У каждого компилятора она своя. Программа, скомпилированная Turbo C, выводит на экран

```
tot.pepper=18.358107
tot.cabbage=358.229645
tot.peas=183.994461
tot.ham=89.342995
```

Задача 8.4. Подумайте, почему программа выводит такие цифры.

Задача 8.5. Напишите программу, которая с помощью функции rand() вычисляла бы число π (ПИ).

Указатель на запись

Было даже предложено увольнять программиста, который слишком изощренно программирует. Действительно, для такого предложения есть основания, так как если написана программа, которая так сложна и непонятна, что только автор может в ней разобраться, то она никому и не нужна.

Д.Ван Тассел, «Стиль, разработка, эффективность, отладка и испытание программ»

В сущности, мы уже встречались с указателем на запись. Ведь параметр функции `struct dinner sum(struct dinner d[])`, показанной в листинге 8.11, — не что иное как массив записей. А мы знаем, что вместо массива функции передается указатель на его первый элемент. В данном случае — указатель на запись. Поэтому заголовок функции `sum()` может быть таким:

```
struct dinner sum(struct dinner *d)
```

Раз `d` — указатель на запись, к нему можно применить оператор раскрытия ссылки и переписать функцию `sum()` так:

```
struct dinner sum(struct dinner d[]){
int i;
struct dinner s={0,0,0,0};
for(i=0;i<365;i++){
    s.pepper     += (*(d+i)).pepper;
    ...
    s.ham        += (*(d+i)).ham;
}
return s;
}
```

Двойные скобки `(*(d+i))` здесь необходимы, потому что точка `'.'` «сильнее» оператора `'*'`[1] и `*(d+i).pepper` означает раскрытие ссылки по отношению к полю `pepper`, что бессмысленно, ведь `d.pepper` — число, а не указатель.

«Лишние» скобки, выделяющие раскрытие ссылки, неудобны, поэтому создатели языка Си придумали новый оператор, применимый только к записям. Если `d` — указатель на запись, то `d->pepper` есть значение ее поля.

Оператор `'->'` состоит из двух символов: «минуса» `'-'` и «стрелки» `'>'`. Применив его в функции, суммирующей поля записей, получим следующее:

```
struct dinner sum(struct dinner d[]){
int i;
struct dinner s={0,0,0,0};
for(i=0;i<365;i++){
     s.pepper     += (d+i)->pepper;
     ...
     s.ham        += (d+i)->ham;
}
return s;
}
```

Скобки `(d+i)` здесь необходимы, потому что слева от оператора `->` должен стоять указатель на запись, а не целое число.

Чтобы избавиться от скобок, можно передвигать указатель к следующей записи в конце цикла `for{}`:

---

[1] В приложении А есть таблица приоритетов различных операторов языка Си.

```
for(i=0;i<365;i++){
    s.pepper      += d->pepper;
    ...
    s.ham         += d->ham;
    d++;
}
return s;
}
```

Наконец, можно совместить продвижение указателя и суммирование полей:

```
for(i=0;i<365;i++){
    s.pepper      += d->pepper;
    s.cabbage     += d->cabbage;
    s.peas        += d->.peas;
    s.ham         += (d++)->ham;
}
```

В последней инструкции цикла к полю `s.ham` сначала прибавляется `d.ham`, и лишь потом `d` продвигается к следующей записи. Поскольку оператор `->` один из самых «сильных», скобки в последней строке не нужны, и можно записать

```
s.ham+=d++->ham;
```

Многие программисты радуются, когда удается написать что-то подобное. Чем непонятней программа, тем выше их самооценка. Им кажется, что только настоящий профессионал способен превратить программу в подобие клинописи.

На самом деле, профессионалы стараются писать программы как можно проще. Уж они-то знают, что через пару недель слишком изощренно написанная программа перестанет быть понятной им самим. При создании любой сложной вещи

необходимо следовать KISS-принципу: KEEP IT SIMPLE, STUPID (делай проще, дурачок!).

Помимо эстетических соображений, которые однозначно против такого стиля, есть еще и здравый смысл, который подсказывает нам, что программы редко пишутся в одиночку. Как правило, программа — лишь часть большого проекта. Чтобы она была понятна коллегам, нужно не просто писать, а писать *просто*, следуя каким-то общим стандартам. Наконец, нужно помнить, что хорошим программам предстоит долгая жизнь, на протяжении которой их будут переписывать разные люди. И, выпуская программу в свет, автор должен быть уверен в том, что в ней можно разобраться.

Язык Си (к сожалению или счастью) не навязывает программистам какой-то определенный стиль. На Си можно писать просто и понятно, а можно так, как показано в листинге 8.13. Выбор за вами.

Листинг 8.13. Программа, выводящая на экран «Hello, World!»

```
#define _ putchar
#define __ ^
#define ___ /
#include <stdio.h>
int main()
{_((2*2*2*2*2*2*3*41)___
64__(3*17));_((2*2*2*13*89)___
104__(2*2*3*5));_((2*2*2*23*61)___
92__(2*11));_((2*2*2*2*2*3*3)___
4__(2*2*3*3));_((2*2*2*2*7*13)___
26__(3*29));_((2*2*3*7*19)___
28__(3*7));_((2*2*2*2*13*29)___
104__(2*13));_((2*2*2*2*2*2*3*19)___
114__(3*29));_((2*2*5*107)___
```

```
20__(2*2));_((3*11*97)___
97__(83));_((2*2*2*2*2*3*11)___
11__(2*2*3));_((2*5*11*13)___
55__(2*3*3*7));_((2*3*7*83)___
83__(2*2));_((79)___
79__(11));
return 0;
}
```

Задача 8.6. Скомпилируйте программу, показанную в листинге 8 13. Напишите похожую программу, которая выводит на экран слово «МАМА».

Связанные записи

Записи и указатели на них позволяют создать причудливые структуры данных, где отдельные числа, слова или другие объекты связаны друг с другом. То, что попадалось нам до сих пор, было рядом разбросанных или сваленных в массив слов, указателей или чисел. Конечно, мы пытались навести порядок во всем этом, расставляли слова в алфавитном порядке, находили суммы, максимальные значения, но все же это были «отдельные» слова или числа. Записи, с которыми мы только что познакомились, позволяют связать между собой различные данные, так, чтобы ясно была видна их структура.

Типичный пример связанных данных — очередь. Герои сказки про репку являются нам не в беспорядке, а в строгой очередности: репка — дедка — бабка — внучка — жучка — кошка — мышка.

Отразить связь персонажей можно, связав записи друг с другом. Запись, соответствующая каждому персонажу сказки, может выглядеть так:

```
struct personage {
char *hero;
struct personage *next;
```

```
} ;
```

В ней всего два поля. Первое — `char *hero` — простой указатель на имя героя. Второе содержит указатель на запись того же типа, и от этого с непривычки может закружиться голова. Кажется, что попадаешь в порочный круг, из которого нет выхода. Но указатель — не сама запись. В нем хранится лишь *адрес* следующего объекта.

В программе из листинга листинге 8.14, создаются две записи — `repka` и `dedka`. Если в них заполнить только первые поля `repka.hero = "РЕПКА"`, `dedka.hero = "ДЕДКА"`, то оба объекта останутся одинокими и разобщенными. Но если указатель в записи `repka` направить на запись `dedka(repka.next=&dedka;)`, то между объектами возникает односторонняя связь. Объект `repka` знает путь к объекту `dedka`. Но дедка не знает пути к репке, потому что указатель `dedka.next` равен `NULL` (см. рис. 8.3)

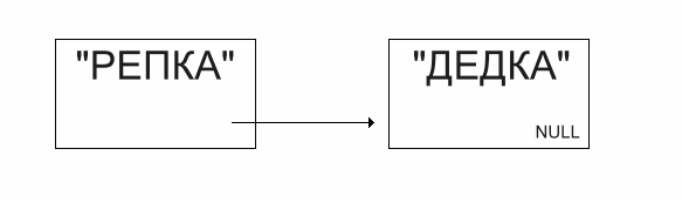

Рис. 8.3. Связанный список: одна запись указывает на другую

Раз `repka.next` указывает на запись `dedka`, то `repka.next->hero` есть поле `hero` записи `dedka` и `printf()` выведет на экран «ДЕДКА»

Листинг 8.14.

```
#include <stdio.h>
#include <stdlib.h>
struct personage {
char *hero;
```

```
struct personage *next;
};
int main(){
struct personage repka, dedka;
repka.hero="РЕПКА";
dedka.hero="ДЕДКА";
dedka.next=NULL;
repka.next=&dedka;
printf("%s\n",repka.next->hero);
}
```

Пример, который мы только что разобрали, дает лишь представление о связанных объектах. Чуточку приблизиться к реальности поможет пример создания и печати *связанного списка* (см. листинг 8.15). Связанный список — это ряд записей, последовательно ссылающихся друг на друга. Каждой записи известен лишь адрес следующего элемента. Последняя запись в списке вместо указателя на следующую запись содержит `NULL`.

Листинг 8.15.

```
#include <stdio.h>
#include <stdlib.h>
struct personage {
    char *hero;
    struct personage *next;
};
void display(struct personage *);
int main(){
int i;
struct personage *repka, *root;
```

```
char *heroes[]= {"репка" ,"дедка",
"бабка","внучка", "жучка",
"кошка","мышка",NULL};
repka=malloc(sizeof(struct personage *));
root=repka;
root->hero=heroes[0];
i=1;
while(heroes[i]){
    repka=repka->next=malloc(sizeof(struct
    personage *));
    repka->hero=heroes[i++];
}
repka->next=NULL;
display(root);
return 0;
}
void display(struct personage *p){
do{
    printf("%s\n",p->hero);
} while(p=p->next);
}
```

Связанный список создается из массива указателей `char *heroes[]= {"репка",...,NULL}`. Сначала создается «затравка» — первая запись, указатель на которую запоминается в переменной `root`. Все остальные записи динамически создаются и связываются друг с другом в цикле `while`:

```
while(heroes[i]){
```

```
repka=repka->next=malloc(sizeof(struct
personage *));
repka->hero=heroes[i++];
}
```

Строка `repka=repka->next=malloc(Ц))` выделяет участок памяти для новой записи. Указатель на него запоминается в поле `repka->next`, а затем и в переменной `repka`[1]. Если не сделать этого последнего шага, новая запись не будет создана, программа будет выделять память под все новые переменные, но указатель на них будет помещать всегда в поле repka->next одной и той же записи. И от связанного списка останется только одна запись, у которой в поле `hero` будет «МЫШКА», а в поле `repka->next y NULL`.

Функция `display()`, быть может, лучше всего дает почувствовать красоту и лаконичность связанных списков. Начав с переменной `root`, которая указывает на первую запись в списке, `display()` высвечивает поле `hero`, затем получает указатель на новую запись и так до тех пор, пока указатель не станет нулевым. `NULL`, как мы уже говорили, действительно равен нулю. Поэтому цикл `do{} while()` перестанет выполняться и вывод на экран прекратится.

Задача 8.7. Можете ли вы сказать, не компилируя и не запуская программу, показанную в листинге 8.15, что она выведет на экран?

Задача 8.8. В программе из листинга 8.15 записи, составляющие связанный список, создаются функцией `malloc()`. Как освободить память, занимаемую связанным списком? Имейте в виду, что если сначала освободить память, занимаемую первой записью `free(root)`, то освободить следующую запись `free(root->next)` уже нельзя, потому

---

[1] Присваивания `a=b=c` выполняются в Си справа налево, то есть сначала значение `c` присваивается переменной `b`, а затем значение `b` присваивается переменной `a`.

что Си не гарантирует, что указатель `root->next` сохранится после освобождения `root`.

Связанные списки, о которых говорилось в этом разделе, могут показаться искусственными и малополезными, потому что затрудняют доступ к отдельному элементу. В массивах можно прямо обратиться к элементу под номером десять, а в связанных списках для этого нужно пройти девять предыдущих. Зато в массивы очень трудно вставить новое значение. Чтобы освободить место в массиве, нужно его «раздвинуть», для чего придется переписать на новое место весь его «хвост». Чтобы вставить новое значение в связанный список, достаточно поменять пару указателей (рис. 8.4). То есть, все как в жизни: идеального хранилища данных не существует. В зависимости от задачи нужно выбирать либо массив, либо список, либо что-то еще.

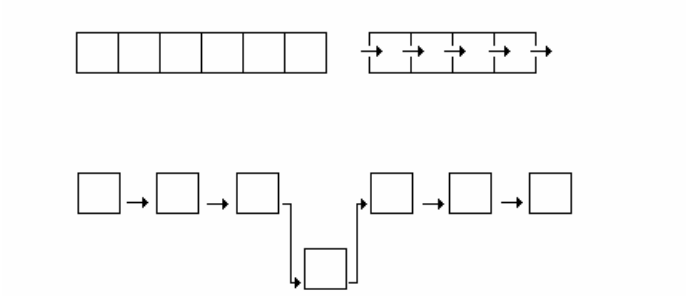

Рис. 8.4. Вставить элемент в связанный список легче, чем в массив

Typedef

Слово `typedef` не создает новые типы, как, например, `enum` или `struct`. Зато `typedef` позволяет по-другому назвать типы уже существующие. И это хорошо — прежде всего потому, что объявления переменных нужного типа становятся короче. В предыдущем пункте объявлялись записи и указатели на них. Получалось что-то очень длинное вроде `struct personage *repka` (см. листинг 8.15). Чтобы создать

новый тип, у которого будет одно короткое имя, предварим объявление записи словом typedef:

```
typedef struct personage {
    char *hero;
    struct personage *next;
} person;
```

Не будь слова typedef, эти строки создавали бы новую запись person типа struct personage, но typedef коренным образом меняет смысл объявления. Вместо записи, создается новый *тип* person, такой же, как int или double. С появлением слова typedef сама запись не создается. Это нужно делать явно: строка

```
person *repka, *root
```

заменяет

```
struct personage *repka, *root;
```

и теперь вместо

```
repka = malloc(sizeof (struct personage *))
```

можно написать

```
repka = malloc (sizeof(person *));
```

Задача 8.9. Перепишите программу, показанную в листинге 8 15, объявляя новые типы с помощью typedef.

Естественно, typedef позволяет создавать и другие новые типы. Чтобы понять, как это делается, полезно простое правило: дописывание typedef слева от любого объявления переменной, *возвышает ее до названия* нового типа. То есть, конкретная переменная, имеющая имя и занимающая память, после приписывания typedef превращается просто в имя нового типа.

Пусть, например, в программе объявлена переменная типа int:

```
int s32;
```

поставив слева `typedef`:

```
typedef int s32;
```

мы уничтожаем *переменную* `s32` и превращаем ее в имя нового *типа,* причем переменные нового типа будут такими же, какой была s32 *до* приписывания слова `typedef`. Новым типом можно пользоваться наравне с базовыми типами: `double`, `char` и т.п:

```
s32 a=0, i,j;
```

Теперь `a`, `i` , `j` — простые переменные типа `s32`. Все они такие же, какой была `s32`, то есть типа `int`.

Возьмем другое объявление:

```
int arr100[100];,
```

создающее массив из ста элементов типа `int`. Приписывание слева слова `typedef` уничтожает сам массив `arr100[100]` и создает новый *тип* `arr100`: массив из ста элементов типа `int`. Теперь переменные этого типа можно объявлять и использовать обычным образом:

```
typedef int arr100[100];

arr100 a;

int i;

for(i=0;i<100;i++)

      a[i]=0;
```

И последний пример. Объявление

```
int  (*f)(int, int);
```

задает указатель на функцию, принимающую две целочисленные переменные и возвращающую целое значение `int`. Приписывание слева слова `typedef` уничтожает указатель и создает новый *тип*: указатель на функцию и т.д.

Эти указатели объявляются так:

```
typedef int  (*f)(int, int);
```

```
f f1ptr, f2ptr;
```

Теперь `f1ptr` и `f2ptr` — указатели на функцию, принимающую два целочисленных значения и возвращающую целое число.

`Typedef` делает программу не только короче, но и мобильнее[1]. Дело в том, что базовые типы в языке Си определены довольно нечетко. Переменная типа `int` может занимать в разных компиляторах и 2, и 4, и даже 8 байт. Переменная `char` может быть как знаковой (`signed`), так и беззнаковой (unsigned). Естественно, программа, использующая переменную, которая занимает 8 байт, едва ли будет работать правильно на компьютере, где переменная `int` занимает 2 байта. Поэтому, перед тем как писать программу, разумно точно определить типы переменных. Если по смыслу задачи нужна знаковая переменная, занимающая четыре байта и в компиляторе, которым вы пользуетесь, этим требованиям отвечает тип `int`, то разумно определить новый тип `typedef int s32` в самом начале программы и всюду далее объявлять переменные как `s32 a,b,c`. Если нужно будет пропустить программу через другой компилятор, где знаковая переменная, занимающая 4 байта, имеет тип `short`, то практически вся программа останется неизменной. Нужно будет лишь поменять определение типа `s32`, написав вместо `typedef int s32;` `typedef short s32;`

[1] нет, она не станет похожей на сотовый телефон, просто сможет работать (после компиляции) на разных компьютерах.

Глава 9. Большие программы

Разделяй и властвуй

В этой книге мы имели дело лишь с очень короткими программами, которые помещались в одном файле. Но так бывает далеко не всегда. Программы вырабатываются из самого пластичного в мире материала — воздуха, и это дает возможность создавать невиданно сложные конструкции, по сравнению с которыми даже самые совершенные «материальные» объекты — ракеты, атомные реакторы, самолеты — кажутся детскими игрушками.

Тексты многих современных программ занимают миллионы строк, и энергия мысли, сохраненная в них, столь велика, что разнесла бы Землю на куски, найдись способ превратить ее в тротил или гексоген.

Естественно, программу, текст которой занимает даже сотни строк, не говоря о миллионах, невозможно хранить в одном файле. Когда размер программы достигает некоторого предела[1], приходится разбивать ее на части, каждая из которых хранится отдельно. Преимущества такого подхода очевидны: разработку и отладку разных частей программы можно поручить разным людям. Но теперь возникает необходимость взаимодействия отдельных ее частей (и людей тоже). Функция из одного файла, должна иметь доступ к переменной или функции из другого файла, при этом из соображений безопасности нельзя делать все переменные общедоступными. Возникает задача разграничения доступа: функция, хранящаяся в файле, должна «видеть» только то, что ее непосредственно касается.

---

[1] Этот предел очень индивидуален. Для кого-то он равен 20 строкам, а для кого-то — тремстам.

Вот об этом и пойдет речь в этой главе, и начнем с того, что вытащим из тьмы забвения программу сортировки, с которой уже имели дело в разделе «Массивы» главы 2. Программа, из листинга 2.6, сортирует 10 чисел, которые хранятся в массиве `dig[10]`. Настало время превратить ее в испытательный стенд для различных функций, выполняющих сортировку, а для этого разумно поместить ее в два файла (см. листинг 9.1): первый `main.c` хранит головную программу, которая создает массив нужного размера, заполненный случайными числами, вызывает функцию и выводит результаты сортировки на экран[1], а во втором файле `sort.c` помещается сама функция сортировки.

Листинг 9.1.

```
------------- файл main.c ---------------
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 1000
void sort(int *, int);
int main(){
long t1,t2;
int i,dig[N];
t1=time(NULL);
srand((unsigned int) t1);
for(i=0;i<N;i++)
    dig[i]=rand();
sort(dig,N);
```

[1] В этом файле есть две новых функции `time()` и `srand()`; о них поговорим в конце этого раздела.

```
t2=time(NULL);
printf("%d секунд\n",t2-t1);
for(i=0;i<N;i++)
    printf("%d \n", dig[i]);
return 0;
}
------------ конец файла main.c ----------
-------------- файл sort.c ---------------
void sort(int *dig, int N){
int i,j,mm,tmp;
for(i=N-1;i>=1;i--){
    mm=0;
    for(j=1;j<=i;j++)
        if(dig[j] > dig[mm])
            mm=j;
    tmp=dig[i];
    dig[i]=dig[mm];
    dig[mm]=tmp;
}
}
----------- конец файла sort.c ---------
```

Теперь у нас есть полный текст программы, размещенный в двух файлах, осталось только сообщить компилятору их имена. Для этого в папке с исходными текстами создается еще один файл с расширением .prj

```
--------------- файл sort.prj ---------------
main.c
sort.c
```

```
---------- конец файла sort.prj -------------
```

Осталось только вызвать Turbo C, выбрать в меню Project пункт Project name, загрузить файл sort.prj, затем выбрать пункт Make EXE file в меню Compile, и если в исходных текстах нет ошибок, возникнут два вспомогательных файла: main.obj и sort.obj (их еще называют *объектными*), из которых компилятор (вернее, та его часть, которая называется *редактором связей* или *компоновщиком*) создаст саму программу — файл с расширением .exe.

Эта последовательность действий (создание сначала *объектных* файлов, а затем самой программы) типична для многих компиляторов и сильно экономит время при разработке больших программ, потому что обрабатываются только *измененные* исходные тексты. Затем объектные файлы поступают редактору связей, и на выходе получается готовая программа.

Завершим этот раздел разбором новых стандартных функций `time()` и `srand()`, которые встретились в программе `main.c` (см. листинг 9.1). Функция `srand()` — спутник `rand()` и служит для того, чтобы сделать последовательность случайных чисел еще более случайной.

Дело в том, что функция `rand()` выдает всегда одни и те же числа, которые только выглядят случайными. Рано или поздно `rand()` будет вынуждена повториться, так что на самом деле о случайных числах можно говорить с некоторой натяжкой.

Задача 9.1. Напишите программу, которая определяет, через сколько чисел `rand()` начинает повторяться. Учтите, что период повторения огромен и его не уместить даже в переменной `long`.

Функция `srand()` указывает `rand()`, с какого места начать выдачу «случайных» чисел. Это место определяется аргументом `srand()`, который сам должен быть достаточно случаен. Например, это может быть время (в секундах, прошедших с 00.00.00 (по Гринвичу) 1 января 1970 года),

выдаваемое функцией `time()`[1]. Функция `time(NULL)` возвращает значение типа `long`, которое преобразуется к типу `unsigned int` (именно таков тип аргумента функции `srand()`).

Функция `time()` используется и для определения времени работы сортировки. Раз уж нам понадобилось значение времени непосредственно *до* сортировки, то разумно узнать время и *после*. Разность двух значений будет временем работы (в секундах) функции `sort()`.

Задача 9.2. Постройте график зависимости времени работы программы от количества сортируемых чисел. На какую функцию похожа эта зависимость?

Задача 9.3. Попробуйте сортировать те же числа с помощью стандартной функции qsort(), о которой говорилось в разделе «Qsort» главы 7. Постройте тот же график, что и в предыдущей задаче. На что он похож? Какая функция сортирует быстрее?

Extern или «Три поросенка»

В этом разделе будет создан программный комплекс, имитирующий поведение поросят в известной сказке. Будем считать, что поросята уже собрались в крепком каменном доме братца Наф-Нафа, и теперь они могут закрыть дверь, когда волк приближается, и открыть — когда он удаляется.

Программа, показанная в листинге 9.2, помещается в двух файлах — wolf.c и door.c.

Листинг 9.2.

```
-----------------файл wolf.c-----------------
  #include <stdio.h>
  #define IN  1
  #define OUT 0
```

[1] Функция `time()` описывается в заголовке `time.h`

```
   #define OPEN 1
   #define CLOSED 0
   int wolf=OUT;
   int main(){
   int door(void);
   if (door()==OPEN)
       printf("Дверь открыта\n");
   if (door()==CLOSED)
       printf("Дверь закрыта\n");
   retutn 0;
   }
-----------конец файла wolf.c----------------
-------------- файл door.c-------------------
   extern int wolf;
   #define IN  1
   #define OUT 0
   #define OPEN 1
   #define CLOSED 0
   int door(){
   if(wolf==IN)
       return CLOSED;
   if(wolf==OUT)
       return OPEN;
   }
-------------конец файла door.c -------------
```

В первом файле — wolf.c — вызывается функция `door()` и, в зависимости от возвращенного ей значения, на экран

выводится «Дверь открыта» или «Дверь закрыта». Во втором файле — door.c — помещается сама функция `door()`, которая возвращает значение `OPEN`, если переменная `wolf` равна `OUT` или `CLOSED`, если `wolf` равна `IN`.

Но как функция `door()` узнает, чему равна переменная `wolf`? Сложность здесь в том, что `wolf` определена в файле wolf.c и там же ей присваивается значение `int wolf=OUT`. А ее значение необходимо знать функции, расположенной в *другом* файле. Поэтому в файле door.c нужно как-то указать, что переменная `wolf` существует и определена в другой части программы. Делается это с помощью слова `extern` (в переводе с английского external — «внешний»). Строка

```
extern int wolf;
```

говорит компилятору, работающему с файлом door.c: «Пожалуйста, не задавай глупых вопросов насчет переменной `wolf`. Она определена в другом файле. Каком – не твоего ума дело, достаточно знать, что она — типа `int`».

Увидев слово `extern`, компилятор успокаивается и создает объектный файл, в котором сказано, что переменная `wolf` — внешняя, определена в каком-то другом файле, и ее поисками займется уже редактор связей.

Нужно отчетливо понимать, что жизнь дается переменной *один* раз. В нашем случае это делается в файле wolf.c, где объявление `int wolf=OUT` велит компилятору выделить участок памяти для целочисленной переменной и заслать туда ее начальное значение. Все остальные упоминания, например, `extern int wolf` лишь успокаивают компилятор, сообщая ему, что есть-де переменная, определенная в другом файле, а в каком — неважно. Слово `extern` ничего не создает и памяти никакой не выделяет.

Обратите внимание, словом `extern` необходимо помечать только *переменные*. Для функций это излишне, потому что компилятор и так считает все функции внешними. Но чтобы подчеркнуть, что функция определена в другом файле, ее прототип можно предварить словом `extern`:

```
extern int door(void);
```

Перед тем как перейти к следующему разделу, полезно взглянуть еще раз на листинг 9.2. Два файла, показанные на нем, содержат один и тот же фрагмент, определяющий имена IN, OUT, OPEN, CLOSE. Чтобы не путаться и не писать несколько раз одно и то же, полезно составить *заголовочный файл* (так часто называют файл с расширением .h, помещаемый в начало исходного текста программы), куда и поместить общие для всех программ определения:

```
--------------- файл  pigs.h ----------------
   #define IN  1
   #define OUT 0
   #define OPEN 1
   #define CLOSED 0
-------------- конец файла pigs.h -----------
```

Заголовочный файл располагается в папке с исходными текстами. Теперь вместо определений имен, нужно включить в каждый файл строчку

```
#include "pigs.h",
```

где имя файла pigs.h обрамляется двойными кавычками "…", а не скобками <…>. Увидев двойные кавычки, компилятор поймет, что перед ним нестандартный заголовочный файл, который нужно искать там же, где и файлы с исходными текстами.

Static

До сих пор мы почти не обращали внимания на место, где объявляются переменные. И (особенно тем, кто сам еще не пробовал программировать) кажется, что это можно делать где угодно. На самом деле, объявлять переменную нужно *до* ее использования. Если, например, нам вздумается объявить переменную wolf в самом конце файла wolf.c (о нем говорилось в предыдущем разделе):

```
#include <stdio.h>
#include "pigs.h"
int main(){
…
}
int wolf=OUT;
```

то программа, составленная из двух файлов wolf.c и door.c, будет работать, как и прежде. Но если нам захочется вывести на экран значение переменной `wolf`:

```
#include <stdio.h>
#include "pigs.h"
int main(){
printf("%d\n", wolf);
…
}
int wolf=OUT;
```

то компилятор выдаст следующее сообщение об ошибке:

```
Undefined symbol 'wolf' in function main,
```

потому что он ничего еще не знает об этой переменной. Помочь делу можно двумя способами: либо объявить переменную до ее использования, либо с помощью слова «extern» дать понять компилятору, что переменная будет объявлена позже:

```
#include <stdio.h>
#include "pigs.h"
int main(){
extern int wolf;
printf("%d\n", wolf);
…
```

```
}
int wolf=OUT;
```

Чтобы показать, как важно правильно выбрать место объявления переменной, попробуем поместить ее внутрь функции `main()`:

```
/* int wolf=OUT; - старое место */
main(){
int wolf=OUT; /* новое место */
…
}
```

Теперь при попытке скомпилировать программу `Turbo C` выдаст следующее сообщение об ошибке:

```
Linker Error: Undefined symbol '_wolf' in module DOOR.C,
```

означающее, что компоновщик (`Linker`) не смог найти внешнюю переменную `wolf` (о том, что переменная внешняя, говорит поставленная перед ней черта _).

Но как же так — спросите вы — ведь вот же она — внутри функции `main()`! Все дело в том, что переменные, попавшие *внутрь* функции, становятся невидимы всему остальному миру. И нет лучше укрытия для виртуальных поросят, чем функция:

```
void fortress(){
double nif_nif;
double nuf_nuf;
double naf_naf;
…
}
```

Только что объявленные переменные `nif_nif, nuf_nuf, naf_naf`; видны лишь внутри функции `fortress()` и у

других функций нет никакой возможности получить к ним доступ. И это еще не все. Оказывается, переменные могут спрятаться и внутри самой функции. Для этого им достаточно окружить себя фигурными скобками:

```
void field(){
double wolf;
double dog;
double lion;
double hunter;
/* убежище для поросят */
{
   float nif_nif;
   float nuf_nuf;
   float naf_naf;
}
…
}
```

Переменные `nif_nif`, `nuf_nuf`, `naf_naf` существуют только внутри фигурных скобок и только там можно присвоить им значения, вывести на экран или использовать как-то иначе. Стоит выйти за скобки, и переменные пропадают без следа.

Итак, то, что объявлено вне функций, доступно всюду *после* объявления. Объявленное внутри функций, доступно или во всей функции или в блоке, окруженном фигурными скобками.

Причем, заметьте, что переменные, объявленные внутри функции, видны не только в определенном месте, но и в определенный промежуток времени: от момента входа в функцию до момента выхода из нее.

Попробуем проследить за переменной внутри функции:

```
void trap(){
int i;    /* содержит ґмусорЄ */
printf("%d\n", i); /* i=????  */
i=1;
printf("%d\n", i); /* выводит единицу */
…
}
```

Объявление `int i` лишь отводит место для переменной. Никакого определенного значения она не получает, и при вызове функции `trap()` содержит «мусор». Поэтому первая функция `printf("%d\n", i)` может вывести на экран любое число, например, 435, -7, 832. Присваивание `i=1` вносит в жизнь переменной `i` какую-то определенность. Теперь она равна единице, что и покажет вторая функция `printf()`. Дальнейшие приключения переменной `i` нам не интересны, но важно понимать, что после выхода из функции память о ней теряется и при повторном входе в `trap()` — `i` снова будет содержать «мусор».

Чтобы переменная помнила о себе и по выходе из функции, необходимо объявить ее со словом `static`:

```
void trap(){
static int i;    /* i=0 */
printf("%d\n", i); /* выводит 0  */
i=1;
printf("%d\n", i); /* выводит единицу */
}
```

Теперь при первом вызове `trap()` `i` будет содержать не «мусор», а ноль, который и выведет на экран функция `printf()`. Следующая функция `printf()` выведет, как и в предыдущем случае, единицу, но теперь эта единица будет запомнена, ведь переменная, объявленная как `static`, после выхода из функции не умирает и помнит себя. При повторном

вызове `trap()` первая функция `printf()` выведет уже не ноль, а единицу — последнее значение, которое имела `i` перед выходом из функции.

На листинге 9.3 показана программа, в которой используются свойства статичных[1] переменных. Функция `bang()` вызывается в ней 100 раз, но благодаря статичным переменным `impatience` и `counter` функция `bang()` помнит свое прежнее состояние. Когда число ее вызовов не превышает трех, функция выводит на экран номер вызова. При четвертом вызове функция сообщает нам, что «ее терпение лопнуло» и при следующих вызовах хранит молчание.

Листинг 9.3.

```
#include <stdio.h>
void bang(void);
int main(){
int i;
for(i=0;i<100;i++)
   bang();
return 0;
}
void bang(){
static int impatience;
static int counter=1;
```

[1] Переменные, которые объявлены как «static», разумно называть *статичными,* потому что они сохраняют свое значение между последовательными вызовами функции. Тогда обычные переменные, которые возникают при входе в функцию и исчезают после выхода из нее, можно назвать *динамическими*.

```
if(impatience) return;
printf("Меня беспокоят %d раз\n",counter);
counter+=1;
if(counter > 3){
    impatience = 1;
    printf("Терпение лопнуло\n");
}
}
```

Задача 9.4. Напишите программу, правильно говорящую по-русски: «меня беспокоят в первый, второй, третий, пятый, шестнадцатый, тридцать первый… и т.д. раз».

В этом разделе мы пока говорили только о применении слова `static` к переменным, которые объявляются внутри функции. Но оказывается, слово `static` можно применить и к внешним (объявленным вне функций) переменным. Когда внешняя переменная предваряется словом «static»:

```
static int unvisible_pig;
```

то становится видимой только в своем файле. Переменная `wolf`, объявленная как `static int` доступна только в файле wolf.c (см. листинг 9.4) и скрывается от функции `door()`, расположенной в другом файле.

Во всем же остальном статичная переменная похожа на обычную переменную, объявленную вне функции: она с самого начала равна нулю и «помнит» свое последнее значение.

Листинг 9.4.

```
-----------------файл wolf.c-----------------
#include <stdio.h>
#include "pigs.h"
static int wolf=OUT;
```

```
   main(){
   if (door()==OPEN){


   …
   }
-----------конец файла wolf.c----------------
-------------- файл door.c-------------------
   extern int wolf;
   #include "pigs.h"
   int door(){
   …
   }
-------------конец файла door.c -------------
```

В предыдущем разделе мы уже говорили о том, что функции в языке Си видны отовсюду[1]. Единственная возможность скрыть их — предварить словом `static`.

Функция

```
static int door(){}
```

видна *только в своем файле*. Если она еще и нужна только в этом файле, то остальным функциям незачем знать о ней, и слово `static`, устраняя возможное влияние переменной из одного файла на другие файлы, облегчает отладку и модификацию программы, делает ее проще и надежнее.

---

[1] Правда, *прототипы* функций видны только в пределах файла. Поэтому их (вместе со значениями констант) нужно хранить в специальном заголовочном файле вроде `pigs.h` (см. предыдущий раздел), который должен подключаться всюду, где эти константы и прототипы используются.

Глава 10. Макросы и переходы

Макросы

Мы долго плыли в декорациях моря, но вот они — фанера и клей.

БГ, «Тень»

Любая программа на Си, прежде чем ее допустят к компилятору, подвергается санитарной обработке в *препроцессоре*, который похож на текстовый редактор, управляемый специальными командами.

Каждая команда препроцессора начинается со значка #, и мы уже хорошо знакомы с двумя из них: `#include <файл>` (вставить содержимое файла в текст программы) и `#define ИМЯ1 ИМЯ2` (всюду в тексте программы заменить «ИМЯ1» на «ИМЯ2»). Определение имени с помощью команды `#define` называется в языке Си *макросом*, и с его помощью можно не только присваивать имена константам, но и создавать подобия функций.

Попробуем, например, создать макрос, возводящий произвольное число в квадрат. Выглядеть он может так:

```
#define SQR(x) x*x
```

Ключевой элемент этого макроса — скобки. Не будь их, макрос

```
#define SQRx x*x
```

просто заменял бы в тексте программы символы `SQRx` на `x*x`. Но открывающая скобка, *непосредственно* следующая за именем `SQR`, говорит препроцессору, что перед ним макрос с *параметрами*, заключенными в круглые скобки. В нашем случае параметр один, это `x`, и встретив в тексте программы

символы `SQR(a)`, препроцессор поставит вместо них уже `a*a`, а не `x*x`.

Программа из листинга 10.1, вычисляет «дважды два» и выводит на экран четверку. Мы видим в ней присваивание `s=SQR(i)`. Но компилятор на его месте увидит `s=i*i,`потому что получит текст, прошедший через препроцессор.

Листинг 10.1.

```
#include <stdio.h>
#define SQR(x) x*x
int main(){
int i=2,s;
s=SQR(i);
printf("%d\n",s);
return 0;
}
```

Казалось бы, макрос с параметрами полностью заменяет функцию, ведь строчка `s=SQR(i)`ничем по виду не отличается от вызова функции и присваивания возвращенного значения переменной `s`. Но стоит немножко по-другому «вызвать» макрос, и «вот они — фанера и клей».

Программа, показанная в листинге 10.2, должна, по идее, выводить на экран 0.25, то есть, единицу, поделенную на 4.

Листинг 10.2.

```
#include <stdio.h>
#define SQR(x) x*x
int main(){
float i=2.0,s;
s=1/SQR(i);
```

```
printf("%g\n",s);
return 0;
}
```

Но препроцессор понимает макрос *буквально*: каждый формальный параметр будет заменен фактическим, и компилятор на месте `s=1/SQR(i)` увидит `s=1/i*i.` А дальше он поделит единицу на `i`, а результат деления умножит на `i`, ведь приоритет умножения и деления одинаков, а выполняются они слева направо. В результате 0.5 (результат деления) умножится на 2, и получится единица!

Чтобы получить правильный результат, достаточно окружить скобками произведение переменных: `s=1/(i*i)`, а для этого можно просто переопределить макрос, поместив `x*x` в скобки:

```
#define SQR(x) (x*x)
```

Но представим теперь, что в квадрат возводится сумма двух чисел:

```
s=SQR(2+2);
```

Как обычно, препроцессор не станет умничать, а тупо заменит формальные параметры макроса фактическими. И после его работы компилятор увидит:

```
s=(2+2*2+2);
```

Умножение, как мы знаем, обладает большим приоритетом, поэтому в результате получим 8:

```
s=2 + (2*2) + 2=2 + 4 + 2 = 8;
```,

а не 16, как нам бы хотелось. Выход из этого затруднения прост: нужно окружить оба параметра скобками, чтобы они сначала вычислялись, а только потом умножались друг на друга:

```
#define SQR(x) ((x)*(x))
```

Такой макрос правильно вычислит `s=SQR(2+2)`, потому что препроцессор преобразует `SQR(2+2)` в `((2+2)*(2+2))`. Но представим теперь, что в программе встретились строчки:

```
i=2;
s=SQR(i++);
```

Если `SQR()` — функция, все будет хорошо, и после ее вызова `s` будет равна четырем, а `i` — трем. Но если `SQR()` — макрос, то `i` и `s` могут оказаться *какими угодно*, потому что препроцессор превратит `SQR(i)` в `((i++)*(i++))` — выражение, значение которого не определено (см. раздел «Действия» главы 4). И из этого последнего затруднения есть, похоже, один выход — просто не использовать оператор ++ и ему подобные в таких макросах.

Как мы уже поняли, макрос с параметрами — это муляж функции, наскоро слепленный из подручных материалов. Но как это обычно бывает, в определенных условиях недостатки оборачиваются достоинствами. Функции безусловно «правильней», но требуют от процессора дополнительных затрат на передачу параметров, вызов и возврат. Если функция часто вызывается, эти затраты (т.е. дополнительное время) могут быть велики и тогда полезно заменить функцию макросом.

Следующий недостаток макроса, который при определенных условиях может превратится в его полезное свойство, — отсутствие проверки типов аргументов. Функция, возводящая число в квадрат, принимает аргумент определенного типа. Для параметров типа `int` придется писать одну функцию, для `double` — другую. Но макрос, возводящий в квадрат, ничего не знает о типе параметра и его можно использовать для *любых* переменных.

Можно, например, написать макрос, обменивающий значения переменных любого типа. Обычно это делается с помощью буферной переменной:

```
tmp=a;
a=b;
b=tmp;
```

Если использовать этот метод в макросе, придется как-то указывать тип переменной tmp.

Задача 10.1. Напишите макрос, который меняет значения двух переменных `a` и `b` (`a` присваивается значение `b`, `a` `b` — значение `a`) с помощью промежуточной переменной . У этого макроса будет три параметра: тип , первое число, второе число. Например, `SWAP(int,a,b);`

Но можно построить макрос, который не использует промежуточную переменную:

```
#define SWAP(a,b) a=a-b; b=b+a; a=b-a ,
```

а значит, пригоден для обмена переменных любого типа.

Задача 10.2. Какие недостатки у такого макроса?

Последний макрос довольно длинен и, возможно, лучше будет смотреться, если займет несколько строк. В этом случае препроцессору, дошедшему до конца строки, нужно сообщить, что макрос еще не кончился. Для этого там ставится обратная косая черта '\':

```
#define    SWAP(a,b)\
           a=a-b;\
           b=b+a;\
           a=b-a
```

Всюду в этом разделе мы говорили о создании макросов. Но препроцессор умеет не только создавать. Командой #undef ему под силу уничтожить любой макрос. Строки

```
#define OUT 0
wolf=OUT;
#undef  OUT
wolf=OUT;
```

пройдя препроцессор, превратятся в

```
wolf=0;
wolf=OUT;
```

потому что перед вторым присваиванием `OUT` перестанет быть именем макроса. Быть может, до сих пор было не совсем

понятно, с какого момента макрос начинает жить, а когда — заканчивает. Теперь ясно, что макрос живет в пределах одного файла — с момента определения и до соответствующей команды `#undef`.

Задача 10.3. Попробуйте (на основании экспериментов) понять, как поведет себя препроцессор, когда встретит макрос, вызывающий сам себя:

```
#define rabbit (4 + rabbit)
```

Обычно препроцессор, встретив макрос и сделав текстовую замену, проверяет, а нет ли в новом тексте других макросов. Но если макрос ссылается сам на себя, препроцессор, чтобы не впасть в бесконечный цикл, должен поступить иначе.

Управление текстом

В предыдущем разделе мы познакомились с возможностями препроцессора менять один текст на другой. Эта замена может быть простой, когда одни буквы встают на место других, или более сложной, когда в окончательный текст вставляются переданные макросу аргументы.

Но ясно, что этим препроцессор ограничиться не может. Неумолимая логика ведет его к тому, что вслед за возможностью менять что-то в тексте, должна появиться возможность менять сам текст. Часто, например, программисту, который занимается отладкой программы, необходимо вывести на экран какие-то переменные. Для этого можно вставить в текст программы несколько функций `printf()`. Но когда отладка закончится, придется искать все эти функции в тексте и удалять их. Чтобы не делать этого, окружим вызовы `printf()` командами препроцессора:

```
#define DEBUG
...
#if defined (DEBUG)
printf("%d\n",i);
#endif
```

и теперь вызов `printf()`, окажется в тексте программы только в том случае, если *определен* макрос `DEBUG`.

Команды препроцессора, обрамляющие `printf()`, похожи на инструкцию `if(){}` самого языка Си. Если условие после инструкции `#if` выполняется, в текст программы вставляются строки, расположенные между `#if` и `#endif`. Само условие проверяется специальным оператором `defined (имя_макроса)`. Если имя_макроса (в нашем случае это DEBUG) определено, оператор `defined()` возвратит единицу, условие выполнится, и строки попадут в текст программы. Если `имя_макроса` не определено (например, потому, что после команды `#define DEBUG` следует команда `#undef DEBUG`), оператор `defined()` возвратит ноль, и помещенный между `#if` и `#endif` текст не дойдет до компилятора.

Кроме включения или отключения каких-то строк текста, часто бывает нужно включить одни строки, если условие выполняется, и другие — если не выполняется. В этом случае применяется команда `#else`.

До сих пор компилятор был для нас запретной территорией, куда любопытно, но боязно заглянуть. Наверное, внутренности заголовочных файлов способны сильно испугать начинающего программиста. Но нам уже известно столь многое, что заголовочные файлы уже не пугают. Наоборот, они могут стать ценнейшим справочным пособием, в котором есть сведения обо всех стандартных функциях и макросах.

Посмотрим, например, определение макроса `NULL` (листинг 10.3).

Листинг 10.3

```
#ifndef NULL
#if defined(__TINY__)  ||\
     defined(__SMALL__) ||\
     defined(__MEDIUM__)
```

```
#define NULL    0
#else
#define NULL    0L
#endif
#endif
```

Самая внешняя пара команд в определении макроса — `#ifndef NULL...#endif` говорит препроцессору, что заниматься макросом нужно лишь в том случае, когда он еще не определен (команда `#ifndef` — это сокращенная форма `#if !defined(NULL)`[1]. Такая «обертка» нужна, чтобы проверить, не был ли определен макрос в каком-то другом файле .

Само определение `NULL` содержит два варианта: если определен один из макросов `__TINY__`, `__SMALL__`, или `__MEDIUM__`, то `NULL` определяется как простой `0`, то есть константа типа `int`. Если же эти макросы не определены, то `NULL` определяется как `0L`, то есть константа типа `long int`. Макросы, которые начинаются с нижней черты, например `__TINY__` обычно определяются самим компилятором. В нашем случае они указывают, какая модель памяти используется. Вдаваться в детали мы не будем, скажем лишь, что в зависимости от модели памяти размер указателя меняется. При моделях памяти `tiny`, `small` и `medium` указатель, создаваемый компилятором Turbo C (другие компиляторы создают указатели других размеров и могут ничего не знать о моделях памяти) занимает два байта, в противном случае — четыре. Вот почему `NULL` определяется в одном случае как `0`, а в другом — как `0L`.

---

[1] перед `defined` стоит `!` — оператор логического отрицания, о котором говорилось в разделе «Условности» главы 4

[1] Обратите внимание, в инструкции goto метка; после имени метки двоеточие не ставится.

Напутствие или GOTO…

Дорогою свободной иди, куда влечет тебя свободный ум, усовершенствуя плоды любимых дум, не требуя наград за подвиг благородный.

А.С.Пушкин, «Поэту»

В названии `goto` слились два английских слова «go» и «to», что в переводе на русский означает «идти». И нет ничего удивительного в том, что `goto` — инструкция, которая позволяет попасть в любое указанное место программы.

Например, с помощью `goto` можно построить самодельный цикл `while()`, заменив

```
while(x > 0){
…
}
```

чуть более сложной комбинацией условий и переходов:

```
nxt:
if(x <= 0) goto endwhile;
…
goto nxt;
endwhile:
```

В этом фрагменте сначала проверяется условие `if(x <= 0)`. Если оно *не* выполняется, то значит, `x > 0` и программа переходит к инструкциям, помещенным внутрь цикла. Пройдя их, она натыкается на инструкцию `goto nxt`[1] и переходит к метке `nxt:`, где снова проверяется условие `if(x < = 0)`. Если оно выполняется, самодельный цикл совершает следующий оборот, если нет — идем к метке `endwhile:`, то есть к инструкциям, следующим непосредственно за циклом.

Как видим, проверка условий и пара инструкций `goto` вполне способны заменить цикл `while()`, да и другие циклы тоже.

Задача 10.4. С помощью инструкций `goto метка` и проверки условия `if()` постройте циклы `do{} while()`и `for(){}`.

Только вот зачем? Ясно, что цикл `while()` выглядит лучше любой своей имитации. То же самое можно сказать и о других конструкциях языка. Существует даже строгое доказательство того, что любую программу можно написать, не используя `goto` — только с помощью тех инструкций, которые нам уже известны.

Более того, переход `goto` во многих случаях не только не улучшает программу, но и ощутимо портит ее, делая громоздкой и малопонятной. Посмотрим, например, как неуместное использование инструкций перехода уродует функцию, определяющую наименьшее из трех чисел (листинг 10.4)

Листинг 10.4.

```
int min(int a, int b, int c){
if (a < b) goto alb;
if (b < c) goto blc;
return c;
alb: if (a < c) return a;
return c;
blc: return b;
}
```

Хоть функция `min()`и верна, логику ее работы довольно трудно проследить просто потому, что сами переходы `goto` к логике никакого отношения не имеют. Ища глазами метку, мы забываем об условии, а, найдя метку, должны вернуться назад, к инструкции `goto`. Гораздо понятнее и короче выглядит вариант функции, где `goto` не используется:

```
int min(int a, int b, int c){
int min=a;
```

```
if (b < min) min = b;
if (c < min) min = c;
return min;
}
```

Раз инструкция `goto` приносит только вред и без нее всегда можно обойтись, зачем включать ее в язык программирования и вообще говорить о ней? Затем, что иногда она все-таки оказывается полезной. Рассмотрим, например, фрагмент программы, в котором используются инструкции перехода:

```
nxts:
while(){
    ...
    goto nfnd;
    ...
    goto nfnd;
    ...
}
strcd=nxtpnt;
goto nxts;

nfnd:
/* Здесь много инструкций */
...
goto nxts;
```

В нем из цикла `while()` программа дважды переходит к инструкциям, помещенных между меткой `nfnd:` и переходом `goto nxts`. И первое, что приходит в голову, — оформить эти инструкции в виде макроса. Но макрос, содержащий множество инструкций, некрасив и неудобен в отладке. Кроме

того, в нем все равно должны быть инструкции `goto`, иначе не перейти к метке `nxts`. Значит, макрос в данном случае мало чем помогает, он не устраняет, а лишь скрывает недостатки программы, «заметает сор под лавку».

Вторая идея — заменить инструкции между меткой `nfnd` и `goto nxts` функцией — тоже не слишком удачна. Ведь функция возвращается к инструкции, непосредственно за ней следующей. А программа устроена так, что нужно вернуться к метке `nxts`! Значит, выбираться из цикла `while()`, придется с помощью инструкции `break`[1], поставленной после вызова функции. Но и это не спасет, потому что вместо перехода к метке `nxts:` мы (после выхода из `while()`) наткнемся на инструкцию `strcd=nxtpnt.` Для ее обхода понадобится специальная переменная, которая покажет, каким образом завершился цикл: естественным, то есть когда перестало выполняться условие в круглых скобках `while(условие)` или же после инструкции `break.` «Исправленная» с учетом всего этого программа будет выглядеть примерно так:

```
while(1){
    while(){
        flag=0;
        …
        newfun();
        flag=1;
        break;
    …
        newfun();
        flag=1;
        break;
```

[1] а `break`, если честно, — все та же инструкция `goto`, только иначе записанная

```
            ...
        }
        if(flag == 0)
            strcd=nxtpnt;
    }
```

В ней действительно нет инструкций `goto`, но стала ли она от этого лучше? Ответ зависит от того, насколько легко и естественно преобразуются в функцию инструкции, стоящие в первом варианте программы между меткой `nfnd:` и переходом `goto nxts.` Если они выполняют разные задачи, то их объединение выглядит некрасиво и неестественно. А если вновь созданной функции придется еще передавать кучу аргументов, то руки опускаются, и программист решает оставить все как есть. И правильно делает.

Впрочем, найдутся люди, которые будут категорически против любого использования `goto`, полагая, что стоит только дать программисту волю, и он изуродует тексты программ беспорядочными переходами.

Мне кажется, что в таком трудном деле как создание программ, не стоит заранее сковывать себя какими-то догмами. Жванецкий говорит: «когда я пишу, то на бумагу делаю все» и надо ему в этом подражать. Быть может, заранее не стоит даже думать о конкретных циклах, а написать что-то с использованием `goto`, если так привычней. Взглянув потом на текст программы, легко можно будет разглядеть в нем циклы `while()` или `for()`, только иначе записанные. Не нужно бояться повторять какие-то куски кода или даже писать инструкцию за инструкцией, как будто не существует циклов. Записав несколько инструкций подряд, легче сообразить, какому циклу они соответствуют.

Существует, пожалуй, единственный абсолютный запрет: нельзя преждевременно бросать работу над программой. Нужно добиваться от нее простоты и ясной структуры, нужно следить, не повторяются ли в ней какие-то инструкции и если да — превратить их в функции или макросы. Следует избегать

ненужных переходов `goto`, но здесь помогут не запреты, а опыт и чувство стиля.

Приложение А

Приоритеты и порядок вычислений операторов

Операторы	Выполняются
`() [] -> .`	Слева направо
`! ~ ++ -- + - * &` (тип) `sizeof`	Справа налево
`* / %`	Слева направо
`+ -`	Слева направо
`<< >>`	Слева направо
`< <= > >=`	Слева направо
`== !=`	Слева направо
`&`	Слева направо
`^`	Слева направо
`|`	Слева направо
`&&`	Слева направо
`||`	Слева направо
`?:`	Справа налево
`= += -= *= /= %= &= ^= |= <<= >>=`	Справа налево
`,`	Слева направо

Приоритет унарных операторов +, - и * выше, чем соответствующих бинарных.

Что дальше?

Здесь не делают домашних работ. Мы поможем предложениями и советами, но сначала вы должны написать что-то сами. Никто не хочет быть невежливым, но большинство из нас считают, что делать чью-то домашнюю работу — жестоко, потому что не дает правильно учиться. Программированием, как и любым другим ремеслом надо заниматься. Мы уже умеем делать домашние задания, по крайней мере, не хуже вас и поэтому в таких занятиях не нуждаемся.

Из конференции comp.lang.c.moderated

В Предисловии я написал, что хочу видеть свою книгу одной из первых в длинном списке книг по программированию и алгоритмам. И раз вы читаете эти строки, мое желание сбылось. Хочу поздравить терпеливого читателя, который стал пусть неопытным, пусть начинающим, но *программистом*. Перед ним длинная и увлекательная дорога, предсказать все многочисленные повороты которой невозможно.

Но хотелось бы пожелать, чтобы следующий шаг был связан с «лучшей второй книгой всех времен и народов по языку Си» — классической «Язык программирования Си» Брайана Кернигана и Денниса Ритчи.

Если бы я начинал изучение языка Си сейчас, то лучшей (по крайней мере, для меня) была бы моя собственная книжка. К сожалению, мне, как и многим другим, приходилось учиться по книге Кернигана и Ритчи. И надо честно сказать, что многое в ней давалось мучительно, а кое-что (рекурсию, например) я так тогда и не понял. Поэтому в своей книге я постарался максимально подробно рассказать о самом трудном. Надеюсь, что теперь чтение K&R (так во всем мире называют книгу Кернигана и Ритчи) будет для вас не только поучительным, но также легким и приятным.

Изучение Си, как и любого другого языка программирования, немыслимо без упражнений. Чтобы научиться программировать, нужно программировать. Задачи, которые

вы будете решать на первых порах, не должны быть слишком сложными. Старайтесь начинать с самого главного, постепенно подключая все новые и новые возможности. Разбивайте задачу на части, которые можно отладить отдельно друг от друга. Читайте документацию. Не поддавайтесь панике. И тогда к вам придет удача.

Литература

1. Б.Керниган, Д. Ритчи «Язык программирования Си». Москва «Финансы и статистика», 1992
2. Ч.Уэзерелл «Этюды для программистов». М. Мир, 1982
3. Б.Керниган, Ф.Плоджер «Элементы стиля программирования», М. «Радио и связь», 1984
4. Ф.П.Брукс мл. «Как проектируются и создаются программные комплексы», М. «Наука», 1979
5. Ч. Петзолд «Программирование для Windows 95 в двух томах», BHV, 1996

Об авторе

Крупник Александр Борисович, родился 26 июня 1957 года в г. Горьком. Окончил Радиофизический факультет Горьковского государственного университета, канд. ф.-мат. наук.