ЯЗЫК ПРОГРАММИРОВАНИЯ

Б. Керниган
Д. Ритчи

ЗАДАЧИ ПО ЯЗЫКУ

А. Фьюэр

Brian W. Kernighan
Dennis M. Ritchie

THE C PROGRAMMING LANGUAGE

PUZZLE BOOK

Allan R. Feuer

BELL
LABORATORIES
MURRAY HILL, NEW JERSEY

PRENTICE–HALL, INC.,
ENGLEWOOD CLIFFS

Б. Керниган
Д. Ритчи

ЯЗЫК ПРОГРАММИРОВАНИЯ Си

ЗАДАЧИ ПО ЯЗЫКУ

А. Фьюэр

Перевод с английского
Д.Б.Подшивалова
и В.А.Иващенко

МОСКВА
"ФИНАНСЫ И СТАТИСТИКА"
1985

ББК 32.973
К36

Керниган Б., Ритчи Д., Фьюэр А.

К 36 Язык программирования Си. Задачи по языку Си/ Пер. с англ. Д. Б. Подшивалова и В. А. Иващенко.— М.: Финансы и статистика, 1985.—279 с., ил.

В пер.: 1 р. 50 к. 15 000 экз.

Книга представляет собой пособие по новому для советского читателя языку программирования Си и содержит его строгое определение. Язык Си позволяет максимально использовать ресурсы ЭВМ и практически полностью отказаться от программирования на языке Ассемблера.

Для программистов, системных программистов и других специалистов, интересующихся проблемами программирования.

К $\frac{2405000000—046}{010(01)—85}$ 122—85

ББК 32. 973
6Ф7. 3

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Книга, которую читатель держит в руках, посвящена описанию языка профессионального программирования Си.

Время от времени появляются сообщения, что в той или иной группе разработчиков программного обеспечения успешно используется некий локальный язык программирования. Обычно подобные локальные языки программирования в определенной степени отражают требования высокого профессионализма. В зависимости от ориентации группы ее язык, как правило, обладает некоторыми гипертрофированно развитыми чертами. Например, в язык включаются специфически «трансляторные» возможности, скажем, возможности построения лексических или синтаксических анализаторов. Так как группы часто «мигрируют» с машины на машину, желание сохранить привычную для себя среду приводит к тому, что язык начинает поддерживать стремление переносить с машины на машину если не все системы программ или отдельные программы, то хотя бы отдельные фрагменты.

Языку Си повезло, что он оказался «встроенным» в операционную систему UNIX (начинающую в последние годы получать широкое распространение), причем в основе самой системы оказался тот же язык Си: многие фрагменты ее были написаны на нём. Между системой и языком существует некоторая внутренняя связь, взаимопроникновение. Например, работа со внешними устройствами, источниками информации, архивами оказывается простой именно из-за аналогичной простоты соответствующих действий в системе UNIX. Точнее, сами границы между языком (в традиционном понимании этого слова) и системой очень размыты: в языке есть только обращения к подпрограммам, но зато в системе предусмотрено много разных подпрограмм. Это делает язык (точнее, систему программирования) очень гибким, но требует хорошего знания системных подпрограмм и самой системы. К сожалению, надо отметить, что литература о системе UNIX все еще довольно бедна, что, возможно, влияет на распространение языка Си за пределы этой системы.

В момент, когда решался вопрос о переводе книги, она была практически единственным источником сведений о языке, если не считать нескольких обзорных статей тех же авторов и руководств по работе в системе UNIX, куда включалось описание языка, приведенное в приложении к данному изданию. Таким образом, книга Кернигана и Ритчи — авторское введение в язык, построенное так, как они хотели. И язык в нем описывается так, как они желали. Почему об этом надо специально упоминать? Дело в том, что сложился определенный шаблон для составления описаний языков программирования, определенная традиция. Авторы не всегда этому шаблону следуют, они не столько учат самому языку, тому как он устроен, сколько тому, как надо на нем программировать.

Специалисты, хорошо знакомые с программированием, в своем стремлении быстро познакомиться с самим языком сразу же наталкиваются на определенные трудности. Например, приводимое в приложении описание синтаксиса содержит некоторые неточности (многие даже считают их ошибками). Однако у авторов нигде нет утверждения, что это абсолютно точное описание синтаксиса, наоборот, они говорят о его вспомогательном характере. При желании подобного рода «неточностей» в книге можно обнаружить много. Мы при переводе видели эти «огрехи», но не сочли возможным или необходимым их исправлять. Во-первых, речь идет о переводе вполне конкретной книги, где допустимо исправление только опечаток, а не авторских «неточностей». Во-вторых, авторы описывают очень подвижную, «живую» систему и язык, стремясь скорее донести до читателя саму идею, лежащую в их основе, а не сформулировать строгие правила. Именно поэтому, как нам кажется, у них и нет строгой терминологии, столь характерной для традиционных описаний языков программирования, и нет строгого описания синтаксиса, да и семантика в некоторых достаточно «туманных» местах столь же «туманна». В общем, Си—живой язык, он развивается, в нем появляются новые конструкции и даже «анахронизмы».

В предисловии мы не будем перечислять все эти изменения. Во-первых, это предисловие, и если мы сейчас вам скажем, что в последних версиях языка появился новый тип данных — «перечисление», то вряд ли это произведет на вас должное впечатление. Во-вторых, если вы будете работать с языком Си, писать на нем программы, то для того, чтобы выполнять эти программы, вам придется познакомиться с руководством по работе на вполне конкретной машине. Там же и будут перечислены все сделанные в конкретной версии изменения.

Теперь подробнее остановимся на переводе. Как правило, при профессиональном программировании часто приходится иметь дело с понятием «адреса». Универсальные языки программирования всячески стараются избегать этого понятия, считая его чисто «машинным», из-за чего возникают свои специфические трудности. Частично их преодолевают, вводя понятие «указателя», просто алгоритмического аналога адреса, а частично — вводя понятие «разыменования». Это уже более сложное понятие, особенно если в языке сложная система описания типов переменных. Дело в том, что имена переменных, скажем a и b, в разных контекстах надо трактовать по-разному. Ясно, что если мы напишем оператор $a=b$, то a и b означают разные вещи: b относится к значению переменной b, а a указывает, именует, адресует переменную a и не имеет никакого отношения к значению этой переменной. Если мы хотим одинаково хорошо манипулировать и значениями переменных, и их адресами, именами и т. д. (а при программировании на низшем уровне это необходимо), мы должны точно говорить, что есть что. Наиболее логичное (или по крайней мере четкое) решение принято, например, в языке Bliss, где употребление имени переменной всегда сопоставляется с ее внутренним именем, адресом. Для ссылки же на значение переменной используется операция «точка». В этом языке оператор $a:=b$ означает, что переменная a принимает значение адреса переменной b. Традиционное же присваивание записывается так: $a:=.b$. Хотя такая система крайне простая и четкая, в языке Си она использована не была. Отчасти, очевидно, из-за того, что профессионалы спокойнее относятся к контекстному толкованию и не хотят видеть в программе «лишние» символы: ведь их в конце концов придется самим и писать. Отчасти же это объясняется еще тем, что Си — это некоторое логическое продолжение языка BCPL, созданного М. Ричардсом и, в свою очередь, основанного на универсальном языке

программирования CPL, в создании которого участвовал один из крупнейших ученых в области информатики — Кристофер Стречи. Так как CPL был универсальным языком, а универсальные языки в силу своей природы тяготеют к системам контекстной зависимости и неявного разыменования, то в Си остался именно такой подход. Для пояснения же соответствующих правил в нем используется в некоторых местах понятие l-value, введенное еще М. Ричардсом.

Понятие l-value, которое можно было бы переводить как «л-значение», — это величина, могущая стоять в левой части оператора присваивания, т. е. адресующая нечто. Когда мы говорим о л-значениях, мы подразумеваем, что в этом случае разыменования, т. е. перехода от имени чего-то именующего или адресующего значения к значению, хранящемуся по этому адресу, не происходит: в операции нужен сам этот адрес. В языках профессионального типа такие л-значения могут и должны встречаться и в правой части операторов присваиваний. В этом случае их уже как-то не совсем логично называть л-значениями. И действительно, если в определении языка, приведенном в приложении, понятие л-значения еще встречается, то в самом тексте книги в соответствующих местах уже фигурирует понятие «адрес». В частности, с понятием адреса связана операция взятия адреса, адресации (&), буквально и явно запрещающая автоматическое разыменование в выражениях, т. е. переход от адреса к значению по этому адресу. Аналогичное положение и с операциями «уменьшения» и «увеличения», где нужны адреса (л-значения) переменных, а их значения получаются не путем разыменования, а в результате более сложного процесса. Все эти соображения привели к тому, что в описании языка вместо понятия «л-значение» (l-value) мы стали использовать понятие «адрес». Если речь идет о выражениях, построенных из таких значений или дающих такие значения, то мы говорим об «адресных выражениях» (или «адресующих»).

Несколько слов о системе записи синтаксических правил. В английском языке прилагательные лексически часто не отличаются от существительных, к тому же существительные порой стоят в позиции определения. Кроме того, что это иногда затрудняет понимание, такая система при записи порождающих правил грамматик (их часто просто называют синтаксическими правилами) требует явного указания связи между словами. Ведь о контекстной зависимости в формальных правилах говорить не приходится. Скажем, если в правилах есть понятие name, type, type name, то, встретив в правилах такой фрагмент: ...type name..., вы не можете однозначно определить, то ли речь идет о понятии type, за которым следует name, то ли упоминается понятие type name. В системе записи синтаксических правил, известной под названием БНФ, все синтаксические понятия заключаются в угловые метаскобки. (При этом решаются все проблемы с терминальными символами). В этой системе мы имели бы понятия <name>, <type> и <type name> и никаких неприятностей с пониманием последнего не возникло бы. Однако авторы при записи правил этой системой не пользовались. Надо сказать, что в большинстве описаний языков программирования такие системы записи почему-то всегда подвергаются пересмотру. Керниган и Ритчи образуют единое понятие, соединяя соседние слова символом дефис. Есть, например, type-name и даже struct-or-union-specifier и др. (А как, интересно, поступать, если требуется перенос?) Н. Вирт в описании языка Modula2 вообще сливает несколько слов в одно, выделяя начало каждого слова заглавной буквой. У него те же понятия записывались бы так: Name, Type, TypeName и StructOrUnionSpecifier. В языке Ада

слова, образующие единое понятие, соединяются символом подчеркивания: type, name, type _ name и struct _ or _ union _ specifier. Нам показалось, что такая особенность русского языка, как наличие падежей и прилагательных, вводит явное управление одного слова другим, что позволяет установить структурную связь двух или нескольких слов, образующих единое понятие, не прибегая к явным «управляющим» символам. Скажем, те же три понятия на русском языке записываются так: имя, тип и имя типа. Родительный падеж (типа) разрешает все двусмысленности. Можно было то же понятие назвать и «ти́повое имя». Опять связь между прилагательными и существительным не оставляет лазеек для другого понимания. Правда, при этом не стоит использовать прилагательные в качестве существительных. К счастью, в языках программирования лишь иногда вместо «первичное выражение» используется просто «первичное». Но это скорее небрежность, чем необходимость. Таким образом, можно записывать синтаксические правила, практически не вводя каких-либо метасимволов (в книге таковой только один — «:») и в соответствии с нормами русского языка. Для выделения правил из текста мы используем курсив. С помощью букв русского алфавита образуются названия синтаксических понятий, а символы латинского алфавита, цифры и другие символы считаются терминальными. (Символ «:», кроме того, играет роль метасимвола, отделяя определяемое понятие от определения.) Само определение начинается с новой строки. Каждая альтернатива определения также начинается с новой строки. «Образ» всего определения синтаксического понятия выглядит так:

синтаксическое понятие:
  первая альтернатива определения
  вторая альтернатива определения
  ⋮
  n-я альтернатива определения

Первоначально перевод и той и другой книги был выполнен с использованием русских служебных слов. Дело в том, что если мы стремимся к информатизации общества, то переход на родной для этого общества язык или хотя бы использование соответствующей лексики представляется необходимым. В конце концов наступит такой момент, когда мы вынуждены будем «транспонировать» используемые нами языки программирования и ввести русскую лексику. Однако опасение, что при наборе текста программ могут появиться досадные опечатки, заставило нас отказаться от использования в переводе русских служебных слов. Ниже для удобства пользователей приводится список эквивалентных английских и русских служебных слов.

цел	int	авто	auto	продолж	continue
сим	char	внешние	extern	если	if
плав	float	регистр	register	иначе	else
двойная	double	тип	typdef	для	for
запись	struct	стат	static	повтор	do
смесь	union	на	goto	пока	while
длинные	long	возврат	return	перекл	switch
короткие	short	размер	sizeof	вариант	case
бз	unsigned	разрыв	break	прочие	default
		вход	entry		

Служебные слова препроцессора менее наглядны, но их английские «братья» не лучше.

# опред	# define	# еслиопр	# ifdef
# неопр	# undef	# еслинеопр	# ifndef
# включ	# include	# конесли	# endif
# если	# if	# строка	# line

Аналогичная участь постигла и «русские» идентификаторы в программах. Здесь сыграли свою роль следующие соображения.

Многие из упоминающихся в книге подпрограмм имеют аналоги в системе UNIX, но так как предметом книги является язык Си, а не система UNIX, то мы не могли взять на себя смелость переделать названия таких процедур из этой системы. Конечно, для удобочитаемости можно было перевести часть идентификаторов, оставляя типично «юниксовские» в первозданном виде, но после решения не изменять служебные слова это было бы уже бессмысленно. Если в текстах программ мы так ничего и не изменили, то зачем, спрашивается, об этом говорить? Причин две.

Первая. В языке Си встроен препроцессор, который позволяет воспользоваться русскими служебными словами даже при трансляторе, воспринимающем английскую лексику. Правда, для этого система должна воспринимать как минимум символы русского алфавита. В этом случае нужно написать соответствующие макроопределения, поместить их в некоторый файл и включать всегда этот файл в нашу программу с русскими служебными словами. Препроцессор переведет этот текст на английский язык и передаст его на трансляцию.

Вторая. Русский перевод английских служебных слов нужно знать и потому, что он определяет, диктует многие названия для понятий, так или иначе связанных с этими ключевыми словами. Весь текст построен на том, что читатель знает, что означает то или иное служебное слово.

При переводе как служебных слов, так и других наименований мы стремились максимально использовать уже сложившуюся, устоявшуюся терминологию, бытующую в языках программирования. В частности, мы во многом используем терминологию, употреблявшуюся при переводах описаний языков Ада и Паскаль, выпущенных издательством «Финансы и статистика». Нам кажется, что единство терминологии только подчеркивает концептуальное единство большинства языков программирования.

Особо следует остановиться на второй книге, включенной в настоящее издание. При переводе ее следовало бы назвать не «Задачник...», а «Головоломки...». Это набор довольно запутанных задач, в которых требуется не написать какую-либо программу, что, вообще говоря, сделать довольно просто, особенно если у вас нет возможности проверить, правильно ли и хорошо ли вы это сделали, а наоборот, вам нужно разобраться в написанных кем-то программках и понять их поведение. Это типичная ситуация для создателя программы, начинающего ее отлаживать. Нам кажется, что написание такой книги — вещь очень трудная, и это, пожалуй, единственный пример издания такого типа.

Таким образом есть все основания предполагать, что читатель получит наиболее полное на сегодняшний день представление о языке Си.

Д. Подшивалов

ПРЕДИСЛОВИЕ

Си — это универсальный язык программирования, характеризующийся экономной записью выражений, современными механизмами управления вычислениями и структурами данных и богатым выбором операций. Это не язык «очень высокого уровня», это не «большой» язык; он не специализирован для каких-либо отдельных приложений. Однако отсутствие ограничений и универсальность делают его более удобным и эффективным для большинства задач, чем предположительно более мощные языки.

Первоначально язык Си был создан Д. Ритчи для операционной системы UNIX на ЭВМ PDP-11 и реализован в этой системе. Сама операционная система, транслятор для Си и, что особенно важно, все прикладные программы (включая и те, с помощью которых готовилось английское издание данной книги) были написаны на языке Си. Для многих машин существуют работающие («промышленные») трансляторы; сюда входят IBM/370, Honeywell 6000 и Interdata 8/32. Язык не ориентирован на какую-либо аппаратуру или систему, однако на нем легко писать программы, которые без изменения работают на любых машинах, поддерживающих язык Си.

Цель нашей книги — помочь читателю научиться программировать на Си. Здесь есть обзор, предназначенный для как можно более быстрого введения новых пользователей в курс дела, несколько глав, посвященных основным свойствам языка и справочник. Изложение основано на чтении, составлении и переделке примеров программ, а не просто на перечислении различных правил. При этом почти всегда наши примеры — не изолированные фрагменты, а полные и реальные программы. Причем они были оттестированы прямо так, как приводятся в тексте, поскольку текст сам по себе пригоден для обработки на машине. Кроме того, чтобы показать, как эффективно использовать язык Си, мы еще стремились, где это было возможно, продемонстрировать полезные алгоритмы и принципы хорошего стиля и здравого проектирования.

Данная книга не есть введение в программирование, она предполагает некоторое знакомство с такими основными концепциями программирования, как переменные, оператор присваивания и функции. Однако новичок также может читать ее и разбираться в языке, хотя при этом помощь более опытных коллег ему не помешает.

Опыт доказал, что Си — это удобный, выразительный и гибкий язык для программирования широкого класса задач. Его легко выучить, и понимание его углубляется по мере работы с ним. Мы надеемся, что наша книга поможет вам освоиться с этим языком.

Брайан В. Керниган
Деннис М. Ритчи

Глава 1. ВВЕДЕНИЕ

Си — название универсального языка программирования. Он очень тесно связан с системой UNIX, так как был создан в этой системе и так как UNIX и ее программное обеспечение написаны именно на нем. Однако язык не связан с какой-либо системой и машиной, и, хотя он уже называется «языком системного программирования», поскольку оказался полезным при написании операционных систем, на нем столь же хорошо можно программировать большинство численных задач и задач, связанных с обработкой текстов или базами данных.

Это язык относительно «низкого уровня». Такая характеристика «не бросает в него камень», она просто означает, что Си работает с теми же объектами, с которыми работает и большинство вычислительных машин, а именно с символами, числами и адресами. К ним и их комбинациям можно применять обычные арифметические и логические операции, существующие в сегодняшних машинах.

В языке нет операций, работающих непосредственно с составными объектами вроде строк символов, множеств, списков или массивов, рассматриваемых как нечто целое. Здесь нет, например, никаких аналогов операций из ПЛ/1 для манипуляции с целыми массивами или строками. В языке не определяются никакие средства управления памятью, кроме статических определений и стековой дисциплины для локальных переменных функций; здесь нет механизмов типа «кучи» из Алгола 68 и его «сборщика мусора». Наконец, в самом Си нет даже операций ввода-вывода, нет операторов ЧИТАТЬ или ПИСАТЬ и нет встроенных методов доступа к файлам. Эти механизмы высшего уровня должны обеспечиваться явно вызываемыми функциями.

Аналогично же в языке предусмотрены только элементарные, простые конструкции управления, ориентированные на единственный процессор: это проверки, циклы, объединения в группы и подпрограммы. Здесь нет никакого мультипрограммирования, параллельных операций, синхронизации или сопрограмм.

Хотя отсутствие некоторых из таких свойств и может показаться существенным недостатком («Это что же, я должен обращаться к функции для сравнения двух строк символов?»), тем не менее то, что язык удалось «уменьшить» до теперешних его размеров, имеет и реальные выгоды. Поскольку Си относительно невелик, то его описание весьма кратко и его очень просто выучить. Любой транслятор можно сделать простым и компактным. Писать его просто. Надеемся, что, используя современную технологию, удастся составить транслятор для новой машины за пару месяцев, причем обнаружить, что почти 80% программы нового транслятора совпадает с уже существующими трансляторами. Это обеспечивает высокую степень переносимости языка. Поскольку типы данных

в языке и его управляющие конструкции непосредственно поддерживаются большинством существующих вычислительных машин, то библиотека административной системы для реализации замкнутых программ очень мала. Например, на PDP-11 в нее входят лишь программы для 32-разрядного умножения и деления и для входа и выхода из подпрограмм. Кроме того, конечно, в каждой реализации существует исчерпывающая, совместимая библиотека для функций ввода-вывода, работ со строками и операций по выделению памяти, однако так как к ним обращаются явно, то в случае необходимости от них можно и избавиться; кроме того, эти функции можно записать и на Си, причем они будут переносимы.

И еще, поскольку язык соответствует возможностям современных машин, то программы на нем достаточно эффективны и нет необходимости пользоваться языком Ассемблера. Наиболее очевидный тому пример — сама операционная система UNIX. Она почти полностью написана на языке Си. Из 13 000 строк программ системы только около 800 строк самого низшего уровня написаны на языке Ассемблера. Кроме того, фактически все прикладное обеспечение в UNIX написано на Си; подавляющее большинство пользователей этой системы (в том числе и один из авторов этой книги) даже не знают язык Ассемблера PDP-11.

Хотя в языке и нашли отражение особенности сегодняшних машин, тем не менее он не связан с какой-либо конкретной архитектурой, и поэтому на нем легко (для этого требуется лишь немного внимания) писать «переносимые» программы, которые будут без изменения работать на самых разных машинах. Теперь это обычный прием, когда обеспечение, развитое в рамках системы UNIX, переносится в окружение систем IBM, Honeywell и Interdata. Фактически трансляторы и административные системы Си на этих четырех системах более совместимы между собой, чем даже стандартная версия Фортрана ANSI. Сама же операционная система сейчас работает и на PDP-11, и на Interdata 8/32. Если не считать программ, которые по необходимости зависят от машины, скажем, трансляторы, Ассемблеры или отладчики, то написанное на Си программное обеспечение на этих машинах полностью идентично. В самих же операционных системах из 7000 строк, не связанных с поддержкой Ассемблера или обслуживанием устройств ввода-вывода, около 95% идентичны.

Для программистов, знакомых с другими языками, может быть будет полезным для сравнения познакомиться с некоторыми историческими, техническими и философскими аспектами языка Си.

Многие из наиболее важных идей Си вытекают из значительно более старого, но все еще вполне жизненного языка BCPL, созданного Мартином Ричардсом. Язык BCPL повлиял на Си косвенно, через язык B, развитый Кэном Томпсоном в 1970 г. для первой системы UNIX на PDP-7.

Хотя Си и присущи некоторые характерные особенности BCPL, нельзя сказать, что это его диалект. И BCPL, и B — языки «без типов», единственными данными в них были машинные слова; обращение к другим объектам идет через специальные операции или функции. В Си же символы, целые разного размера и числа с плавающей запятой — это «фундаментальные» объекты (данные), и из них с помощью ссылок, массивов, записей, смесей и функций строится целая иерархия производных типов.

В нашем языке предусмотрены и основные управляющие конструкции, требующиеся для хорошего структурирования программ, — группирование, ветвление (if), циклы с проверкой на окончание в начале (while, for) и в конце (do), выборы одного из нескольких вариантов (switch). (Все это было предусмотрено и в BCPL, хотя синтаксис и был другим; этот язык на несколько лет предвосхитил моду на «структурное программирование».)

В Си предусмотрены ссылки и можно выполнять операции «адресной» арифметики. Аргументы передаются функциям путем копирования значений аргументов; функции, к которым произошло обращение, не могут изменять фактические аргументы в обратившейся программе. Если нужно добиться «вызова по ссылке», то нужно явно передать ссылку, а функция может уже изменить объект, на который указывает эта ссылка. Имена массивов передаются как местоположения начал этих массивов, поэтому аргументы-массивы фактически передаются по ссылке.

К любой функции можно обращаться рекурсивно, и ее локальные переменные обычно бывают «автоматическими», т. е. заново порождаются при каждом вызове. Определения функций не могут вкладываться одно в другое, но описания переменных носят блочный характер. Функции в Си могут транслироваться отдельно. По отношению к некоторой функции переменные могут быть внутренними, внешними (однако известными лишь в одном единственном исходном файле) и действительно глобальными. Автоматические переменные могут для увеличения эффективности размещаться в регистрах, однако описание переменной как регистровой — это только подсказка для транслятора, не связанная с конкретными регистрами машины.

Си не есть язык, строго основанный на концепции типа, как Паскаль или Алгол 68. К преобразованию типов в нем относятся вполне терпимо, хотя, конечно, здесь нет никакого автоматического преобразования типов данных вроде тех, что в изобилии предусмотрены в ПЛ/1. Существующие трансляторы не обеспечивают во время работы программы никаких проверок индексов массивов, типов аргументов и т. д.

В тех случаях, когда желателен строгий контроль типов, используется особая версия транслятора. Эта программа называется lint, в частности, из-за того, что она разбирает программу «по ниткам». Lint не формирует никакой программы, вместо этого она очень тщательно проверяет программу с разных точек зрения, т. е. контролирует все, что можно проконтролировать при трансляции и загрузке. Программа lint обнаруживает неверное использование типов, несостоятельное использование аргументов, неиспользуемые и, конечно, неинициированные переменные, потенциальные трудности при переносе и тому подобные вещи. Программы; которые беспрепятственно прошли через «сито» lint, как правило, не содержат ошибок, связанных с типами, так же как их не могут содержать, скажем, программы на Алголе 68.

Ну и, конечно, Си, как и любой другой язык, имеет свои отрицательные стороны. Для некоторых операций выбран неверный приоритет, в некоторых местах синтаксис мог быть и получше; язык существует в нескольких версиях, лишь немного отличающихся одна от другой. Однако, как бы то ни было, опыт показал, что язык Си — это очень эффективный и выразительный язык для широкого спектра применений.

Дальше книга строится следующим образом. Глава 2 представляет собой обзор основных характерных особенностей языка. Цель ее — дать читателю насколько возможно быстро основные сведения, так как мы глубоко убеждены, что единственный способ выучить новый язык — это начать на нем программировать. Наш обзор предполагает наличие рабочих знаний основных элементов программирования; мы не объясняем, что такое вычислительная машина, что такое трансляция и что означает оператор $n = n + 1$. Хотя мы и пытаемся, где это возможно, показать полезные приемы программирования, книга не претендует на роль справочника по структурам данных и алгоритмам. Если нужно выбирать, мы всегда отдаем предпочтение вопросам, связанным с языком.

В главах с 3-й по 7-ю рассматриваются более детально различные аспекты языка. Часто это делается и более формально, хотя мы опять же стремимся, чтобы примеры были полными и полезными программами, а не изолированными фрагментами. Глава 3 посвящена основным типам данных, операциям и выражениям. В главе 4 рассматриваются управляющие операторы: if-else, while, for и т. д. В главе 5 обсуждаются функции и структура программы—внешние переменные, области действия и т. п. Глава 6 связана со ссылками и адресной арифметикой. В главе 7 приводятся детали таких данных, как записи и смеси.

В главе 8 мы описываем стандартную библиотеку ввода-вывода, обеспечивающую общие правила взаимодействия с операционной системой. Такая библиотека поддерживается на всех машинах, где есть язык Си, поэтому программы, использующие эту библиотеку для ввода, вывода и других системных функций, можно фактически без изменений переносить из одной системы в другую.

Глава 9 посвящена описанию правил взаимодействия программ, написанных на Си, с операционной системой UNIX. Причем обсуждение концентрируется на вводе-выводе, файловой системе и вопросах переноса. Хотя эта глава и несколько специфична, тем не менее программисты, не работающие конкретно с UNIX, смогут найти здесь полезный материал, скажем, о том, как реализована некоторая версия стандартной библиотеки, или соображения о том, как достигается переносимость программ.

Приложение представляет собою справочник по языку Си. Это «официальное» описание синтаксиса и семантики языка и окончательный арбитр при любых неточностях или пропусках в предыдущих главах (если, конечно, не считать таковым ваш собственный транслятор).

Поскольку Си — это развивающийся язык, существующий во многих системах, некоторый материал нашей книги может и не соответствовать сегодняшнему этапу развития той или иной системы. Мы пытались выделить эти аспекты и предупредить о потенциальных трудностях. Если у нас были сомнения, то мы, как правило, останавливались на варианте, характерном для UNIX, поскольку именно с этим окружением связано большинство программистов, использующих Си. Кроме того, в приложении мы описали различия в реализациях для большинства систем.

Глава 2. ОБЗОР ЯЗЫКА

Начнем с быстрого введения в язык Си. Наша цель — продемонстрировать важные элементы языка в реальных программах, не опускаясь до деталей, формальных правил и исключений из них. Исходя из этого принципа, мы не будем пытаться быть точными и охватить все, что можно (однако примеры должны быть правильными). Мы хотим как можно быстрее подвести читателя к тому, чтобы он сам смог писать полезные программы. Для этого необходимо обратить внимание на самые основные вещи: на переменные и константы, на арифметику, на управление, на функции и на самый простейший ввод-вывод. Мы намеренно опускаем в этой главе те особенности языка, которые жизненно необходимы для написания больших программ. Сюда входят ссылки, записи, большинство операций (их набор в языке весьма обширен), некоторые операторы управления и тьма различных деталей.

Конечно, такой подход имеет свои недостатки. Наиболее существенным представляется то, что одно отдельное характерное свойство языка не описывается полностью в одном месте; с точки же зрения обучения краткость может приводить к неверному пониманию. Примеры, так как в них не используется полностью вся «мощь» языка, не столь четки и элегантны, как могли бы быть. Конечно, мы пытались свести эти эффекты к минимуму, но сказать об этом нужно.

Недостатком представляется и то, что по необходимости в других главах придется повторяться. Однако мы надеемся, что польза от этих повторений более чем компенсирует возможное раздражение при чтении.

В любом случае опытный программист должен стремиться «экстраполировать» материал этой главы применительно к своим собственным нуждам, а начинающий — писать небольшие собственные программы. И те, и другие могут использовать материал данной главы как некоторый набросок, который будет уточняться при более детальных описаниях, начинающихся лишь с гл. 3.

2.1. НАЧНЕМ, ПОЖАЛУЙ

Единственный способ обучиться новому языку программирования — начать писать на нем программы. Первая программа, которую пишут на всех языках, одна.

Напечатайте слова:

```
hello, world
```

Скачка началась, и вот уже первое препятствие. Чтобы перепрыгнуть через него, вы должны уметь где-то создать текст программы, успешно

его оттранслировать, загрузить, выполнить и разобраться, что же у вас получилось. После того как вы освоите эти механические действия, все остальное будет сравнительно простым.

На нашем языке программа, печатающая hello, world, выглядит так:

```
main()
{
    printf("hello, world\n");
}
```

Как выполнить такую программу, зависит от системы, которую вы используете. Например, в операционной системе UNIX вы должны в некотором файле с именем, заканчивающимся на «.c», скажем hello.c, создать входную программу и затем оттранслировать ее, дав команду

cc hello.c

Если вы сделали все верно и не пропустили, например, символа или не натворили еще чего-либо, то трансляция пройдет «молча» и будет создан исполняемый файл с именем a.out. Если выполнить его, дав команду

a.out

то на выходе получим

```
hello, world
```

В других системах правила будут другими; чтобы узнать их, нужно получить консультацию у местных экспертов.

Упражнение 2.1. Выполните эту программу в вашей системе. Попробуйте выбросить что-либо из программы и посмотрите, какие сообщения при этом вы получите.

Поясним теперь саму программу. В языке Си любая программа, каким бы ни был ее размер, состоит из одной или более «функций», задающих действия (вычисления), которые нужно выполнить. Функции в нашем языке подобны функциям или подпрограммам в Фортране или процедурам в ПЛ/1, Паскале и других языках. В нашем примере это функция main. Обычно вы вольны давать функциям какие угодно имена, но main — это особое имя: выполнение программы начинается с функции main. Это означает, что в любой программе main где-нибудь должна быть. Такая главная функция обычно обращается к другим функциям, и они выполняют свою работу. Некоторые функции находятся в самой программе, а некоторые извлекаются из библиотеки предварительно написанных функций.

Одним из механизмов организации взаимодействия между функциями являются аргументы. Список аргументов в круглых скобках идет следом за именем функции; наша функция main аргументов не имеет, поэтому список выглядит так: ( ). Скобки { } обрамляют операторы, которые и «выполняют». саму работу. Эти скобки аналогичны DO-END в ПЛ/1, begin-end в Алголе, Паскале и т. д. Обращаются к функции по имени. За ним в скобках следует список аргументов. Ничего подобного оператору CALL в Фортране или ПЛ/1 у нас нет. Скобки должны быть даже если нет никаких аргументов.

Строка, в которой написано

```
printf("hello, world\n");
```

это обращение к функции; вызывается функция с именем printf, аргумент—"hello, world\n". Функция printf—это библиотечная функция, печатающая информацию на терминале (если не указано какое-либо дру-

гое устройство). В данном случае печатается строка символов, заданная как аргумент.

Последовательность любого числа символов, заключенная в двойные кавычки "...", называется *строкой символов* или *строковой константой*. Пока мы будем использовать строки символов только в качестве аргументов для printf и других функций.

Комбинация \n в строках языка Си означает символ перехода на новую строку. Если его «напечатать» на терминале, то курсор перейдет в самую левую позицию следующей строки. Если \n опустить (стоит провести эксперимент), то обнаружим, что наша выдача не заканчивается пустой строкой. Единственный способ включить символ перехода на новую строку в аргумент для printf — использовать комбинацию \n. Если вы попытаетесь написать, скажем, так:

```
printf("hello, world
");
```

то транслятор мрачно сообщит вам, что пропущены кавычки.

Сама функция printf автоматически на новую строку не переходит, так что ею можно пользоваться для формирования выходной строки в несколько этапов. Нашу первую программу можно было написать и таким образом:

```
main()
{
        printf("hello, ");
        printf("world");
        printf("\n");
}
```

Результат был бы точно таким же.

Обратите внимание, что \n представляет ровно один символ. *Образы*, аналогичные \n, есть общий и гибкий механизм представления управляющих или «невидимых» символов*. Среди тех, которые предусмотрены в Си, \t означает табуляцию, \b—возврат каретки на одну позицию, \"—двойную кавычку, а \\—сам символ \.

Упражнение 2.2. Поэкспериментируйте, что произойдет, если в строку аргумент для printf включить \x, где x — символ, не входящий в число перечисленных выше.

2.2. ПЕРЕМЕННЫЕ И АРИФМЕТИКА

Следующая наша программа печатает приведенную ниже таблицу температур по Фаренгейту и их эквивалент по стоградусной шкале (по Цельсию). Для перевода используется формула $C = (5/9)(F - 32)$.

0	— 17.8
20	— 6.7
40	4.4
60	15.6
...	...
260	126.7
280	137.8
300	148.9

Вот сама программа:

---

* Часто вместо слова «образ» мы говорим просто о «комбинации» символов.—*Примеч. пер.*

```
/* печать таблицы Фаренгейт—Цельсий
     для  f = 0, 20, ..., 300 */
main()
{
     int lower, upper, step;
     float fahr, celsius;

     lower = 0;      /* нижний предел температур */
     upper = 300;    /* верхний предел */
     step = 20;      /* шаг */

     fahr = lower;
     while (fahr <= upper) {
          celsius = (5.0/9.0) * (fahr-32.0);
          printf("%4.0f %6.1f\n", fahr, celsius);
}         fahr = fahr + step;
```

Первые две строки — *примечания*:

```
/* печать таблицы Фаренгейт — Цельсий
     для f = 0,20,...,300 */
```

Оно кратко поясняет, что делает программа. Любые символы между /* и */ транслятор пропускает, и их можно спокойно использовать для облегчения понимания программы. Примечания могут появляться везде, где допускаются пробелы или переходы на новую строку.

Все переменные должны до их использования описываться; обычно это делается в начале функции, перед любым из исполняемых операторов. Если вы забыли поставить описание, то транслятор об этом сразу же вам напомнит. Описание состоит из некоторого *типа* и списка переменных, имеющих этот тип, например

```
int lower, upper, step;
float fahr, celsius;
```

Тип int предполагает, что перечисленные переменные — целые, float — *с плавающей точкой*, т. е. числа могут иметь дробную часть. Точность представления для типов int и float зависит от конкретной машины, на которой вы работаете. Например, на PDP-11 числа типа int это 16-разрядные числа со знаком, т. е. они лежат в диапазоне от -32768 до $+32767$. Числа же типа float—это 32-разрядные величины, в них приблизительно семь значащих цифр, а абсолютное значение лежит в интервале 10^{-38} -10^{+38}. В главе 3 приведены соответствующие характеристики для других машин.

Кроме типов int и float в языке предусмотрено еще несколько основных типов:

```
char   — символ — один байт;
short  — короткие целые;
long   — длинные целые;
double — числа с плавающей точкой двойной точности.
```

Размеры объектов этих типов также зависят от машины и приводятся в гл. 3. Кроме того, существуют *массивы, записи* и *смеси* из этих основных типов, *ссылки* на объекты, *функции*, дающие объекты разных типов. В свое время мы о них будем говорить.

Фактически сами вычисления в программе преобразования температуры начинаются с присваиваний:

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

где происходит установка начальных значений переменных. Каждый оператор заканчивается точкой с запятой.

Каждая строка таблицы вычисляется одним и тем же способом, поэтому мы используем циклическое повторение этих вычислений для каждой строки. Для этого есть оператор while.

```
while (fahr <= upper) {
    ...
}
```

Проверяется условие, стоящее в скобках. Если оно справедливо (т.е. fahr меньше или равно upper), то выполняется тело цикла (все операторы, заключенные в скобки {и}). Затем повторяется проверка условия, и, если оно справедливо, тело вновь выполняется. Если проверка говорит, что условие ложное (fahr превышает upper), то цикл заканчивается, и выполняется оператор, следующий за циклом. В нашей программе больше операторов нет, и она заканчивается.

Тело цикла (оператора while) может состоять из нескольких, как в случае преобразования температуры, операторов или из одного оператора, тогда, как в приведенном ниже примере, ставить скобки нет нужды.

```
while (i < j)
    i = 2 * i;
```

В любом случае операторы, которыми управляет цикл while, лучше выделять путем «табуляции», т. е. перед ними остается пустое пространство. Такое выделение подчеркивает логическую структуру программы. Хотя сам язык Си никак не регламентирует расположение операторов, тем не менее удачное выделение с помощью пробелов и распределения по строкам весьма важно для чтения программы человеком. Мы рекомендуем писать в каждой строке по одному оператору и обычно выделять пробелами операции. Расположение скобок менее важно, мы выбрали просто один из нескольких популярных стилей. Остановитесь на подходящем вам и используйте его постоянно.

Большая часть вычислений нашей программы проделывается в теле цикла. Температура по Цельсию вычисляется и присваивается переменной celsius в операторе

```
celsius = (5.0/9.0) * (fahr-32.0);
```

Деление 5.0/9.0 вместо 5/9 используется из-за того, что в Си, как и во многих других языках, целое деление идет с *отбрасыванием,* поэтому всякая дробная часть отбрасывается. В результате 5/9 дает нуль, и такое же значение получат все температуры. Десятичная точка в константе указывает на плавающую точку, так что 5.0/9.0 дает 0.555..., именно то, что мы хотим.

Мы пишем 32.0 вместо 32 даже если fahr типа float и 32 должно быть перед вычитанием автоматически преобразовано к float (к 32.0). Это уже элементы стиля; целесообразно писать константы с плавающей точкой, явно указывая эту десятичную точку, даже если это и целое значение. Такое написание лишь подчеркивает их природу для человека,

читающего программу, и гарантирует, что транслятор обратит на это внимание.

Подробности правил преобразования целых в числа с плавающей точкой приводятся в гл. 3. А теперь обратите внимание, что присваивание

```
fahr = lower;
```

и проверка

```
while (fahr <= upper)
```

работают, как и ожидалось, — int превращается перед выполнением операции в float.

Наш пример дает чуть больше знаний о работе функции printf. Фактически это универсальная функция преобразования формата. Полностью мы ее опишем в гл. 8. Ее первый аргумент — строка символов, которые нужно печатать, причем символ % указывает на места, где должны вставляться другие аргументы (второй, третий, ...) и в каком виде они должны печататься. Например, в операторе

```
printf("%4.0f %6.1f\n", fahr, celsius);
```

спецификация преобразования %4.0f, указывает, что число с плавающей точкой нужно напечатать в поле по крайней мере из четырех позиций и без цифр после десятичной точки. Спецификация %6.1f указывает, что число занимает по крайней мере 6 позиций и после точки есть одна цифра. Это похоже на формат F6.1 в Фортране или F(6,1) в ПЛ/1. Части спецификаций можно опускать: %6f указывает, что число должно быть по крайней мере в шести позициях; %.2f требует, чтобы после точки было две позиции, а общий размер не ограничивается; %f просто говорит, что нужно печатать число с плавающей точкой. В printf предусмотрены: %d — для десятичных целых, %o — для восьмеричных, %x — для шестнадцатеричных, %c — для символов, %s — для символьных строк и %% — для самого символа %.

Каждая конструкция с % в первом аргументе printf сопоставляется соответственно со вторым, третьим и т. д. аргументами, и должно быть соответствие по числу аргументов и типу. Если это не так, то ответ бессмыслен.

Между прочим, функция printf *не входит* органически в сам язык Си; в нем ни вывод, ни ввод не определены. И в printf нет ничего таинственного; это просто полезная функция, входящая в стандартную библиотеку программ, обычно доступную для любой вашей программы. Для того чтобы сконцентрироваться на самом языке, разговор о вводе-выводе мы откладываем до гл. 8. Там же речь пойдет и о форматном вводе. Если же вам потребуется вводить числа, то посмотрите разд. 8.4, где описана функция scanf; эта функция во многом похожа на printf, но только она вводит информацию, а не выводит.

Упражнение 2.3. Модифицируйте программу преобразования температур, чтобы она печатала заголовок у таблицы.

Упражнение 2.4. Напишите программу для печати таблицы соответствия температур по Цельсию температурам по Фаренгейту.

2.3. ОПЕРАТОР FOR

Как и можно было ожидать, существует множество способов написать некоторую программу. Попытаемся изменить программу преобразования температур:

```
main()      /* таблица Фаренгейт–Цельсии */
{
     int fahr;

     for (fahr = 0; fahr <= 300; fahr = fahr + 20)
          printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Эта программа дает тот же результат, но выглядит она, конечно, совсем по-иному. Основное отличие в том, что почти все переменные исчезли, осталась только fahr, но типа int (что позволяет продемонстрировать в printf спецификацию %d). Верхний и нижний пределы и размер шага фигурируют в виде констант в операторе for. Это новая конструкция. Кроме того, выражение, вычисляющее температуру по Цельсию, появляется как третий аргумент printf, а не пишется в отдельном операторе присваивания.

Это последнее изменение — пример весьма общего в Си правила: везде, где можно, указывать значение переменной некоторого типа, можно использовать и выражение такого же типа. Так как третий аргумент у printf должен быть типа float, чтобы соответствовать %6.1f, то на этом месте может стоять любое выражение с плавающей точкой.

Сам оператор for — это цикл, обобщение цикла while. Если вы сравните его с ранее приведенным оператором while, то ясными станут операции, которые в этом случае выполняются. Оператор содержит три части, отделяемые друг от друга точкой с запятой. Первая:

```
fahr = 0
```

выполняется лишь однажды, перед входом в собственно цикл. Вторая часть — проверка или условие, которое управляет циклом:

```
fahr <= 300
```

Это условие вычисляется. Если оно справедливо, то выполняется тело цикла (в данном случае единственное обращение к printf). Затем следует повторная инициация цикла:

```
fahr = fahr + 20
```

и опять повторное вычисление условия. Цикл заканчивается, если условие становится ложным. Как и в операторе while, тело цикла может состоять из единственного оператора или из нескольких, заключенных в скобки операторов. Инициация и повторная инициация могут быть любым, но единственным выражением.

Выбирать между циклом while и for следует, ориентируясь на ясность записи. Оператор for подходит для циклов, где инициация и повторная инициация представлены каждая одним выражением и логически связаны; это делает цикл более компактным, чем цикл while, все действия по управлению циклом сосредоточиваются в одном месте.

Упражнение 2.5. Модифицируйте программу преобразования температуры для печати в обратном порядке, т. е. от 300 до 0°.

2.4. СИМВОЛИЧЕСКИЕ КОНСТАНТЫ

И наконец, прежде чем навсегда покончить с программой преобразования температур, последнее замечание. Очень плохо, если в программе встречаются «загадочные числа» вроде 300 или 20: для тех, кто будет

читать затем программу, они несут очень мало информации и их очень трудно заменить на другие некие систематическим способом. Если использовать в начале программы конструкцию со словом #define, то можно для некоторой конкретной строки символов ввести *символическое имя*, т. е. определить *символическую константу*. Везде потом транслятор будет заменять вхождение указанного имени (если оно встречается вне кавычек) на соответствующую строку. Замена имени связана не только с числами, а вообще с любыми текстами.

```
#define   LOWER   0        /* нижний предел */
#define   UPPER   300      /* верхний предел */
#define   STEP    20       /* шаг */

main()    /* таблица Фаренгейт—Цельсий */
{
     int fahr;

     for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
          printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Величины LOWER, UPPER и STEP суть константы, поэтому в описаниях они не появляются. Символические имена обычно записываются заглавными буквами, так что их можно легко отличить от написанных строчными буквами имен переменных. Обратите внимание, что в конце описания нет точки с запятой. Так как вместо имени подставляется вся следующая за ним строка, то оказалось бы, что в операторе for слишком много точек с запятой.

2.5. НЕКОТОРЫЕ ПРОСТЫЕ ПРОГРАММЫ

Теперь мы рассмотрим семейство связанных друг с другом программ, выполняющих простые действия с символьными данными. Мы потом обнаружим, что многие из программ будут просто расширенными версиями тех прототипов, о которых сейчас пойдет речь.

Ввод и вывод символа

В стандартной библиотеке предусмотрены функции для чтения и записи одного символа (за одно обращение). Функция getchar( ) берет следующий входной символ каждый раз, когда к ней обращаются, и выдает его как значение функции. Таким образом, после обращения

```
c = getchar()
```

переменная c содержит следующий входной символ. Обычно этот символ приходит с терминала, но пока нам нет нужды об этом знать (до гл. 8).

Функция putchar(c) дополняет getchar:

```
putchar(c)
```

печатает где-то, содержимое переменной c, опять обычно это «где-то»— терминал. Обращения к putchar и к printf можно чередовать, выдача будет идти в том порядке, в каком шли обращения.

Функции getchar и putchar не есть нечто специальное, так же как и printf. Они не принадлежат языку Си, но всегда доступны.

Копирование файла

Уже с помощью getchar и putchar вы можете написать удивительно много полезных программ, не зная ничего больше о вводе-выводе. Наиболее простым примером будет копирование по одному символу входа на выход. Схематически это выглядит так:

```
взять один символ
    (символ не есть признак конца файла)
     выдать только что полученный символ
     взять один новый символ
```

Преобразование этой схемы в программу на Си дает

```
main()     /* копирование входа на выход; 1-я версия */
{
     int c;

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

Операция отношения != означает «не равно».

Основная проблема — выделить конец файла. Обычно getchar в случае обнаружения конца ввода выдает значение, не являющееся допустимым символом. Используя этот факт, программа может обнаружить, что она дошла до конца файла. Однако есть некоторое усложнение: существуют *два* соглашения, касающиеся фактического значения для конца файла. Мы обходим этот нюанс, используя для этого значения символическое имя EOF. На практике EOF бывает —1 или 0, так что программе должно предшествовать или определение

```
#define  EOF  -1
```

или определение

```
#define  EOF  0
```

И тогда все будет работать верно. Используя символическую константу EOF, представляющую значение, которое выдает getchar, встретив конец файла, мы предполагаем, что только одна эта вещь в программе зависит от специфического числового значения.

Кроме того, c мы описываем как int, а не char, поскольку здесь должно храниться значение, выдаваемое из getchar. В гл. 3 мы убедимся, что это действительно целое значение, так как мы должны уметь представлять EOF дополнительно ко всем возможным символам.

Программу копирования более опытный и знакомый с Си программист мог бы написать и более кратко. В нашем языке любое присваивание, например

```
c = getchar ( )
```

можно использовать в любом выражении; его значение — это просто значение, присваиваемое левой части. Если присваивание символа переменной c «спрятать» внутрь проверяющей части оператора while, то программу копирования файла можно записать и так:

```
main()     /* копирование входа на выход ; 2-я версия */

     int c;
```

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

Эта программа берет символ, присваивает его c, а затем проверяет, не является ли этот символ признаком конца файла. Если нет, то выполняется тело цикла while и символ печатается. Затем повторяется цикл. Если же наконец достигнут конец ввода, то цикл кончается, а с ним кончается и программа main.

В такой версии программы ввод централизован, есть только одно обращение к getchar и программа упрощается. Включение присваиваний в проверки — одно из тех мест в языке, которые приводят к столь ценимой компактности. (Ввод можно выполнить и в другом месте и породить недоступную для понимания программу. Эту тенденцию мы старались обуздать.)

Важно понять, что скобки вокруг присваивания внутри условия действительно необходимы. *Приоритет* операции != выше приоритета =, это означает, что если бы не было скобок, то != должна была бы выполняться перед присваиванием =. Таким образом, оператор

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

эквивалентен

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

Этот нежелательный эффект приведет к тому, что c получит значение 0 или 1, в зависимости от того, встретится ли при обращении к getchar конец файла или нет. (Подробнее об этом в гл. 3.)

Подсчет символов

Следующая программа считает символы; это некоторое развитие программы копирования.

```
main()      /* подсчет вводимых символов */
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Оператор

```
++nc;
```

вводит новую операцию, ++, означающую *увеличение на единицу*. Вы могли бы написать и nc=nc+1, но ++nc более коротко и часто более эффективно. Существует и соответствующая операция --, уменьшающая на 1. Операции ++ и -- могут быть и префиксными (++nc), и постфиксными (nc++). В выражениях эти две формы, как мы увидим в гл. 3, дают различные значения, однако и ++nc, и nc++ обе увеличивают nc. Пока же мы будем придерживаться префиксной формы.

Программа счета символов накапливает счетчик в длинной переменной, а не в целой. На PDP-11 максимальное значение для int 32767 и уже

относительно небольшой ввод переполнит счетчик, если он будет типа int; в IBM и Honeywell long и int — синонимы и значительно больше по размеру. Спецификация преобразования %ld сигнализирует printf, что соответствующий аргумент—длинное целое.

Для того чтобы копировать еще больше символов, вы можете использовать двойные («плавающие» числа удвоенного размера). Кроме того, вместо цикла while, чтобы показать другой способ построения цикла, мы используем цикл for.

```
main()     /* подсчет вводимых символов */
{
     double nc;

     for (nc = 0; getchar() != EOF; ++nc)
          ;
     printf("%.0f\n", nc);
}
```

В printf %f используется и для float, и для double; %.0f гасит печать несуществующей дробной части.

В данном случае тело цикла for *пустое*, ибо вся работа делается при проверке и повторной инициации. Однако грамматические правила языка требуют, чтобы у оператора цикла было тело. Изолированная точка с запятой (*пустой оператор*) стоит просто для удовлетворения этого требования. Мы ее поставили в отдельной строке, чтобы выделить ее.

Прежде чем покончить с программой, считающей символы, заметим, что если на входе нет ни одного символа, то операторы while и for получат отрицательный ответ при самом первом обращении к getchar, и, таким образом, программа даст нуль — верный ответ. Это важное замечание. Одно из достоинств операторов for и while то, что в них проверка идет в самом *начале* цикла, перед работой его тела. Если нечего делать, то ничего и не делается, даже если это означает: никогда не входить в тело цикла. Программы должны умно работать, когда они обрабатывают ввод, «не содержащий символов». И операторы while и for позволяют быть уверенными, что при граничных условиях делаются разумные вещи.

Подсчет строчек

Следующая наша программа считает *строки* входного текста. Предполагается, что входные строки заканчиваются символом перехода на новую строку \n, который «с чисто религиозным упорством» добавляется к каждой выдаваемой наружу строке.

```
main()     /* подсчет строчек */
{
     int c, nl;

     nl = 0;
     while ((c = getchar()) != EOF)
          if (c == '\n')
               ++nl;
     printf("%d\n", nl);
}
```

Тело оператора while теперь состоит из одного оператора if, который управляет опять же одним увеличением ++nl. Оператор if проверяет заключенное в скобки условие, и, если оно справедливо, то выполняется

следующий оператор (или группа операторов в фигурных скобках). Мы здесь снова делаем отступ, дабы показать, чем управляет оператор.

Двойной знак равенства == в языке означает «равно» (подобно .EQ. в Фортране). Такая комбинация используется в проверках из-за того, что одиночный символ = используется в присваивании. Так как присваивание почти в два раза чаще встречается в обычных программах, то будет естественным, если знак операции для него будет вдвое короче.

Любой (но только один) символ можно записать в одиночных кавычках, порождая тем самым значение, равное численному значению этого символа во множестве символов машины. Такое значение называется *символьной константой*. Так например, 'A' — символьная константа, во множестве ASCII этот символ имеет значение 65, это внутреннее представление для A. Конечно, предпочтительнее употреблять 'A', а не 65: значение константы очевидно и не зависит от конкретного множества символов.

Специальные комбинации в строках, в символьных константах допускаются, так что в проверках и арифметических выражениях '\n' заменяет значение символа перехода на новую строчку. Однако следует помнить, что '\n' — это один символ, а в выражении это одно целое; с другой стороны "\n" есть символьная строка, содержащая только один символ. Комментарии по поводу строк из одного символа можно найти в гл. 3.

Упражнение 2.6. Напишите программу, считающую пробелы, символы табуляции и новые строки.

Упражнение 2.7. Напишите программу копирования ввода на вывод с заменой строки из одного или более пробелов на один пробел.

Упражнение 2.8. Напишите программу, заменяющую каждый символ табуляции на последовательность из трех символов: >, возврат на один шаг,—(при печати это ⩾, а символ возврата на один шаг—на ⩽ Это позволит «увидеть» символы табуляции и возврата.

Подсчет слов

Четвертая из серии наших полезных программ считает строки, слова и символы, причем словом считается любая последовательность символов не содержащая пробелов, символов табуляции или переходов на новую строку. Это некоторая версия обслуживающей программы из системы UNIX с именем wc.

```
#define   YES  1
#define   NO   0

main()    /* подсчет строчек, слов и символов */
{
      int c, nl, nw, nc, inword;

      inword = NO;
      nl = nw = nc = 0;
      while ((c = getchar()) != EOF) {
            ++nc;
            if (c == '\n')
                  ++nl;
            if (c == ' ' || c == '\n' || c == '\t')
                  inword = NO;
```

```
            else if (inword == NO) {
                inword = YES;
                ++nw;
            }
        }
        printf("%d %d %d\n", nl, nw, nc);
}
```

Как только программа встречает первыи символ, она начинает счет. Переменная inword фиксирует, находимся ли мы сейчас внутри слова, или же нет; в самом начале мы «не в слове» и этой переменной присваивается значение NO. Мы отдаем предпочтение символическим константам YES и NO, а не буквальным значениям 1 и 0 из-за легкости чтения программы. Конечно, в такой «мизерной» программе разница не особенно заметна, но в больших программах увеличение «прозрачности» вполне оправдывает дополнительные усилия, потраченные на то, чтобы уже с самого начала писать программы в таком стиле. Кроме того, программу легко и изменять, если числа в ней фигурируют только как символические константы.

В строке

```
nl = nw = nc = 0;
```

всем трем переменным присваивается нуль. Это не какой-то специальный случай, а просто следствие того факта, что присваивание имеет значение и присваивания выполняются справа налево, как если бы мы написали

```
nc = (nl = (nw = 0));
```

Операция || означает ИЛИ, так что в строке

```
if (c == ' ' || c == '\n' || c == '\t')
```

говорится: «если с есть пробел или с — переход на новую строку или с — символ табуляции... (Комбинация \t суть видимое представление символа табуляции). Есть соответствующая операция и для И — &&. Выражения, связанные && или ||, вычисляются слева направо и гарантируется, что вычисления окончатся, как только станет известна истинность или ложность высказывания. Таким образом, если с хранит пробел, то нет нужды проверять, не хранится ли там переход на новую строку или символ табуляции, и эти проверки действительно не делаются. В данном случае этот факт не так важен, но он может оказаться весьма существенным, как мы скоро увидим, в более сложных ситуациях.

В этом примере мы знакомимся и с оператором else, определяющим альтернативное действие, которое нужно выполнять, если условие в операторе if ложно. Общий вид условного оператора

if (*выражение*)
    оператор-1
else
    оператор-2

Из двух операторов, связанных с комбинацией if-else, выполняется один и только один. Если *выражение* истинно, то выполняется *оператор-1*, если нет — *оператор-2*. Каждый из операторов может быть довольно сложным. В программе счета слов оператор после else есть условный оператор, управляющий двумя операторами в фигурных скобках.

Упражнение 2.9. Как вы хотели бы проверить программу подсчета слов? Что ограничивает проверку?

Упражнение 2.10. Напишите программу, печатающую вводимые слова по одному в строке.

Упражнение 2.11. Переделайте программу подсчета слов, введя улучшенное определение для понятия «слово». Например, слово — это последовательность букв, цифр и апострофов, начинающаяся с буквы.

2.6. МАССИВЫ

Попробуем написать программу для подсчета вхождений каждой из цифр, невидимых (пустых) символов (пробелов, табуляций, переходов на новую строку) и других символов. Конечно, это несколько искусственная задача, но на ней можно показать некоторые особенности языка.

Так как мы имеем дело с входными символами двенадцати категорий, то для хранения числа вхождений каждой из цифр удобнее использовать массив, а не десять отдельных переменных. Вот один из вариантов программы:

```
main()     /* подсчет цифр, пустых символов и пр. */
{
     int  c, i, nwhite, nother;
     int  ndigit[10];

     nwhite = nother = 0;
     for (i = 0; i < 10; ++i)
          ndigit[i] = 0;

     while ((c = getchar()) != EOF)
          if (c >= '0' && c <= '9')
               ++ndigit[c-'0'];
          else if (c == ' ' || c == '\n' || c == '\t')
               ++nwhite;
          else
               ++nother;

     printf("digits =");
     for (i = 0; i < 10; ++i)
          printf(" %d", ndigit[i]);
     printf("\nwhite space = %d, other = %d\n",
          nwhite, nother);
}
```

Описание

```
int  ndigit[10];
```

вводит ndigit как массив из 10 целых чисел. Индексы массива в языке всегда начинаются с 0, а не с 1 как в Фортране или ПЛ/1, т. е. есть элементы ndigit[0], ndigit[1], ..., ndigit[9]. Это отражается на циклах for, инициирующих и печатающих этот массив.

Индекс может быть любым целым выражением, в которое, конечно, входят целые переменные вроде i и целые константы.

Эта конкретная программа в значительной степени основывается на свойствах представления символов, соответствующих цифрам. Например, при проверке

```
if (c >= '0' && c <= '9') ...
```

определяется, цифра ли c или нет. Если да, то числовое значение этой цифры c — '0'. Это верно только в случае, если '0', '1' и т. д. — целые числа в возрастающем порядке и между '0' и '9' нет ничего, кроме цифр. К счастью, так чаще всего и бывает.

По определению при арифметических действиях, включающих величины типа char и int, сначала проводится преобразование к типу int, поэтому переменные и константы типа char фактически в таких местах эквивалентны целым. Это вполне естественно и удобно: например, c — '0' — целое выражение со значением между 0 и 9, соответствующее некоторому символу между '0' и '9', хранящемуся в c; это значение допустимо как индекс для массива ndigit.

Выяснение, представляет ли символ цифру, невидимый символ или нечто другое, идет в операторах:

```
if (c >= '0' && c <=  9')
     ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
     ++nwhite;
else
     ++nother;
```

Конструкция типа

```
if (условие)
     оператор
else if (условие)
     оператор
else
     оператор
```

в программах часто встречается при вынесении решения более чем с двумя возможностями. Мы просто проходим по программе до тех пор, пока не удовлетворится одно из *условий*, в этот момент вычисляется соответствующий *оператор*, и на этом вся конструкция кончается. (*Оператор* же, конечно, может состоять из нескольких заключенных в фигурные скобки операторов.) Если не выполняется ни одно из условий, то выполняется *оператор* после последнего else, если оно есть. Если последнего else нет (как было в программе подсчета слов), то не выполняется никаких действий. Между начальным else и последним else может встречаться произвольное число комбинаций:

```
else if (условие)
     оператор
```

Рекомендуется размещать эти конструкции именно так, как показано, при этом в длинных последовательностях решений мы не будем сдвигаться далеко вправо на странице.

«Многоходовые» ветвления можно программировать и с помощью оператора переключателя (switch). О нем речь пойдет в гл. 4. Переключатель, в частности, удобен, если проверка связана с совпадением целого или символьного выражения с одной из множества констант. Для сравнения в гл. 4 мы приведем версию нашей программы, написанную с использованием переключателя.

Упражнение 2.12. Напишите программу, печатающую гистограмму длин читаемых слов. Гистограмму легче располагать горизонтально; вертикальная ее реализация более трудна для программирования.

2.7. ФУНКЦИИ

В языке Си *функции* представляют собою аналог подпрограмм или функций в Фортране и процедур в ПЛ/1, Паскале и т. п. Функция дает возможность удобно выразить некоторые вычисления и «заключить их в чер-

ный ящик», так что в дальнейшем ими можно пользоваться, не беспокоясь о содержимом. В действительности же с потенциальной сложностью больших программ можно справиться только с помощью функций. Если функции построены надлежащим образом, можно игнорировать вопрос, *как* они выполняют работу; достаточно знать, *что* они делают. При проектировании языка мы стремились облегчить использование функций, стремились к удобству и эффективности. Часто вы будете встречаться с функциями в несколько строк, причем вызываемыми лишь единожды, только для более ясного написания некоторого фрагмента программ.

До сих пор мы пользовались только функциями вроде printf, getchar и putchar, уже для нас приготовленными; теперь самое время написать несколько своих собственных. Так как у нас в языке нет операции возведения в степень вроде ** в Фортране или ПЛ/1, можно проиллюстрировать механизм определения функции на примере функции power (m, n) для возведения целого числа m в целую положительную степень n. Таким образом, значение power(2,5) — 32. Конечно, наша функция не делает ту же работу, что и **, поскольку речь идет о положительных степенях небольших целых чисел, но лучше продвигаться последовательно.

Мы приводим как функцию power, так и главную программу, обращающуюся к ней; таким образом, вы видите всю структуру взаимодействия целиком.

```
main()     /* тест функции power  */
{
     int i;

     for (i = 0; i < 10; ++i)
          printf("%d %d %d\n", i, power(2,i), power(-3,i));
}

 power(x, n)     /*  возведение x в n-ю степень ; n > 0 */
(int x, n)
 (
      int i, p;

      p = 1;
      for (i = 1; i <= n; ++i)
           p = p * x;
      return(p);
 )
```

Всякая функция имеет такой вид:

```
имя (список аргументов, если он есть)
описания аргументов, если они есть
{
      описания
      операторы
}
```

Функции могут появляться в любом порядке в одном или нескольких входных файлах. Конечно, если на вход идут два файла, то для их трансляции и загрузки придется сказать больше, чем в случае одного файла, но это скорее относится к операционной системе, чем к самому языку. Пока же мы предполагаем, что обе функции находятся в одном файле, и поэтому наши знания о выполнении программ на языке Си не изменяются.

К функции power обращаются дважды в одной строчке:

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

При каждом обращении функции передаются два аргумента, а она возвращает целое число, которое нужно привести к должному формату и напечатать. В выражениях power(2, i) такое же целое, как 2 и i. (Не все функции дают целые значения; об этом мы будем говорить в гл. 5.)

В функции power аргументы должны быть соответствующим образом описаны, так что их типы известны. Это делается в строчке

```
int x, n;
```

следующей за именем функции. Описание аргументов идет между списком аргументов и открывающей левой фигурной скобкой; каждое описание заканчивается точкой с запятой. Имена, использованные в power для аргументов, полностью локальны по отношению к данной функции и недоступны для любых других функций. Эти имена можно использовать в других подпрограммах, не опасаясь каких-либо конфликтов. Это же справедливо и для переменных i и p; в функции power i не имеет отношения к i в главной подпрограмме.

Значение, вычисляемое в функции power, передается в главную подпрограмму с помощью оператора return (возврат), так же как и в ПЛ/1. В скобках у него может встречаться любое выражение. Функция не обязательно возвращает какое-либо значение; оператор возврата без выражения просто передает управление, возвращаясь к обратившейся программе; то же происходит и при «выходе через конец», т. е. по достижении закрывающей правой фигурной скобки.

Упражнение 2.13. Напишите программу для преобразования входной информации в символы нижнего регистра. Используйте функцию lower (c), выдающую c, если это не буква, и значение c на нижнем регистре, если c — буква.

2.8. АРГУМЕНТЫ — ВЫЗОВ ПО ЗНАЧЕНИЮ

У функций в языке Си есть особенность, возможно, и незнакомая программистам, работающим с другими языками, в частности с Фортраном и ПЛ/1: все аргументы функций передаются «по значению», т. е. вызванная функция получает в промежуточных переменных (фактически в стеке) значения своих аргументов, а не их адреса. Это приводит к некоторым различиям по сравнению с Фортраном и ПЛ/1, где передача идет «по ссылке», т. е. вызванная подпрограмма обрабатывает адреса аргументов, а не их значения.

Основное отличие заключается в том, что в Си вызванная функция не может изменить переменные в функции, обратившейся к ней, она может менять только свои собственные, промежуточные их копии.

Однако вызов по значению имеет кроме ограничений и некоторое ценное качество. Обычно он приводит к более компактным программам без многих вспомогательных переменных, так как аргументы можно рассматривать в вызванной подпрограмме как обычным образом инициированные переменные. Вот, например, вариант функции power, где используется этот факт.

```
power(x, n)   /* возведение x в n-ю степень; n > 0; версия 2 */
int x, n;
{
     int p;

     for (p = 1; n > 0; --n)
          p = p * x;
}    return(p);
```

Аргумент n будет промежуточной переменной: он выполняет роль счетчика, уменьшающегося до нуля. Надобность в переменной i отпадает. Что бы ни делалось с n внутри функции, это не влияет на значение аргумента, с которым первоначально обращались к функции power.

Если необходимо, то функцию можно приспособить для изменения переменных в обратившейся программе. Обращающийся должен давать *адрес* переменной, которую нужно менять (мы называем это *ссылкой* на переменную), а вызванная функция должна описывать аргумент как некоторую ссылку и обращаться к фактической переменной косвенно, через эту ссылку. Детали этого процесса будут рассматриваться в гл. 5.

Если в качестве аргумента берется имя массива, то передаваемое функции значение фактически есть адрес начала массива. (*Никакое* копирование элементов массива не производится.) Индексируя это значение, функция может выбирать и изменять любые элементы массива. Об этом речь пойдет в следующем разделе.

2.9. МАССИВЫ СИМВОЛОВ

Вероятно, наиболее часто встречающимся типом массива в языке Си будет массив символов. В качестве иллюстрации работы с массивами символов и функций, манипулирующих с ними, разберем программу, читающую несколько входных строк и печатающую самую длинную из них. Ее схема достаточно проста:

```
while (есть другие строки)
    if (строка длиннее, чем старая самая длинная)
    сохранить строку и ее размер
печатать самую длинную строку
```

Из схемы ясно, что программа естественным образом делится на части. Одна часть выбирает новую строку, другая проверяет ее, третья сохраняет, а остальные лишь управляют процессом.

Так как процесс великолепно разделяется на отдельные части, его следовало бы также и описывать. Поэтому давайте вначале соответственно напишем отдельную функцию getline, выбирающую *следующую строку* из входного потока; это обобщение getchar. Чтобы функция была полезна и для других применений, попытаемся сделать ее насколько возможно гибкой. Как минимум она должна сигнализировать о возможном конце файла; в более общем случае хорошо бы, если бы она давала размер строки и нуль при обнаружении конца файла. Нуль как возможный размер строки не встречается, ибо каждая строка имеет по крайней мере один символ — символ перехода на новую строку.

Если мы обнаруживаем, что строка длиннее, чем предыдущая самая длинная, то мы должны ее где-то сохранить. Значит, у нас должна быть функция copy, сохраняющая строку в надежном месте.

И наконец, нам нужна основная программа, управляющая функциями getline и copy. В результате имеем

```
#define   MAXLINE   1000 /* максимальный размер строки */

main()    /* поиск длиннейшей строки */
{
    int   len; /* длина текущей строки */
    int   max; /* максимально зафиксированная длина */
    char  line[MAXLINE]; /* текущая строка */
    char  save[MAXLINE]; /* длиннейшая строка */
```

```
        max = 0;
        while ((len = getline(line, MAXLINE)) > 0)
                if (len > max) {
                        max = len;
                        copy(line, save);
                }
        if (max > 0)    /* строка уже была */
                printf("%s", save);
}

getline(s, lim)         /* считывание строки в s, вычисление размера */
char s[];
int lim;
{
        int c, i;

        for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
                s[i] = c;
        if (c == '\n') {
                s[i] = c;
                ++i;
        }
        s[i] = '\0';
        return(i);
}

copy(s1, s2)    /* копирование s1 в s2; s2 достаточного размера */
char s1[], s2[];
{
        int i;

        i = 0;
        while ((s2[i] = s1[i]) != '\0')
                ++i;
}
```

Функции main и getline взаимодействуют через пару аргументов и выдаваемое значение. В getline аргументы описываются в строках:

```
char s[];
int lim;
```

где задается, что первый аргумент есть массив, а второй — целое значение. Длина массива в getline не задается, так как она определяется в main. Как и в функции power, в getline для выдачи значения в обратившуюся программу используется оператор возврата. Иногда функции выдают некоторые полезные значения, хотя иные из них, вроде copy, полезны лишь своим побочным эффектом и не выдают что-либо осмысленное.

Функция getline в конце порождаемого ею массива помещает символ \0 (*нулевой* символ, со значением нуль), указывающий на конец строки символов. Такое же соглашение принято и в трансляторе с языка Си: если в программе встречается строковая константа вроде

```
"hello\n"
```

то транслятор порождает массив символов, содержащий символы этой строки, и заканчивает его символом \0, так что функция типа printf всегда может обнаружить конец массива:

h	e	l	l	o	\n	\0

В printf спецификация формата вида %s как раз и говорит, что строка представлена именно в таком виде. Если посмотреть функцию copy, вы обнаружите, что она также ориентирована на то, что входной аргумент s1 заканчивается \0, и копирует этот символ в выходной аргумент s2. (Все это предполагает, что \0 в обычные тексты не включается.)

Между прочим, стоит заметить, что даже такие небольшие программы, как наша, уже порождают некоторые чисто «конструкторские» проблемы. Например, что надо делать, если встречается строка, превышающая заданный предел? Программа getline работает правильно и останавливает ввод, если массив заполнился, а переход на новую строку еще не пришел. В основной программе по размеру и последнему выданному символу проверяется, не была ли строка слишком большой, затем идет копирование. В интересах краткости мы все эти подробности в нашей программе опустили.

При использовании getline способа заранее узнать, сколь длинными могут быть входные строки нет, поэтому в getline предусматривается контроль на переполнение. С другой стороны, при использовании функции copy уже известно (или можно найти), какого размера будут строки, и поэтому мы решили не включать сюда дополнительный контроль.

Упражнение 2.14. Переделайте главную программу поиска самой длинной строки так, чтобы она правильно печатала размер произвольно длинной входной строки и воспроизводила ее текст, насколько это возможно.

Упражнение 2. 15. Напишите программу печати всех строк размером более 80 символов.

Упражнение 2.16. Напишите программу, убирающую начальные пробелы и символы табуляции из каждой входной строки и исключающую пустые строки.

Упражнение 2.17. Напишите функцию reverse(s), переставляющую символы строки s в обратном порядке. Используйте ее для программы, «реверсирующей» входные строки.

2.10. ОБЛАСТИ ДЕЙСТВИЯ; ВНЕШНИЕ ПЕРЕМЕННЫЕ

Переменные программы main (line, save и т. д.) будут собственными или локальными по отношению к main, так как они описаны внутри ее. Никакие другие функции не имеют к ним прямого доступа. Это же справедливо и для переменных других функций, например переменная i в getline никак не связана с i в copy. Любая локальная в программе переменная «начинает существовать» только после обращения к этой функции и *уничтожается* по выходе из функции.

Такие переменные, следуя терминологии других языков программирования, можно называть *автоматическими*. Поэтому и мы начиная с этого момента будем называть такие динамические переменные автоматическими. (В гл. 5 мы будем говорить о *статической* памяти, в ней переменные сохраняют свои значения между обращениями к функциям.)

Так как возникновение и существование автоматических переменных связано с обращением к функции, то их значения не сохраняются от вызова к вызову и их надо явно устанавливать при каждом обращении. Если этого не сделать, то они будут содержать «мусор».

Как нечто противоположное автоматическим переменным можно определить и переменные, являющиеся *внешними* для всех функций. Это глобальные переменные, и к ним можно обращаться по именам из любой функции, где это нужно. (Такой механизм подобен общим блокам в Фортране или внешним переменным в ПЛ/1.) Поскольку внешние переменные доступны везде, их можно использовать вместо аргументов для связи между функциями. Более того, так как эти переменные существуют всегда, а не возникают и исчезают по вызове и выходе из функции, их значения будут

сохраняться даже после того, как мы покинем функцию, в которой они устанавливались.

Внешние переменные должны *определяться* вне какой-либо функции, при этом для них выделяется фактическая память. В любой функции, обращающейся к таким переменным, они должны *описываться;* это делается либо с помощью явного описания — extern (внешние), либо неявно, по контексту. Для большей конкретности перепишем программу для самой большой строки, сделав переменные line, save и max внешними. Это приведет к изменениям в обращениях, описаниях и телах всех трех функций.

```
#define   MAXLINE   1000 /* максимальный размер строки */

char line[MAXLINE]; /* входная строка */
char save[MAXLINE]; /* длиннейшая строка */
int  max; /* максимальная зафиксированная длина */

main()    /* поиск длиннейшей строки ; специализированная версия */
{
     int len;
     extern int max;
     extern char save[];

     max = 0;
     while ((len = getline()) > 0)
          if (len > max) {
               max = len;
               copy();
          }
     if (max > 0)    /* строка уже была */
          printf("%s", save);
}

getline() /* специализированная версия */
{
     int c, i;
     extern char line[];

     for (i = 0; i < MAXLINE-1
          && (c=getchar()) != EOF && c != '\n'; ++i)
               line[i] = c;
     if (c == '\n') {
          line[i] = c;
          ++i;
     }
     line[i] = '\0';
     return(i);
}

copy()    /* специализированная версия */
{
     int i;
     extern char line[], save[];

     i = 0;
     while ((save[i] = line[i]) != '\0')
          ++i;
}
```

Внешние переменные для main , getline и copy *определяются* в первых строках нашего примера. Здесь указывается их тип, и они (определения) приводят к выделению памяти для описанных переменных. Синтаксически определения внешних абсолютно подобны ранее использованным описаниям, но, так как они находятся вне функции, переменные, о которых идет речь,— внешние. Прежде чем функция может использовать внешнюю переменную, нужно сделать имя этой переменной известным функции. Один из способов сделать это — включить в функцию описание внешней переменной. От обычного описания оно отличается только добавлением зарезервированного слова extern.

При некоторых обстоятельствах описание внешних может быть опущено. Если определение внешней переменной встречается во входном файле раньше ее использования в какой-либо конкретной функции, то в этой функции нет необходимости описывать эту переменную как внешнюю. Таким образом, описания extern в main, getline и copy избыточны. Фактически обычно принято располагать определения всех внешних переменных в начале входного файла, а затем все описания со словом extern опускать.

Если программа находится в нескольких входных файлах и некоторая переменная определяется, скажем, в *файл1*, а используется в *файл2*, то в *файл2* для фиксации связи между двумя вхождениями переменной необходимо описание внешних. Этот вопрос рассматривается в гл. 5.

Нужно заметить, что в этом разделе, когда речь идет о внешних переменных, мы точны в употреблении слов *описание* и *определение*. «Определение» относится к месту, где фактически порождается или отводится память; слово же «описание» относится к месту, где фиксируется природа переменной, но память для нее не отводится.

Между прочим, есть тенденция все делать через внешние переменные, так как кажется, что они упрощают связи: и списки аргументов становятся короче, и переменные всегда есть, когда они нужны. Однако такие внешние переменные есть всегда, даже когда они не нужны. Такой стиль программирования крайне опасен, так как он ведет к программам, у которых связи по данным не всегда очевидны, — переменные могут изменяться неожиданным и даже таинственным способом; программу такого типа в случае надобности трудно модифицировать. Вторая версия программы поиска самой длинной строки хуже первой отчасти из-за этих причин, а отчасти и из-за того, что в ней разрушена универсальность двух крайне полезных функций: мы вписали в них имена переменных, с которыми они будут работать.

Упражнение 2.18. Проверка в операторе for в приведенной выше программе getline несколько неудачна. Перепишите эту программу, сделав проверку «яснее», но сохраните поведение программы в случае конца файла или переполнения буфера. А так ли уж осмысленно это поведение?

2.11. ЗАКЛЮЧЕНИЕ

Мы уже описали то, что можно было бы назвать стандартным ядром языка Си. Имея под рукой такой набор «кирпичиков», мы можем уже строить полезные программы приличного размера, и, вероятно, было бы хорошо, если бы читатель на некоторое время занялся именно такой работой. Упражнения, приводимые ниже, предоставляют читателю повод написать программы несколько более сложные, чем те, о которых шла речь в данной главе.

После того как вы проделаете это, вы будете представлять себе большую часть языка, и поэтому стоит направлять усилия уже на чтение следующих немногочисленных глав, где выявляется и становится очевидной мощность и выразительность Си.

Упражнение 2.19. Напишите программу detab, заменяющую символы табуляции во входном потоке на надлежащее (до следующей установленной табуляции) число пробелов. Предположим, что «остановки» табуляции идут на фиксированном расстоянии друг от друга, скажем, через n позиций.

Упражнение 2.20. Напишите программу entab, заменяющую строки из пробелов минимальным числом символов табуляции и пробелов, обеспечивающих тот же эффект. Используйте ту же установку табуляции, что в detab.

Упражнение 2.21. Напишите программу, «складывающую» длинные входные строки после последнего непустого символа, встретившегося до n-го входного (n — параметр). Убедитесь, что ваша программа сделает нечто разумное, даже если к заданному моменту она не встретит ни пробелов, ни символов табуляции.

Упражнение 2.22. Напишите программу, убирающую из программы на языке Си все примечания. Не забудьте нужным образом обрабатывать строки, заключенные в кавычки, и символьные константы.

Упражнение 2.23. Напишите программу поиска в программах на Си элементарных синтаксических ошибок вроде несбалансированных фигурных, круглых и квадратных скобок. Не забывайте про двойные и одиночные кавычки и примечания. (Если такая программа делается достаточно универсально, то это будет весьма сложная программа).

Глава 3. ТИПЫ, ОПЕРАЦИИ И ВЫРАЖЕНИЯ

Основные объекты, с которыми работает программа, представляют собою переменные и константы. Переменные перечисляются в описаниях, где указывается их тип и, может быть, их начальные значения. Операции определяют, что делается с этими переменными. Выражения комбинируют переменные и константы и позволяют получать новые значения. Именно об этом и пойдет речь в данной главе.

3.1. ИМЕНА ПЕРЕМЕННЫХ

Не будем сейчас все уточнять, и скажем только, что существуют некоторые ограничения на имена переменных и символических констант. Имена составляются из букв и цифр, причем первый символ должен быть буквой. Символ подчеркивания "—" считается буквой. Он полезен для удобочитаемости длинных имен. Различаются строчные и прописные буквы; в языке сложилась традиция использовать строчные буквы для переменных, а прописные—для констант.

Для «внутренних» имен значимыми будут только восемь первых символов, хотя написать можно и больше. Для «внешних» имен, таких, как имена функций или внешних переменных, число символов может быть и меньше восьми, так как с такими именами работают различные ассемблеры и загрузчики. Детали приводятся в приложении 1. Кроме того, в качестве имен переменных нельзя использовать и зарезервированные слова вроде if, else, int, float и т. д.: это *служебные* слова самого языка. (Они должны быть на нижнем регистре.) Имена переменных, естественно, лучше выбирать так, чтобы они были как-то связаны со смыслом самих переменных, и не стоит вводить шрифтовую путаницу.

3.2. ТИПЫ ДАННЫХ И РАЗМЕРЫ

Основных типов данных в Си немного:

char	один байт, где можно хранить один символ из «местного» множества символов;
int	целое число, обычно соответствующее естественному размеру целых чисел в конкретной машине;
float	число с плавающей точкой одинарной точности;
double	число с плавающей точкой двойной точности.

Кроме того, есть несколько «уточнителей», применимых к целым: short, long и unsigned. Уточнения short и long связаны с размером целых чисел, а числа с уточнением unsigned (без знака) подчиняются законам арифметики по mod 2^n, где n — число разрядов в значениях типа int. Числа без знака всегда положительные. Описания с уточнениями выглядят так:

```
short int x;
long int y;
unsigned int z;
```

В таких ситуациях слово int можно опускать, что обычно и делают.

Точность различных целых чисел зависит от конкретной машины. В приведенной таблице даны характеристики некоторых основных представлений:

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8 разрядов	9 разрядов	8 разрядов	8 разрядов
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

Мы хотели, чтобы short и long соответствовали длинам, отличающимся от обычных целых, а int соответствовало бы наиболее «естественному» для данной машины размеру. Как можно видеть, любой транслятор «волен» интерпретировать short и long в соответствии с имеющейся аппаратурой. Все, о чем следует заботиться, — чтобы short не оказались длиннее, чем long.

3.3. КОНСТАНТЫ

С примерами констант типа int и float мы уже встречались, нужно лишь еще раз подчеркнуть, что для типа float допускается и обычная «научная» нотация:

123.456e — 7

или

0.12E3

Любая константа с плавающей точкой считается двойной точности, так что запись с «e» используется и для float, и для double.

Длинные константы записываются вроде этого: 123L. Обычная целая константа, если она слишком велика для int, также считается длинной.

Существуют правила записи восьмеричных и шестнадцатеричных чисел: если перед целым числом идет 0 (нуль), то это восьмеричная константа, начальные 0x или 0X указывают на шестнадцатеричное число. Например, десятичное число 31 как восьмеричное записывается в виде 037, а как шестнадцатеричное — 0X1F или 0x1f. И за восьмеричным, и за шестнадцатеричным числом может следовать буква L, указывающая, что это длинные константы.

Символьная константа — это единственный символ, заключенный в одиночные кавычки, например 'x'. Значение символьной константы — числовое значение символа в представлении, принятом на машине. Во множестве символов ASCII, скажем, значение символа нуль или '0' равно 48, а во множестве EBCDIC '0' равно 240; и то, и другое «сильно» отличается от числа 0 (нуль). Записывая '0' вместо значений вроде 240 или 48, мы делаем программу независимой от конкретного значения. Символьные константы, как и любые другие числа, можно использовать в арифметических операциях, однако чаще всего они используются для сравнения с другими символами.

Символы, не имеющие графического представления, можно записывать, используя специальные комбинации, вроде \n (для новой строки), \t (табуляция), \0 (нуль), \\ (обратная дробная черта), \' (одиночная кавычка) и т. д. Такая комбинация выглядит как два символа, хотя фактически это один символ. Кроме того, аналогично можно записать и

любой двоичный образ одного байта: '\ццц', где *ццц* — от одной до трех восьмеричных цифр. Например:

```
#define   FORMFEED   '\014'     /*  авторегистр в ASCII  */
```

Символьная константа '\0' задает символ с нулевым значением. Ее часто записывают вместо 0, чтобы подчеркнуть символьную природу некоторого выражения.

Константное выражение — это выражение из одних констант. Такие выражения вычисляются во время трансляции, а не во время выполнения программы, поэтому их можно использовать везде, где может стоять константа. Например, в:

```
#define   MAXLINE   1000
char line[MAXLINE+1];
```

или

```
seconds = 60 * 60 * hours;
```

Строковая константа — это последовательность нуля или более символов, заключенная в двойные кавычки, скажем,

```
"I am a string"
```

или

```
"" /* пустая строка */
```

Кавычки не входят в строку, а лишь ограничивают ее. Для представления в строках символов могут применяться и специальные комбинации, в частности \" представляет символ двойной кавычки.

Технически строка представляет собою массив с элементами из одного символа. В конце каждой такой строки транслятор автоматически помещает нулевой символ \0, так что программе «удобно» искать конец строки. Такое представление означает, что размер строки не ограничен каким-либо пределом, но для определения ее длины строку нужно полностью просмотреть. Число необходимых для хранения строки ячеек на единицу превышает число символов в строке. Приведенная ниже функция strlen(s) дает число символов в строке s, включая и последний \0.

```
strlen(s) /* выдается длина s */
char s[];
{
     int i;

     i = 0;
     while (s[i] != '\0')
          ++i;
     return(i);
}
```

Следует отчетливо понимать, что символьная константа и строка из одного символа не одно и то же: 'x' не есть "x". Первое—это один символ, использованный для представления числового значения буквы x в некотором множестве символов. Второе же — строка, содержащая один символ (букву x) и \0.

3.4. ОПИСАНИЯ

Все переменные до их использования должны быть описаны, хотя некоторые описания и выводятся неявно из контекста. Описание задает тип, а затем идет список одной или более переменных этого типа. Например:

```
int  lower, upper, step;
char c, line[1000];
```

Переменные можно распределять по описаниям произвольным образом, упомянутые описания можно записать и так:

```
int  lower;
int  upper;
int  step;
char c;
char line[1000];
```

Хотя при этом и занимается больше места, зато удобно добавлять в каждое из описаний примечания или модифицировать описания в случае необходимости.

Переменным в описаниях можно задавать начальные значения (инициировать), хотя есть и некоторые ограничения. Если за именем идет знак равенства и константа, то это инициация. Например:

```
char backslash = '\\'
int  i = 0;
float eps = 1.0e-5;
```

Если переменная, о которой идет речь, внешняя или статическая, то инициация проводится лишь однажды, до начала выполнения программы. Явно инициированные переменные получают начальные значения при каждом обращении к функции, где они встречаются. Автоматические переменные без явной инициации получают неопределенные значения («мусор»). Подразумевается, что внешние и статические переменные инициируются нулевыми значениями, но лучше будет, если эту инициацию где-либо явно провести.

Мы еще будем говорить об инициации при введении новых типов данных.

3.5. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

Существуют бинарные арифметические операции: +, —, *, / и взятие модуля — %. Есть унарная операция —, но нет унарного плюса.

При делении целых любая дробная часть отбрасывается. Выражение

x % y

дает остаток от деления x на y, поэтому если x точно делится на y, то ответ—нуль. Например, год будет високосным, если он делится на 4, но не делится на 100, однако годы, делящиеся на 400, тоже високосные. Поэтому

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
    год високосный
else
    год не високосный
```

Операцию % нельзя использовать для чисел типа float и double.

Операции + и — имеют один приоритет, причем он ниже, чем (идентичный) приоритет операций *, /, и %, а у них он, в свою очередь, ниже приоритета унарного минуса. Арифметические операции выполняются (группируются) слева направо. (В конце главы приводится таблица, где указываются все приоритеты и порядок выполнения для всех операций.) Для ассоциативных и коммутативных операций вроде * и + порядок вы-

числения не определяется, и транслятор может переупорядочить вычисления даже при наличии скобок. Так, например, a + (b + c) может вычисляться как (a + b) + c. К различным результатам это приводит редко, а если нужен обязательно определенный порядок, то можно использовать явные промежуточные переменные.

Действия, предпринимаемые при переполнении или исчезновении значимости, зависят от конкретной машины.

3.6. ОТНОШЕНИЯ И ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Существуют такие операции отношений:

```
>   >=   <   <=
```

У всех них один и тот же приоритет. У операции равенства приоритет ниже; таких операций две:

```
==   !=
```

Их приоритет один и тот же. Отношения имеют приоритет ниже арифметических операций, так что выражения типа i < lim − 1, как и следовало ожидать, вычисляются как i < (lim − 1).

Логические связки && и || более интересны. Выражения с && и || вычисляются слева направо, и вычисление заканчивается, как только определится истинность или ложность результата. Эти свойства критичны для написания работающих программ. Возьмем, например, цикл из функции getline, описанной в гл. 2:

```
for (i=0; i<lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Ясно, что, прежде чем считывать новый символ, нужно проверить, есть ли в массиве s для него место, поэтому в начале *необходимо* выполнить проверку i < lim − 1. Как бы то ни было, если эта проверка дает отрицательный результат, мы не должны двигаться и читать следующий символ.

Также было бы нелогично проверять c на EOF, не обратившись к getchar; обращение должно идти раньше проверки c.

Приоритет && выше, чем у ||, но оба они ниже, чем у операций отношений и равенства, поэтому в выражениях типа

```
i<lim-1 && (c = getchar()) != '\n' && c != EOF
```

нет необходимости ставить дополнительные скобки. Однако, поскольку приоритет != выше, чем у присваивания, то в

```
(c = getchar()) != '\n'
```

для получения нужного результата требуются дополнительные скобки.

Унарная операция ! преобразует ненулевой или «истинный» операнд в 0, а нулевой или «ложный» операнд — в 1. Операцию ! обычно используют, например, в таких конструкциях:

```
if (!inword)
```

вместо того, чтобы писать

```
if (inword == 0)
```

Трудно сказать, что лучше. Первая конструкция великолепно читается: «если не в слове», но более сложные ее формы могут оказаться тяжелы для понимания.

Упражнение 3.1. Напишите цикл, эквивалентный приведенному циклу for, но не используйте операции &&.

Если в выражении появляются операнды различных типов, то они преобразуются к некоторому общему типу. Этот процесс управляется всего лишь несколькими правилами. В общем, автоматически производятся лишь преобразования, имеющие смысл, скажем, преобразования целых чисел в представление с плавающей точкой в выражениях вроде f + i. Выражения без смысла, например, индексирование числом с плавающей точкой, не допускаются.

Во-первых, значения типа char и int в арифметических выражениях можно свободно «перемешивать»: каждый символ в любом выражении автоматически преобразуется в целое. Это значительно облегчает некоторые трансформации символов. Примером служит функция atoi, превращающая строку цифр в ее числовой эквивалент.

```
atoi(s)    /* преобразование s в целое */
char s[];
{
     int i, n;

     n = 0;
     for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
          n = 10 * n + s[i] - '0';
     return(n);
}
```

Как мы уже говорили в гл. 2, выражение

```
s[i] - '0'
```

дает числовое значение символа, хранящегося в s [i], так как '0', '1' и т. д. образуют «плотную», возрастающую, положительную последовательность.

Еще пример преобразования символов в целые значения. Функция lower отображает всякий символ в соответствующий «строчной» символ, однако это верно *лишь для множества ASCII*. Если символ не относится к заглавным буквам, его функция lower возвращает без изменения.

```
lower(c)   /* преобразование с в строчную; только для ASCII */
int c;
{
     if (c >= 'A' && c <= 'Z')
          return(c + 'a' - 'A');
     else
          return(c);
}
```

Функция будет правильно работать, так как в ASCII между символами верхнего и нижнего регистра, если их рассматривать как числовые значения, сохраняется фиксированная дистанция, а кроме того, каждый алфавит плотный, т. е. между буквами A и Z нет ничего, кроме букв. Это положение для множества EBCDIC (IBM 360/370) *не* выполняется, и в такой системе наша программа будет преобразовывать не только буквы.

В преобразовании символов в целые числа есть одна тонкость: в языке не оговаривается, будет ли переменная типа char считаться числом со знаком или без него? Если char преобразуется в int, то не получится ли в результате отрицательное число? К несчастью, на разных машинах преобразование идет по-разному, это зависит от архитектуры. На некоторых

(например, PDP-11) символы, имеющие 1 в самом левом разряде, превращаются в отрицательные целые (идет «размножение знака»). На других преобразование заключается в добавлении слева «нулей» и поэтому дает всегда положительное целое.

Определение языка гарантирует, что любой символ из стандартного множества символов машины никогда не будет отрицательным, поэтому такие символы можно спокойно употреблять в выражениях как положительные величины. Но если в символьной переменной хранится произвольный набор разрядов, то на некоторых машинах он может оказаться положительным числом, а на некоторых — отрицательным.

Чаще всего с этой ситуацией приходится встречаться, если для представления EOF выбрано значение — 1. Возьмем такой фрагмент программы:

```
char c;

c = getchar();
if (c == EOF)
    ...
```

На машине без размножения знака с всегда положительно, ибо это символ, а EOF — отрицательно. В результате тест всегда будет давать отрицательный ответ. Чтобы избежать этого, нужно переменные, где хранится результат функции getchar, описывать как int, а не char.

Фактически же размножение знака не особенно важно, и в этом случае int следует использовать вместо char просто из-за того, что getchar должна возвращать все возможные символы (поэтому она используется для чтения произвольного входного потока) плюс еще дополнительное значение EOF. Таким образом, значение EOF невозможно представить как символ, и его нужно сохранять как целое значение.

Еще один полезный вид автоматического преобразования типа встречается в отношениях вида i > j или логических выражениях со связками && и ||; значение этих выражений будет 1, если отношение или выражение истинно, и 0 — если ложно. Таким образом, присваивание

```
isdigit = c >= '0' && c <= '9';
```

делает переменную isdigit равной 1, если c — цифра, и 0 — если нет. (При проверке в if, while, for и т. д. «истина» просто означает «не нуль»).

Неявные арифметические преобразования в большинстве случаев выполняются естественным образом. В общем случае при операциях вроде + или * с двумя операндами («бинарные операции»), если операнды разных типов, перед тем как выполнить операцию, «младший» тип подтягивается к «старшему». Результат — старшего типа. Более же точно, к каждому арифметическому операнду применяется следующая последовательность правил преобразования:

char и short преобразуются в int, float преобразуется в double.

Затем, если один из операндов двойной точности, то другой преобразуется к двойной точности, и результат будет двойной точности.

В противном случае, если один из операндов long, то другой преобразуется в long и результат — long.

В противном случае, если один из операндов типа unsigned, то другой преобразуется к тому же типу и результат будет типа unsigned.

В противном случае операнды должны быть типа int и результат получается типа int.

Обратите внимание, что в выражениях все величины типа float преобразуются к двойной точности; в языке Си все действия с плавающей точкой идут с двойной точностью.

Преобразования типов происходят и при присваивании: значение из правой части преобразуется к типу левой части, он и будет типом результата. Любые символы преобразуются, как описывалось выше, в целые, с размножением знака или без размножения. Обратная операция, из целых в символы, идет без осложнений, лишние старшие разряды просто отбрасываются. Таким образом, после выполнения такого фрагмента:

```
int  i;
char c;

i = c;
c = i;
```

значение c не изменяется. Это верно и в случае размножения знака и без него.

Если x типа float, а i — int, то x = i и i = x вызовут преобразование. Преобразование float в int приведет к отбрасыванию любой дробной части. При преобразовании из двойной точности к типу float происходит округление. Длинные целые преобразуются в короткие или символы просто отбрасыванием старших разрядов.

Так как аргументы функции представляют собою выражения, то при передаче аргументов функции также происходит преобразование типов. В частности, char и short переходят в int, а float — double. Именно поэтому аргументы функции должны быть описаны как int и double, даже если к ней обращаются с char и float.

И наконец, преобразование можно сделать принудительно, использовав конструкцию, называемую *приведением*. Выражение вида:

(имя типа) выражение

преобразует *выражение* к поименованному типу по указанным выше правилам. Такую конструкцию, если быть точными, следует рассматривать как присваивание *выражения* переменной указанного типа, и она затем подставляется на место всей конструкции. Например, библиотечная программа sqrt рассчитана на аргумент типа double и выдает чепуху, если к ней обратиться неподобающим образом. Поэтому, если n — целое, то при

```
sqrt((double) n)
```

прежде чем передать параметр функции sqrt, он преобразуется к типу double. (Заметим, что при приведении к нужному типу преобразуется лишь *значение* n, содержимое же n не изменяется.) Операция приведения имеет тот же приоритет, что и другие унарные операции, и это указано в таблице в конце данной главы.

Упражнение 3.2. Напишите функцию htoi(s), преобразующую строку s шестнадцатеричных цифр в эквивалентное целое значение. Допускаются цифры от 0 до 9, от a до f и от A до F.

3.8. ОПЕРАЦИИ УВЕЛИЧЕНИЯ И УМЕНЬШЕНИЯ

В языке предусмотрены две нетрадиционные операции для уменьшения или увеличения значения переменной. Операция увеличения ++ прибавляет 1 к операнду, уменьшения, --, вычитает 1. Мы уже часто использовали ++ для увеличения переменной, например в таких ситуациях:

```
if (c == '\n')
    ++nl;
```

Необычайным является то, что ++ и -- можно использовать и как префиксную операцию (записывая ее перед переменной, ++n), и как

постфиксную (после переменной, n ++). В любом случае эффект заключается в увеличении n. Но в выражении ++n увеличение проводится *до* использования значения n, а в n++ увеличение идет *после* того, как используется значение n. Это означает, что в тех местах, где используется значение, выражения ++n и n++ существенно различны. Например, если n равно 5, то x = n++ присвоит значение 5, а x= ++n присвоит x значение 6. И в том и в другом случае n станет равным 6. Операции увеличения и уменьшения можно применять лишь к переменным, выражения вроде x=(i+j)++ не допускаются.

Если мы сталкиваемся с ситуацией, где имеет значение лишь сам факт увеличения, как, скажем, в

```
if (c == '\n')
    nl++;
```

то выбор между постфиксной или префиксной операцией — дело вкуса. Однако существуют ситуации, где использование той или иной формы определенно напрашивается. Возьмем, например, функцию squeeze(s, c), «выбрасывающую» все вхождения символа c из строки s.

```
squeeze(s, c) /* исключение из s всех c */
char s[];
int c;
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Если встречается символ, отличный от c, он копируется в текущую j-ю позицию, а затем j увеличивается, готовясь тем самым для следующего символа. Эти действия в точности эквивалентны таким:

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Еще один пример подобной конструкции встречается в функции getline, уже приведенной в гл. 2. Здесь можно заменить

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

на более компактную запись

```
if (c == '\n')
    s[i++] = c;
```

В качестве третьего примера рассмотрим функцию strcat(s, t), присоединяющую строку t в конец строки s. В этой функции предполагается, что в s хватает места, чтобы сохранить новую строку.

```
strcat(s, t)    /* добавление t в конец s  */
char s[], t[]; /* s достаточно велика  */
{
     int i, j;

     i = j = 0;
     while (s[i] != '\0')       /* поиск конца s  */
          i++;
     while ((s[i++] = t[j++]) != '\0')   /* копия t  */
          ;
{
```

По мере копирования из t в s, операция ++ в постфиксной манере применяется и к i, и к j, готовя их к следующему проходу по циклу.

Упражнение 3.3. Напишите еще одну версию функции squeeze (s1, s2), которая исключает из s1 любой из символов, встречающихся в строке.

Упражнение 3.4. Напишите функцию any (s1, s2), дающую первое местоположение в строке s1 символа, встречающегося в s2. Если s1 не содержит символов из s2, то функция дает — 1.

3.9. ПОРАЗРЯДНЫЕ ЛОГИЧЕСКИЕ ОПЕРАЦИИ

В языке предусмотрено несколько операций для работы с разрядами; их невозможно применять к float или double.

&	поразрядное И;
|	поразрядное ИЛИ;
^	поразрядное исключающее ИЛИ;
<<	сдвиг влево;
>>	сдвиг вправо;
~	обращение (унарная операция).

Поразрядная операция И (&) часто используется для выделения некоторой группы разрядов. Например:

```
c = n & 0177;
```

устанавливает в нуль все разряды, кроме младших семи. Поразрядная операция ИЛИ (|) используется для «включения» разрядов, например

```
x = x | MASK;
```

устанавливает в единицу те разряды x, которым соответствует 1 в MASK.

Следует отличать поразрядные операции & и | от логических связок && и ||, используемых при вычислении слева направо истинностных значений. Например, если x равно 1, а y равно 2, то x & y равно 0, хотя x && y дает единицу (почему?).

Операции сдвига << и >> выполняют сдвиг левого операнда влево или вправо на число разрядов, задаваемое правым операндом. Таким образом, x<<2 сдвигает x влево на два разряда, заполняя освобождающиеся разряды 0; это соответствует умножению на 4. Сдвиг вправо величины без знака приводит к заполнению нулями освобождающихся разрядов. Сдвиг вправо величины со знаком на некоторых машинах, скажем на PDP-11, приводит к «размножению» знакового разряда («арифметический» сдвиг), а на других машинах освобождающиеся разряды заполняются нулями («логический» сдвиг).

Унарная операция ~ «обращает» целое, т. е. преобразует каждый «единичный» разряд в нулевой и наоборот. Обычно эта операция встречается в выражениях вроде

```
x & ~077
```

которое устанавливает в нуль шесть разрядов x, «маскируя» их. Заметим, что x & ~077 не зависит от размера слова, и поэтому такая запись предпочтительнее, чем, например, x & 0177700, где предполагается, что x — 16-разрядное значение. Такая переносимая запись выражения ничего не стоит, так как ~ 077 суть константное выражение, т. е. оно вычисляется во время трансляции.

Для иллюстрации работы с некоторыми из поразрядных операций рассмотрим функцию getbits(x, p, n), дающую n-разрядное поле (выравненное вправо) из x, начинающееся с позиции p. Мы предполагаем, что 0-й разряд — это самый правый разряд, а n и p — явно положительные величины. Обращение getbits (x, 4, 3) выдает три разряда из 4, 3 и 2-й позиций, сдвинутые вправо.

```
getbits(x, p, n)      /* взять n разрядов, начиная с p-го */
unsigned x, p, n;
{
     return((x >> (p+1-n)) & ~(~0 << n));
}
```

x >> (p + 1 — n) сдвигает требуемое поле в правый конец слова. Описание аргумента x как величины без знака означает, что при сдвиге вправо освобождающиеся разряды заполняются нулями, а не знаковым разрядом, причем это не зависит от машины, на которой выполняется программа. Величина ~0 — все единицы, сдвиг ее на n разрядов (~0<<n) порождает маску с нулями в n правых разрядах и единицами в других местах. Беря дополнение, мы получаем маску из единиц в правых разрядах.

Упражнение 3.5. Модифицируйте функцию getbits, считая, что разряды нумеруются слева направо.

Упражнение 3.6. Напишите функцию wordlength( ), вычисляющую размер слова конкретной машины, т. е. число разрядов в int. Функция должна быть переносимой, т. е. одна и та же программа должна работать на любых машинах.

Упражнение 3.7. Напишите функцию rightrot(n, b), «вращающую» целое число n вправо на b разрядов.

Упражнение 3.8. Напишите функцию invert(x, p, n), инвертирующую (т. е. заменяющую 1 на 0 и обратно) n разрядов из x, начинающиеся с p-й позиции. Оставшиеся разряды остаются без изменения.

3.10. ОПЕРАЦИИ ПРИСВАИВАНИЯ И ВЫРАЖЕНИЯ

Выражение вида

$$i = i + 2$$

где левая часть повторяется в правой части, могут записываться в сжатом виде:

$$i += 2$$

с использованием *операции присваивания* +=.

Большинству бинарных операций (подобных + и имеющих левый и правый операнды) соответствуют операции присваивания вида *op*=, где *op* одна из:

```
+   -   *   /   %   <<   >>   &   ^   |
```

Если *e1* и *e2* — выражение, то

$$e1\ op{=}\ e2$$

эквивалентно

$$e1 = (e1)\ op\ (e2)$$

однако *e1* вычисляется лишь единожды. Обратите внимание на скобки вокруг *e2*: присваивание x*=y+1 фактически означает x=x*(y+1), а не x=x*y+1.

В качестве примера познакомимся с функцией bitcount, считающей число «единичных» разрядов* в целом аргументе.

```
bitcount(n)     /* подсчет 1-х разрядов в n  */
unsigned n;
{
     int b;

     for (b = 0; n != 0; n >>= 1)
          if (n & 01)
               b++;
     return(b);
}
```

Кроме краткости, преимущество операций присваивания состоит и в том, что они больше соответствуют природе человеческого мышления. Мы говорим: «добавить 2 к i» или «уменьшить i на 2», а вовсе не «взять i, добавить 2, вернуть результат в i», Так что i + = 2. Кроме того, в случае сложных выражений типа

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

операция присваивания делает программу более легкой для понимания, так как читатель не будет тщательно следить за совпадением двух длинных выражений или интересоваться, почему они не совпадают. К тому же операции присваивания могут даже «помочь» транслятору создавать более эффективную программу.

Мы уже использовали тот факт, что оператор присваивания имеет значение и может встречаться в выражениях. Наиболее популярный пример такого использования:

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

Присваивание с другими операциями присваивания (+=, -= и т. д.) также может встречаться в выражении (хотя это и происходит несколько реже). Тип выражения с присваиванием — тип его левого операнда.

Упражнение 3.9. Если для представления чисел используется дополнительный код, то x&(x — 1) уничтожает самый правый единичный разряд в x. (Почему?) Используйте это наблюдение и напишите более быстрый вариант функции bitcount.

---

*Так будем называть разряды двоичного представления чисел, содержащие единицу. — *Примеч. пер.*

3.11. УСЛОВНЫЕ ВЫРАЖЕНИЯ

Операторы

```
if (a > b)
    z = a;
else
    z = b;
```

очевидно присваивают z max из a и b. *Условное выражение*, записанное с помощью тернарной операции «?:», открывает другой путь для написания этого и аналогичных действий. В выражении

e1 ? *e2* : *e3*

первым вычисляется *e1*. Если оно отлично от нуля (истина), то затем вычисляется выражение *e2*, и его значение будет значением всего выражения. В противном случае вычисляется *e3*, и оно дает значение выражению. Из *e2* и *e3* вычисляется лишь одно. Таким образом, для присваивания z max из a и b достаточно написать

```
z = (a > b) ? a : b;      /* z = max(a, b) */
```

Следует заметить, что условные выражения действительно выражения; их можно использовать так же, как и другие выражения. Если *e2* и *e3* различных типов, то тип результата задается правилами преобразования, о которых мы уже говорили ранее. Например, если f типа float, а n типа int, то выражение

```
(n > 0) ? f : n
```

будет типа double вне зависимости от того, положительное n или нет.

Скобки вокруг первого выражения в условном выражении ставить не обязательно, так как приоритет ?: очень низкий, ниже он только у присваивания. Однако мы их рекомендуем всегда ставить, так как при этом условие зрительно выделяется.

Условные выражения часто приводят к более краткой программе. Вот, например, цикл печати N элементов массива по 10 чисел в строке; колонка от колонки отделяется одним пробелом, а каждая строка (включая и последнюю) заканчивается ровно одним переходом на новую строку.

```
for (i = 0; i < N; i++)
   printf("%6d%c", a[i], (i%10==9 || i==N-1) ? '\n' : ' ');
```

Переход на новую строку «печатается» после каждого десятого элемента и после n-го. За любыми другими элементами следует один пробел. Хотя эта программа и выглядит как трюкачество, попытайтесь написать ее без условного выражения, это полезно.

Упражнение 3.10. Напишите функцию lower, переводящую буквы верхнего регистра в буквы нижнего регистра. Вместо if-else используйте условные выражения.

3.12. ПРИОРИТЕТЫ И ПОРЯДОК ВЫЧИСЛЕНИЙ

В приведенной ниже таблице сведены приоритеты операций и порядок их выполнения; причем сюда включены и операции, о которых мы еще не говорили. Операции, перечисленные в одной строке, имеют одинаковый приоритет. Строки расположены в порядке убывания приоритетов, так что, скажем, *, / и % имеют один и тот же приоритет, и он выше приоритета для + и −.

Операция	Порядок
() [] -> .	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
= += -= etc.	справа налево
, (Chapter 3)	слева направо

Операции — > и . используются для доступа к элементам записей, о них речь пойдет в гл. 7, там же будет обсуждаться и операция sizeof (размер объекта). В гл. 6. рассматриваются операции * (косвенность) и & (адрес).

Обратите внимание, что приоритет поразрядных логических операций, &, ^ и | ниже приоритета = =и !=. Это предполагает, что в выражениях, связанных с проверкой разрядов, вроде такого:

```
if ((x & MASK) == 0) ...
```

для получения ожидаемого результата нужно точно расставлять скобки.

Как уже отмечалось ранее, выражения, включающие одну из ассоциативных или коммутативных операций (*, +, &, ^, |), даже при наличии скобок могут быть переупорядочены. В большинстве случаев это не имеет значения, но в тех ситуациях, где это может оказаться существенным, для фиксации конкретного порядка вычислений можно использовать явную промежуточную переменную.

Как и в большинстве языков в Си не задан порядок вычислений операндов для операции. Например, в выражении

```
x = f() + g();
```

f может вычисляться ранее g и наоборот; таким образом, если f или g изменяют некоторую внешнюю переменную, от которой каким-то образом зависит другая функция, то может оказаться, что x зависит от порядка вычисления. В этом случае опять же для задания определенного порядка выполнения промежуточный результат можно сохранить во «временной» переменной.

Аналогично не фиксируется и порядок вычисления аргументов функций, поэтому оператор

```
printf("%d %d\n", ++n, power(2, n));      /* НЕВЕРНО */
```

на различных машинах может давать различные результаты, в зависимости от того, будет ли n увеличиваться до или после обращения к функции power. Правильно следовало бы написать так:

```
++n;
printf("%d %d\n", n, power(2, n));
```

Обращение к функциям, вложенные операторы присваивания, операции увеличения и уменьшения обладают «побочным эффектом», т. е. в результате вычисления таких выражений могут изменяться некоторые переменные. В любом выражении, включающем побочный эффект, могут обнаружиться довольно тонкие ситуации, связанные с порядком записи в переменные, фигурирующие в этом выражении. Вот пример наиболее типичной из таких ситуаций:

```
a[i] = i++;
```

Спрашивается: индексирование идет старым значением i или новым? Транслятор может работать по-разному, и результаты будут разными. Если возникнут побочные эффекты (присваивания действительным переменным), то лучше оставлять их на усмотрение трансляторов, так как выбор наилучшего порядка зависит от архитектуры машины.

Мораль, которую можно вынести из приведенного выше обсуждения, заключается в том, что писать программы, зависящие от порядка выполнения выражений, в любом языке плохо. Естественно, что необходимо знать подобные вещи, чтобы избегать их, но если вы не знаете, *как* это делается на различных машинах, то ваша невинность поможет вам избежать неприятностей. (Верификатор для Си под названием lint выделяет большинство зависимостей от порядка вычислений.)

Глава 4. УПРАВЛЕНИЕ

Операторы управления задают порядок выполнения вычислений. С большинством наиболее распространенных конструкций, управляющих прохождением программы, мы уже встречались в предыдущих примерах. Сейчас мы завершим этот список и дадим более точное описание работы уже упоминавшихся операторов.

4.1. ОПЕРАТОРЫ И БЛОКИ

Любое выражение вроде x=0 или i++ или printf(...) становится оператором, если за ним идет точка с запятой:

```
x = 0;
i++;
printf(...);
```

В нашем языке точка с запятой заканчивает оператор, а не отделяет один оператор от другого, как это делается в алголоподобных языках.

Фигурные скобки { и } употребляются для группирования описаний и операторов в один *составной оператор* или *блок*, в этом случае они синтаксически эквивалентны одному единственному оператору. Скобки, обрамляющие операторы функции,— это один из очевидных примеров. Можно еще упомянуть скобки вокруг последовательности операторов после if, else, while или for. Фактически переменные можно описывать внутри любого блока; об этом мы поговорим в гл. 5. После правой фигурной скобки в конце блока точка с запятой не ставится.

4.2. IF-ELSE

Условный оператор if-else употребляется для принятия решения. Формально синтаксис оператора таков:

```
if (выражение)
    оператор-1
else
    оператор-2
```

где часть с else может присутствовать, а может и не присутствовать. Сначала вычисляется *выражение;* если оно «истина» (т. е. значение выражения отлично от нуля), то выполняется *оператор-1*. Если значение — «ложь» (выражение равно нулю) и есть часть со словом else, то выполняется *оператор-2*.

Так как if просто проверяет числовое значение выражения, то в некоторых случаях это позволяет сокращать программу. Наиболее очевидный прием писать

if (выражение)

вместо

if (выражение != 0)

Иногда это естественно и ясно, а иногда выглядит как криптограмма.

Поскольку часть с else возможна, но не обязательна, то, если она пропускается во вложенных условных операторах, возникает неясность в толковании. Эту неясность решают обычным способом: else сопоставляют с ближайшими if, не имеющим своего else. Например, в

```
if (n > 0)
     if (a > b)
          z = a;
     else
          z = b;
```

else относится к внутреннему if, что мы и показали с помощью отступов. Если при таком толковании получается не то, что вам нужно, то для введения требуемого порядка нужно использовать фигурные скобки.

```
if (n > 0) {
     if (a > b)
          z = a;
}
else
     z = b;
```

Неясности особенно неприятны в ситуациях типа такой:

```
if (n > 0)
     for (i = 0; i < n; i++)
          if (s[i] > 0) {
               printf("...");
               return(i);
          }
else      /* НЕВЕРНО*/
     printf("error - n is zero\n");
```

Отступы однозначно показывают, что мы хотим. Однако транслятор в этих сообщениях не разбирается и связывает else с внутренним if. Ошибки такого рода находить очень трудно.

Между прочим, обратите внимание, что в

```
if (a > b)
     z = a;
else
     z = b;
```

после z = a стоит точка с запятой. Здесь все грамматически верно, так как за if следует оператор, а операторы-выражения вроде z = a всегда заканчиваются точкой с запятой.

4.3. ELSE-IF

Конструкция

```
if (выражение)
    оператор
else if (выражение)
    оператор
else if (выражение)
    оператор
else
    оператор
```

встречается настолько часто, что о ней стоит поговорить особо. Такая последовательность if — наиболее общий способ описать принятие не одного из двух, а одного из многих решений. *Выражения* вычисляются по порядку; если какое-либо *выражение* истинно, то выполняется сопоставленный с ним *оператор,* и на этом заканчивается выполнение всей цепочки. Каждый оператор — это либо один единственный оператор, либо группа операторов, заключенная в фигурные скобки.

Последняя часть с else обрабатывает случай, не сопоставленный ни с одним из перечисленных выше. Это «аварийный» вариант, на случай, если ни одно из условий не было выполнено. Иногда в этом случае нет никакого явного действия и заключительное

```
else
    оператор
```

можно опустить или использовать его для обнаружения ошибки, фиксируя «невозможное» условие.

В качестве примера «тройного» ветвления рассмотрим функцию двоичного поиска*, определяющую, встречается ли значение x в упорядоченном массиве v. Элементы v должны быть расположены в возрастающем порядке. Функция возвращает положение x в v (число между 0 и n — 1), если оно там встречается, и — 1, если его нет.

```
binary(x, v, n)      /* поиск x в v[0] ... v[n-1] */
int x, v[], n;
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else /* совпадение */
            return(mid);
    }
    return(-1);
}
```

Основное решение, которое нужно принять на каждом шагу,— меньше ли x, чем средний элемент v[mid], больше ли или равен ему. Это естественное применение else-if.

---

*Такой поиск часто называют поиском «методом деления пополам». — *Примеч. пер.*

Переключатель (оператор switch) специально предназначен для описания принятия одного из многих решений. В нем проверяется совпадение значения выражения с одной из нескольких констант и выполняется соответствующая «ветвь». В гл. 2 мы уже приводили программу для подсчета вхождения каждой из цифр, пробелов и любых других символов. Тогда мы использовали последовательность if ... else if ... else. Теперь приведем ту же программу, но с переключателем:

```
main()      /* подсчет цифр, пустых символов и пр. */
{
     int c, i, nwhite, nother, ndigit[10];

     nwhite = nother = 0;
     for (i = 0; i < 10; i++)
          ndigit[i] = 0;

     while ((c = getchar()) != EOF)
          switch (c) {
          case '0':
          case '1':
          case '2':
          case '3':
          case '4':
          case '5':
          case '6':
          case '7':
          case '8':
          case '9':
               ndigit[c-'0']++;
               break;
          case ' ':
          case '\n':
          case '\t':
               nwhite++;
               break;
          default:
               nother++;
               break;
          }

     printf("digits =");
     for (i = 0; i < 10; i++)
          printf(" %d", ndigit[i]);
     printf("\nwhite space = %d, other = %d\n",
          nwhite, nother);
}
```

В операторе switch вычисляется целое выражение в скобках (в данном случае это c — символ), и его значение сравнивается со всеми вариантами. Каждый вариант должен быть помечен целой или символьной константой или константным выражением. Вариант со словом default выполняется, если ни один из вариантов не подошел. Default лишь допускается; если такого варианта нет и нет совпадения ни с одним из вариантов, то не выполняется никаких действий. Помеченные варианты и вариант с default могут идти в любом порядке. Все варианты должны быть различными.

Оператор break (разрыв) вызывает немедленный выход из переключателя. Так как варианты представляют собой только некоторые метки специального вида, то после обработки программы одного варианта, если не предпринять явных действий, чтобы прервать этот процесс, начнется обработка очередного фрагмента. Чаще всего переключатель покидают с помощью операторов break и return. Оператор разрыва употребляют и для немедленного выхода из циклических конструкций со словами while, for и do; об этом мы еще будем говорить в данной главе.

Проход через варианты носит противоречивый характер. С одной стороны, это полезно, так как позволяет сводить многие варианты в одно действие. Так делается в нашем примере в случае пробелов, табуляций или переходов на новую строку. Но это же и предполагает, что обычно каждый вариант должен заканчиваться оператором разрыва, предохраняющим от «прохода» в следующий вариант. Проход из одного варианта в другой — вещь ненадежная; в случае модификации программы это может привести к развалу ее работы. За исключением многих меток для одного вычисления, проходом через варианты следует пользоваться весьма осторожно.

Обычно принято ставить оператор разрыва после последнего варианта (на месте default), даже если в этом нет необходимости. Однажды эта мелочь сослужит вам хорошую службу, когда придется добавлять еще один вариант.

Упражнение 4.1. Напишите функцию expand(s, t), преобразующую символы вроде новой строки и табуляции в видимые комбинации \n и \t по мере копирования строки s в t. Используйте переключатель.

4.5. ЦИКЛЫ WHILE и FOR

С циклами while и for мы уже встречались. В операторе

while (*выражение*)
оператор

вычисляется *выражение*. Если оно отлично от нуля, выполняется *оператор* и повторяется вычисление *выражения*. Такое циклическое выполнение продолжается до тех пор, пока *выражение* не станет нулем; в этот момент начнет исполняться то, что стоит после оператора.

Оператор

for (*выраж 1; выраж 2; выраж 3*)
оператор

эквивалентен

выраж 1;
while (*выраж 2*) {
оператор
выраж 3;
}

Грамматически три компоненты оператора for суть выражения. Чаще всего *выраж1* и *выраж3* — присваивания или обращения к функциям, а *выраж2* — выражение-отношение. Любая из трех частей может быть пропущена, хотя точка с запятой должна остаться. Если нет *выраж1* или *выраж3*, то параметр цикла i просто выпадает из рассмотрения. Если же нет проверки *выраж2*, то считается, что она постоянно дает истину. Таким образом,

```
for (;;) {
    ...
}
```

«бесконечный» цикл, из которого, по-видимому, выходят другими способами (такими, как разрыв или возврат).

Использовать ли оператор while или оператор for — во многом дело вкуса. Например, в цикле

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;     /*        пропуск пустых символов          */
```

нет никакой инициации и изменения параметра, поэтому while выглядит вполне естественно.

Оператор же for представляется более очевидным, если есть простая инициация и изменение параметра; в этом случае операторы управления циклом собраны в начале цикла и они «видны». Например, начало

```
for (i = 0; i < N; i++)
```

в языке Си типично для обработки первых N элементов массива, это аналог цикла в Фортране или ПЛ/1. Однако аналог не совсем точный, так как пределы цикла for можно менять внутри цикла, и, кроме того, как бы ни закончился цикл, параметр цикла сохраняет свое значение. Поскольку компоненты оператора for — произвольные выражения, то цикл for не ограничен только арифметическими прогрессиями. Хотя это и допустимо, но включение в заголовок посторонних вычислений считается плохим стилем; лучше его сохранять для операций по управлению циклом.

В качестве большего примера приведем другую версию программы atoi, преобразующей строку в числовой эквивалент. Данная версия более общая, она работает с пустыми символами в начале и допускает знаки + или −. (В гл. 5 будет приведена функция atof, проделывающая такие же преобразования для чисел с плавающей запятой.)

Структура программы отражает вид ее входной информации:

пропускать пустые, если они есть
взять знак, если он есть
взять целую часть, преобразовать ее

Каждый этап выполняет свои действия, и при переходе к следующему этапу все четко фиксируется. Весь процесс оканчивается на первом символе, который не может быть частью числа:

```
atoi(s)    /* перевод s в целое */
char s[];
{
      int i, n, sign;

      for (i=0; s[i]==' ' || s[i]=='\n' || s[i]=='\t'; i++)
            ;     /* пропуск пустых символов */
      sign = 1;
      if (s[i] == '+' || s[i] == '-')     /* знак  */
            sign = (s[i++]=='+') ? 1 : -1;
      for (n = 0; s[i] >= '0' && s[i] <= '9'; i++)
            n = 10 * n + s[i] - '0';
      return(sign * n);
}
```

Преимущества от централизованного управления циклом становятся еще более очевидными, если речь идет о нескольких вложенных циклах. Следующая функция, shell, упорядочивает массив целых чисел. Основная идея такой сортировки заключается в том, что на ранних этапах сравниваются элементы, расположенные далеко друг от друга, а не соседние, как в обычных перестановочных сортировках. Это приводит к быстрому исключению «большой» неупорядоченности, поэтому на более поздних этапах ра-

боты становится меньше. Интервалы между сравниваемыми элементами постепенно уменьшаются до единицы, т. е. до момента, когда сортировка сводится к обычным перестановкам соседних элементов.

```
shell(v, n)  /* сортировка v[0]...v[n-1] в возрастающем порядке */
int v[], n;
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

Здесь три вложенных цикла. Самый внешний управляет интервалом (gap) между сравниваемыми элементами, начиная с n/2 и уменьшая его в два раза, пока он не станет нулем. Средний цикл сравнивает каждую пару элементов, разделенных интервалом, а самый внутренний цикл переставляет элементы, не стоящие по порядку. Так как интервал в конце концов уменьшается до единицы, то и все элементы в заключение будут упорядочены. Обратите внимание, что универсальность цикла for позволяет сделать внешний цикл по форме похожим на другие, хотя он и не связан с арифметической прогрессией.

В Си есть еще операция "," (запятая), чаще всего она используется в операторе for. Пара выражений, разделенных запятой, вычисляется слева направо, а тип и значение результата суть тип и значение правого операнда. Таким образом в операторе for можно в каждой из частей помещать по нескольку выражений, например, для параллельной обработки двух индексов. В качестве примера приведем функцию reverse(s), «на месте» обращающую строку s:

```
reverse(s)      /* поворот строки s на месте */
char s[];
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Запятая, разделяющая аргументы функций, переменные в описаниях и т. п., не есть операция «запятая», и она не гарантирует вычисление слева направо.

Упражнение 4.2. Напишите функцию expand(s1, s2), превращающую сокращенную запись вида a—z в строке s1 в эквивалентный полный список abc ... xyz в s2. Допускаются буквы на двух регистрах и цифры. Надо быть готовыми обрабатывать варианты вроде a—b—c, a—z0—9 и —a—z. Полезно считать, что начальные или конечные символы «—» воспринимаются буквально.

В циклах while и for желаемые атрибуты проверок условия окончания помещаются в начале, а не в конце цикла. Об этом мы уже говорили в гл. 2. В третьем виде циклов, do-while, проверка идет в конце, уже после прохода по телу цикла, т. е. тело всегда выполняется по крайней мере один раз. Синтаксис этого цикла таков:

```
do
    оператор
while (выражение);
```

Выполняется *оператор*, а затем вычисляется *выражение*. Если оно истинно, то снова выполняется оператор, и т. д. Если выражение становится ложным, то повторения заканчиваются.

Можно ожидать, что комбинация do-while используется значительно реже, чем циклы с while и for, — таких циклов, наверное, около 5% от всех. Однако время от времени оказывается, что их полезно использовать, например, в функции itoa, преобразующей число в строку символов (обратную по отношению к atoi). Работа такой функции оказывается несколько более сложной, чем можно было ожидать, из-за того, что удобный способ формирования цифр порождает их в «плохом» порядке. Поэтому мы формируем строку в обратном порядке, а затем обращаем ее:

```
itoa(n, s)      /* перевод n в символы строки s */
char s[];
int n;
{
    int i, sign;

    if ((sign = n) < 0) /* запись знака */
        n = -n;         /* сделать n положительным */
    i = 0;
    do {        /* формирование цифр в обратном порядке */
        s[i++] = n % 10 + '0';    /* взять очередную цифру */
    } while ((n /= 10) > 0); /* убрать ее */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

Конструкция do-while нужна, или по крайней мере удобна, так как в массиве s должен быть хотя бы один символ, каким бы ни было значение n. Кроме того, мы ставим единственный оператор тела цикла do-while в скобки, хотя в этом и нет необходимости, чтобы неискушенный читатель не принимал while за *начало* цикла while.

Упражнение 4.3. Наша версия функции itoa не обрабатывает самое большое отрицательное число, если для представления чисел используется дополнительный код (значение такого числа есть — $2^{\text{размер слова}-1}$). Объясните, почему это так. Модифицируйте программу, чтобы она печатала верное значение вне зависимости от того, на какой машине она работает.

Упражнение 4.4. Напишите функцию, аналогичную itob(n, s), преобразующую целое без знака n в двоичное символьное представление s. Напишите itoh, преобразующую целое в шестнадцатеричное представление.

Упражнение 4.5. Напишите версию itoa с тремя аргументами. Третий аргумент — размер минимального поля представления. Преобразованное число должно, если это необходимо, дополняться слева пробелами.

4.7. BREAK (РАЗРЫВ)

Иногда бывает удобно иметь возможность прервать выполнение цикла где-то в произвольном месте, а не только после проверок в начале или в конце его. Оператор break обеспечивает такое преждевременное прекращение циклов со словами for, while и do, так же как и переключателя. Этот оператор вызывает немедленный выход из самого внутреннего из объемлющих его циклов или переключателей.

Приведенная ниже программа выбрасывает излишние символы пробела или табуляции из конца каждой входной строки, при этом оператор break используется для окончания цикла при обнаружении самого правого символа, отличного от пробела или табуляции.

```
#define   MAXLINE   1000

main()      */изъятие пробелов и табуляций */
{
     int  n;
     char line[MAXLINE];

     while ((n = getline(line, MAXLINE)) > 0) {
          while (--n >= 0)
               if (line[n] != ' ' && line[n] != '\t'
                 && line[n] != '\n')
                    break;
          line[n+1] = '\0';
          printf("%s\n", line);
     }
}
```

Программа getline выдает размер строки. Внутренний цикл while начинается с последнего символа в line (напоминаем, что `--n` уменьшает n, прежде чем будет использовано его значение), и идет просмотр строки в обратном направлении в поисках первого символа, который не есть пробел или табуляция. Цикл заканчивается, если таковой будет найден или же n станет отрицательным, т. е. строка будет полностью просмотрена. Вам следует убедиться, что такое поведение будет корректным, даже если строка состоит только из пустых символов.

Вместо оператора break можно включить проверку в сам цикл:

```
while ((n = getline(line, MAXLINE)) > 0) {
     while (--n >= 0
       && (line[n]==' ' || line[n]=='\t' || line[n]=='\n'))
          ;
     ...
}
```

Такой вариант хуже основного, ибо саму проверку тяжелее понимать. Вообще, следует избегать проверок, требующих смешения &&, ||, ! и скобок.

4.8. CONTINUE (ПРОДОЛЖЕНИЕ)

Оператор continue связан с нарушением последовательности действий, но используется реже. Он вызывает начало следующей итерации объемлющего цикла while, for, do. В циклах while и do это означает, что сразу же выполняется проверяющая часть, а в цикле for управление передается на этап повторной инициации. (Оператор continue применим лишь к циклам, но не к переключателям. Если написать его в переключателе, расположенном внутри цикла, то управление перейдет на следующую итерацию.)

Рассмотрим в качестве примера фрагмент обработки только положительных элементов массива; отрицательные элементы пропускаются.

```
for (i = 0; i < N; i++) {
      if (a[i] < 0)   /* пропуск отрицательных элементов */
            continue;
      ...   /* обработка положительных элементов */
}
```

Оператор продолжения часто используется в тех случаях, когда последующая часть цикла сложна, в ней есть проверки, связанные с неявными циклами, и введение новых уровней делает вложенность циклов слишком глубокой.

Упражнение 4.6. Напишите программу, копирующую входную информацию на выход; она должна печатать только одну из идущих подряд идентичных строк. (Это простой вариант служебной программы uniq в системе UNIX.)

4.9. ПЕРЕХОДЫ И МЕТКИ

В языке Си существуют и всячески поносимые переходы и метки для них. Формально в переходах необходимости нет, и на практике почти всегда проще написать программу без них. В этой книге мы переходы не используем.

Однако можно представить себе небольшое число ситуаций, где переходы окажутся на месте. Чаще всего это случается, если нужно выйти из структуры с некоторой глубиной вложенности, например покинуть сразу два цикла. Оператор break непосредственно использовать нельзя, так как он позволяет выйти только из самого внутреннего цикла. Например:

```
for ( ... )
      for ( ... ) {
            ...
            if ( бедствие )
                  goto ошибка ;
      }
...

ошибка:
      устранение последствий
```

Такая организация удобна, если исправление требует нетривиального программирования, а ошибки могут обнаруживаться в нескольких местах. Метка имеет вид обычного имени переменной, за которым следует двоеточие. Метку можно поставить перед любым оператором в той же функции, где находится оператор перехода (со служебным словом goto).

Как пример рассмотрим задачу поиска первого отрицательного элемента в двумерном массиве. (Многомерные массивы разбираются в гл. 6.)

Можно составить такую программу:

```
        for (i = 0; i < N; i++)
             for (j = 0; j < M; j++)
                  if (v[i][j] < 0)
                       goto found;
        /* не найден */
        ...
found:
        /* найден в позиции i, j */
        ...
```

Программы с переходами всегда можно написать и без них. За это, может быть, придется заплатить несколькими повторяющимися проверками или дополнительной переменной. Например, поиск в массиве можно было бы записать и так:

```
found = 0;
for (i = 0; i < N && !found; i++)
     for (j = 0; j < M && !found; j++)
          found = v[i][j] < 0;
if (found)
     /* был в i-1, j-1 */
     ...
else
     /* не найден */
     ...
```

Хотя мы и не догматики в этом вопросе, но нам кажется, что оператор перехода нужно употреблять осмотрительно, если вообще употреблять.

Глава 5. ФУНКЦИИ И СТРУКТУРА ПРОГРАММЫ

Функции разбивают большие вычислительные задачи на более мелкие и позволяют людям строить программы из них более регулярным образом, чем если бы они начинали с бесформенных, случайных соображений. Выбранные соответствующим образом функции часто могут «прятать» детали обработки от тех частей программы, которым нет нужды их знать. Это делает всю программу более ясной и позволяет более безболезненно вносить исправления.

Язык Си проектировался так, чтобы функции были эффективными и легкими в использовании. Обычно программы на этом языке состоят из большого числа небольших функций, а не из нескольких больших. Программа может находиться в одном (это наиболее частый случай) или нескольких файлах. Файлы можно раздельно транслировать и вместе загружать, добавляя предварительно оттранслированные функции из библиотек. В этот процесс мы сейчас не будем углубляться, так как детали его варьируются в соответствии с той или иной конкретной системой.

Большинству программистов знакомы библиотечные функции для ввода и вывода (getchar, putchar) и численных расчетов (sin, cos, sqrt). В этой главе мы более детально обсудим составление новых функций.

5.1. ОСНОВЫ

Для начала спроектируем и напишем программу для печати вводимых строчек, содержащих некоторый отдельный «образ» или же просто строку символов. (Это особый случай служебной программы grep, предусмотренной в системе UNIX.) Например, поиск образа «the» в последовательности строчек:

```
Now is the time
for all good
men to come to the aid
of their party.
```

даст на выходе

```
Now is the time
men to come to the aid
of their party.
```

Принципиально структура этой работы четко делится на три части:

while (*есть другие строки*)
    if (*строка содержит образ*)
        напечатать ее

Хотя можно, конечно, запрограммировать все это в одной программе, однако лучше воспользоваться естественностью структуры и выиграть, написав каждую часть как отдельную функцию. С тремя небольшими частями легче иметь дело, чем с одной большой программой, так как не относящиеся к делу детали можно скрыть в функциях и тем самым уменьшить риск нежелательного взаимодействия. Кроме того, части могут оказаться полезными и сами по себе.

«Пока есть другие строки» — это функция getline, построенная нами в гл. 2, а «напечатать ее» — функция printf, уже составленная для нас кем-то другим. Это означает, что нам надо написать только подпрограмму для определения, содержится ли данный образ в строке. Мы можем решить эту задачу, позаимствовав ее конструкцию из ПЛ/1: функция index(s, t) выдает положение или индекс в строке s, начиная с которого размещается строка t, если же s не содержит t, то выдается — 1. В качестве начальной позиции мы используем не 1, а 0, так как в языке массивы начинаются с нулевой позиции. Если нам позже понадобится более сложная программа поиска образа, то нам нужно будет только заменить index, а все другое в программе останется тем же.

После такого «проектирования» детализация программы уже совсем простое дело. Ведь перед нами нечто целое, и поэтому мы уже можем разобраться как части подогнаны друг к другу. Сейчас отыскиваемый образ представляет собою буквальную строку, заданную как аргумент для функции index; конечно, это отнюдь не самый общий механизм его задания. Мы еще вернемся, коротко, к обсуждению вопроса, как инициировать массив символов, и в гл. 6 покажем, как организовать передачу в качестве параметра образа, формируемого во время выполнения программы. Мы приводим здесь и новый вариант функции getline; поучительно сравнить ее с программой, приведенной в гл. 2.

```
#define   MAXLINE   1000

main()    /* поиск всех строчек, совпадающих с образом */
{
     char line[MAXLINE];

     while (getline(line, MAXLINE) > 0)
          if (index(line, "the") >= 0)
               printf("%s", line);
}

getline(s, lim)      /* записать строчку в s, вернуть длину */
char s[];
int lim;
{
     int c, i;

     i = 0;
     while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
          s[i++] = c;
     if (c == '\n')
          s[i++] = c;
     s[i] = '\0';
     return(i);
}
```

```
index(s, t)     /* выдать начало t в s, если нет, то – 1 */
char s[], t[];
{
     int i, j, k;

     for (i = 0; s[i] != '\0'; i++) {
          for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
               ;
          if (t[k] == '\0')
               return(i);
     }
     return(-1);
}
```

Каждая функция имеет такой вид:

имя (список аргументов, если они есть)
описания аргументов, если они есть
{
    описания и операторы, если они есть
}

Предполагается, что любая из частей может отсутствовать; минимальная функция

```
dummy() {}
```

не делает ничего. Такие пустые функции иногда полезны как «хранители места» в процессе создания программы. Если функция возвращает значение, отличное от целого, то перед именем функции может стоять некоторый тип. Об этом мы поговорим в следующем разделе.

Любая программа есть просто множество определений отдельных функций. Связь между функциями идет с помощью аргументов и возвращаемых функциями значений (как в данном случае); но она может идти и через внешние переменные. Во входном файле функции идут в любом порядке, а сама входная программа может быть разделена на много файлов; саму же функцию разделять на несколько файлов пока нельзя. Передача значения из вызванной функции в вызывавшую происходит с помощью оператора возврата. Следом за словом return идет любое выражение:

return *(выражение)*

Если это нужно, то вызвавшая функция может игнорировать возвращаемое значение. Более того, после слова return можно вообще не ставить никакого выражения, в этом случае вызвавшей функции никакое значение не передается. Управление возвращается в вызывавшую функцию и в случае выхода «по концу», т. е. если функция доходит до закрывающей правой скобки. Значение при этом не передается. Функция может выдавать значение при возврате из одного места и не выдавать при возврате из другого. Это не запрещено, но, вероятно, это признак какой-то оплошности. Во всяком случае, «значение» функции, не выданное ею, просто какой-то «мусор». Верификатор lint для языка Си обнаруживает такие ошибки.

Способы трансляции и загрузки программ, находящихся в нескольких файлах, от системы к системе меняются. В системе UNIX, например, эта работа выполняется с помощью команды cc, упомянутой в гл. 2. Предположим у нас есть три функции в трех файлах с именами main.c, getline. c и index.c. В этом случае команда

cc main.c getline.c index.c

транслирует эти три файла, помещает полученные перемещаемые модули в файлы main.o, getline.o и index.o и загружает их все в некоторый выполняемый файл, называемый a.out.

Если, скажем, в main.c встретилась ошибка, то файл можно повторно оттранслировать, а результат загрузить вместе с файлом модулей, полученными перед этим, что делается командой

cc main.c getline.o index.o

Версии имен «.c» и «.o» используются в команде cc для того, чтобы отличать исходные файлы от файлов модулей.

Упражнение 5.1. Напишите функцию rindex(s, t), выдающую положение *самого правого* вхождения t в s; если вхождений нет—выдается—1.

5.2. ФУНКЦИИ, ВЫДАЮЩИЕ НЕ ЦЕЛЫЕ ЗНАЧЕНИЯ

Пока что ни одна из наших программ не имела какого-либо описания типа функции, так как по умолчанию функция неявно описывается своим вхождением в выражение или оператор, например таким:

```
while (getline(line, MAXLINE) > 0)
```

Если не описанное ранее имя встречается в выражении и за ним следует левая скобка, то контекстуально оно описывается как имя функции. Более того, считается, что эта функция возвращает величину типа int. Так как тип char в выражениях приводится к int, то описывать функцию как возвращающую значение типа char не имеет смысла. Такие допущения покрывают большинство случаев, включая и все приведенные нами до этого момента примеры.

Но что же делать, если нужно, чтобы функция возвращала значение какого-либо другого типа? Многие числовые функции вроде sqrt, sin и cos возвращают величины двойной точности, а другие специализированные функции — другие типы. Для знакомства с такими функциями давайте напишем, а потом воспользуемся ею, функцию atof(s), преобразующую строку s в эквивалентное ей число с плавающей точкой и двойной точностью. Функция atof представляет собою расширение функции atoi (см. гл. 3 и 4). Она обнаруживает возможный знак числа и десятичную точку в присутствии или отсутствии как целой, так и дробной частей. (Это *не* подпрограмма ввода высокого уровня, такая подпрограмма занимала бы значительно больше места, чем мы намерены использовать.)

Во-первых, программа atof «сама» должна описать, величины какого типа она возвращает, поскольку это не величины типа int. Так как в выражениях float преобразуется в двойную точность, то не имеет смысла говорить, что функция возвращает float*, лучше воспользоваться дополнительной точностью, поэтому мы пишем, что она возвращает величину двойной точности. Имя типа ставится перед именем функции, как в примере:

```
double atof(s) /* перевод строки s в число двойной точности */
char s[];
{
     double val, power;
     int  i, sign;

     for (i=0; s[i]==' ' || s[i]=='\n' || s[i]=='\t'; i++)
          ;          /* пропуск пустых символов */
     sign = 1;
     if (s[i] == '+' || s[i] == '-')     /* знак */
          sign = (s[i++]=='+') ? 1 : -1;
```

* Если описать функцию как float, то при возврате точность уменьшится и последующее преобразование в double потерянной точности не вернет! — *Примеч. пер.*

```
    for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
        val = 10 * val + s[i] - '0';
    if (s[i] == '.')
        i++;
    for (power = 1; s[i] >= '0' && s[i] <= '9'; i++) {
        val = 10 * val + s[i] - '0';
        power *= 10;
    }
    return(sign * val / power);
}
```

Во-вторых, и это очень важно, вызывающая подпрограмма должна указать, что atof возвращает не целое значение. Такое описание приведено в программе примитивного настольного вычислителя, она считывает по одному числу на строку, возможно, со знаком, складывает их все и печатает после каждого ввода сумму:

```
#define   MAXLINE   100

main()    /* простейший калькулятор */
{
    double sum, atof();
    char line[MAXLINE];

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%.2f\n", sum += atof(line));
}
```

Описание

```
double sum, atof();
```

указывает, что sum — переменная типа double, а atof — функция, возвращающая величину двойной точности. Такая запись мнемонична: она предполагает, что и sum и atof(...) — величины с плавающей точкой и двойной точности.

Если atof не будет явно описана и там, и там, то транслятор предположит, что функция возвращает целое значение, и мы получим глупый ответ. Если тип в самой atof не совпадает с типом, указанным в вызывающей ее подпрограмме, причем они находятся в одном файле, транслятор обнаружит ошибку. Но если, а чаще всего так и бывает, функция atof транслировалась отдельно, то несовпадение обнаружено не будет, atof будет возвращать значение двойной точности, которое главная программа будет интерпретировать как int. Результат в таком случае бессмыслен. Программа lint эту ошибку зафиксирует.

Написав atof, мы могли бы в принципе с ее помощью написать и atoi (строка преобразуется в целое):

```
atoi(s)    /* перевод строки s в целое */
char s[];
{
    double atof();

    return(atof(s));
}
```

Обратите внимание на структуру описаний и оператора возврата. Значение выражения в

return *(выражение)*

всегда преобразуется к типу функции, и только после этого следует возврат. Поэтому значение atof (двойной точности) автоматически преобразуется в int, если оно входит в оператор возврата, поскольку atoi возвращает int. (Преобразование значения с плавающей точкой в целое отбрасывает любую дробную часть; об этом уже говорилось в гл. 3.)

Упражнение 5.2. Расширьте atof, чтобы она воспринимала запись чисел в нотации, принятой в научной литературе, т. е. в виде 123.45e—6 Здесь за числом с плавающей точкой может следовать e или E и, возможно, со знаком порядок.

5.3. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ ОБ АРГУМЕНТАХ

В гл. 2 мы уже говорили, что аргументы функции передаются по значению, т. е. вызванная функция получает свою собственную, временную копию каждого аргумента, а не его адрес. Это означает, что функция не может воздействовать на сам оригинальный аргумент в вызвавшей ее программе. Фактически внутри функции каждый аргумент есть просто локальная переменная, инициированная значением, с которым к этой функции обратились.

Если в качестве аргумента функции фигурирует имя массива, то передается начало массива, сами же элементы не копируются. Функция может изменять элементы массива, сдвигаясь (индексированием) от этого начала. Фактически массив передается как бы ссылкой. В гл. 5 мы будем говорить о том, как использовать ссылки, чтобы функция воздействовала в вызывающей функции на объекты, отличные от массивов.

Далее, не существует вполне удовлетворительного способа, чтобы писать переносимые функции, воспринимающие переменное число аргументов, так как для вызванной функции нет переносимого механизма, с помощью которого она могла бы определить, сколько именно аргументов ей передано в данном обращении. Таким образом, мы, например, не можем написать действительно переносимую функцию, вычисляющую максимум из произвольного числа аргументов, как это делают встроенные функции MAX в Фортране или ПЛ/1.

В переменном числе аргументов нет ничего страшного, если вызванная функция не использует аргументов, которые не были фактически заданы, и если есть подобающие типы. В языке Си наиболее употребительной функцией с переменным числом аргументов является printf: в ней используется информация из первого аргумента, позволяющая определить, сколько в данном обращении аргументов и каковы их типы. Будет очень плохо, если обратившийся не задаст достаточного числа аргументов или их типы не будут такими, как их определяет первый аргумент.

Можно поступать и по-другому. Если типы аргументов известны, то маркировать каким-то образом конец списка аргументов, скажем, использовать для указания на последний аргумент какое-то специальное значение аргумента (часто это нуль).

5.4. ВНЕШНИЕ ПЕРЕМЕННЫЕ

Любая программа на языке Си есть множество внешних объектов, либо переменных, либо функций. Определение «внешние» используется главным образом для того, чтобы подчеркнуть отличие от «внутренних» объектов — аргументов и автоматических переменных, определяемых внутри функции. Внешние переменные определены вне любой из функций и, следовательно, доступны для многих функций. Сами функции всегда внешние, так как язык не позволяет определять функции внутри других функций.

Подразумевается, что внешние переменные являются так же и глобальными, поэтому все ссылки к любой такой переменной через некоторое имя (даже если функция оттранслирована отдельно) относятся к одному и тому же объекту. В этом смысле внешние переменные аналогичны фортранным блокам COMMON или EXTERNAL в ПЛ/1. Позже мы увидим, как нужно определить внешнюю переменную или функцию, чтобы она не была глобально доступной и к ней можно было бы обращаться только из одного единственного файла (она была бы видима в нем).

Поскольку внешние переменные доступны везде, их можно использовать для передачи аргументов функциям и для получения от них результатов, т. е. это еще один механизм связи между функциями. К внешней переменной имеет доступ любая функция, она может ссылаться на эту переменную с помощью имени, если оно уже было где-то описано.

Если взаимодействие функций связано с большим числом переменных, внешние переменные представляются более удобным и эффективным средством, чем длинные списки аргументов. Однако мы уже говорили в гл. 2, что этими соображениями нужно пользоваться с осторожностью, так как это плохо сказывается на структуре программы и приводит к программам со многими связями (по данным) между функциями.

Вторая причина использования внешних переменных связана с инициацией. В частности, внешние массивы можно инициировать, а автоматические массивы — нет. В конце этой главы мы будем говорить об инициации.

Третья причина употребления внешних переменных кроется в их области действия и времени существования. Автоматические переменные по отношению к функциям являются внутренними: они начинают существовать при входе в подпрограмму, а при выходе из нее они уничтожаются. Внешние же переменные постоянные, они не возникают и не уничтожаются и поэтому сохраняют значения от одного обращения до следующего. Таким образом, если две функции должны иметь общие данные, причем ни одна из них не обращается к другой, то эти данные удобнее всего хранить во внешних переменных, а не передавать через аргументы.

Давайте теперь дополнительно разберем большой пример. Нужно написать еще одну программу калькулятора, лучше чем до этого. Программа будет допускать операции +, —, *, / и = (для печати ответа). Чтобы ее легче было реализовать, наш калькулятор будет использовать для выражений обратную польскую запись, а не традиционную инфиксную. (Обратная польская запись используется, например, в карманных калькуляторах фирмы Hewlett-Packard.) При такой нотации каждая операция идет следом за своими операндами и, например, инфиксное выражение

$$(1 - 2) * (4 + 5) =$$

вводится в таком виде:

$$1\ 2\ -\ 4\ 5\ +\ *\ =$$

Скобки здесь не нужны.

Реализация очень простая. Каждый операнд помещается в стек; если приходит операция, из стека выбирается («поднимается») нужное число операндов (для бинарной операции — два), к ним применяется операция и результат вновь «опускается» в стек. В нашем примере 1 и 2 помещаются в стек, затем они заменяются их разностью, —1. Следом за этим в стек опускаются 4 и 5 и заменяются на их сумму, 9. Далее, вместо —1 и 9 в стеке оказывается их произведение, —9. Операция = печатает верхний элемент, не убирая его (поэтому можно следить за промежуточными шагами вычисления).

Действия по опусканию в стек и подниманию величины из стека сами тривиальны, но по мере добавления к ним *обнаружения ошибок* и исправления таковых становятся достаточно длинными. Поэтому луч-

ше не повторять соответствующие части по всей программе, а сосредоточить действия в отдельных функциях. И конечно, нужны отдельные функции для считывания следующего операнда или операции. Таким образом, структура программы выглядит так:

```
while (следующая операция или операнд не есть конец файла)
if (число)
    вниз его
    else if (операция)
        вверх операнды
        сделать операцию
        вниз результат
    else
        ошибка
```

Основное, что надо решить при конструировании программы и о чем мы еще не говорили,— где находится стек и как он непосредственно доступен программам. Один из вариантов — сохранять стек в главной программе и передавать его и текущую позицию в нем программам, передающим в него информацию (push) и извлекающим ее (pop). Но у главной функции нет необходимости знать что-либо о переменных, управляющих стеком: она должна «мыслить» понятиями «поместить» и «извлечь». Поэтому мы решили, что стек и связанная с ним информация будут внешними переменными, доступными для функций push и pop, но недоступными для функции main.

Превращение этого наброска в программу достаточно простое. Программа main, по существу, большой переключатель по типу операции или операнду. Это, наверное, более типичный случай использования переключателя, чем приведенный в гл. 4:

```
#define MAXOP   20    /* максимальный размер операнда, оператора */
#define NUMBER  '0'   /* признак прихода числа */
#define TOOBIG  '9'   /* признак очень большой строки */

main()    /* калькулятор для польской записи */
{
    int  type;
    char s[MAXOP];
    double op2, atof(), pop(), push();

    while ((type = getop(s, MAXOP)) != EOF)
        switch (type) {

        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
```

```
                else
                    printf("zero divisor popped\n");
                break;
            case '=':
                printf("\t%f\n", push(pop()));
                break;
            case 'c':
                clear();
                break;
            case TOOBIG:
                printf("%.20s ... is too long\n", s);
                break;
            default:
                printf("unknown command %c\n", type);
                break;
            }
    }

    #define MAXVAL  100   /* максимальная глубина стека значений */

    int sp = 0;     /* указатель стека */
    double val[MAXVAL]; /* стек значений */

    double push(f) /* запись f в стек значений */
    double f;
    {
        if (sp < MAXVAL)
            return(val[sp++] = f);
        else {
            printf("error: stack full\n");
            clear();
            return(0);
        }
    }

    double pop()    /* взять из стека верхнее значение */
    {
        if (sp > 0)
            return(val[--sp]);
        else {
            printf("error: stack empty\n");
            clear();
            return(0);
        }
    }

    clear()    /* очистить стек */
    {
        sp = 0;
    }
```

Команда c очищает стек, обращаясь к функции clear, с ней работают и push, и pop, если они обнаруживают ошибку. К функции getop мы еще вернемся.

В гл. 2 мы уже упоминали, что переменная будет внешней, если она определена вне тела какой-либо функции. Поэтому стек и указатель стека, с которым должны работать push, pop и clear, определяются

вне этих трех функций. Однако сама main на стек и указатель стека не ссылается — реализация для нее тщательно скрыта. Поэтому программа операции = для обращения к вершине стека без ее уничтожения должна использовать обращение

```
push(pop());
```

Заметим еще, что так как операции + и * коммутативны, то порядок, в котором поднимаются операнды, не имеет значения, однако для — и/ нужно следить, где левый операнд, а где правый.

Упражнение 5.3. Имея в руках основную схему калькулятора, его можно легко усовершенствовать. Добавьте операцию взятия модуля (%) и унарный минус. Добавьте команду «erase», уничтожающую верхний элемент стека. Добавьте команды работы с переменными (просто сделать двадцать шесть переменных с однобуквенными именами).

5.5. ПРАВИЛА ОБЛАСТЕЙ ДЕЙСТВИЯ

Функции и внешние переменные, составляющие программу на языке Си, не обязательно транслировать одновременно. Исходный текст программы можно хранить в нескольких файлах, а предварительно оттранслированные подпрограммы можно загружать из библиотеки. Интересно обратить внимание на два вопроса:

Как писать описания, чтобы во время трансляции переменные были описаны надлежащим образом?

Как расставлять описания, чтобы все части во время загрузки были связаны между собой нужным образом?

Область действия имени — часть программы, в которой это имя определено. Для автоматической переменной, описанной в начале функции, областью действия будет та функция, где это имя описано; переменные с этим же именем в других функциях с данной никак не связаны. Это же справедливо и для аргументов функции.

Область действия внешней переменной простирается от точки во входном файле, где она была описана, до конца этого файла. Если, например, переменные val, sp и push, pop и clear определены в одном файле в таком порядке:

```
int sp = 0;
double val[MAXVAL];

double push(f) { ... }

double pop() { ... }

clear() { ... }
```

то переменные val и sp можно использовать в функциях push, pop и clear, обращаясь к ним просто по именам, никаких описаний не нужно.

С другой стороны, если на внешнюю переменную нужно ссылаться еще до ее определения или же она определена в *другом* входном файле, отличном от того, где она сейчас используется, то тогда вступает в силу описание extern.

Важно понимать различие между описанием внешней переменной и ее определением. *Описание* декларирует свойства переменной: ее тип, размер и т. д. , а *определение,* кроме этого, вызывает выделение памяти. Если такие, скажем, строки:

```
int sp;
double val[MAXVAL];
```

появляются вне какой-либо функции, то они *определяют* внешние переменные sp и val, приводят к выделению памяти и, кроме того, служат описаниями для оставшейся части файла. С другой стороны, строки

```
extern int sp;
extern double val[];
```

описывают для оставшейся части файла, что sp типа int, а val — массив значения с удвоенной точностью (его размер определяется где-либо еще), однако сами переменные не порождаются и память для них не выделяется.

Любая из внешних переменных во всех файлах, составляющих входную программу, должна *определяться* лишь *один* раз, а другие файлы могут содержать, для обеспечения обращения к ней, описания ее как extern. (Описание extern может встречаться и в файле, содержащем определение.) Инициация внешней переменной возможна только в ее определении. Размер массивов должен задаваться в определении, но допускается и в описании extern.

Хотя это и не особенно нужно для нашей программы, но переменные val и sp можно было бы определить и инициировать в одном файле, а функции push, pop и clear определить в другом. Затем эти определения и описания нужно было бы связать вместе так:

В файле 1:

```
int sp = 0;     /* указатель стека    */
double val[MAXVAL]; /* значение стека  */
```

В файле 2:

```
extern int sp;
extern double val[];

double push(f) { ... }

double pop() { ... }

clear() { ... }
```

Так как описание внешних в *файле 2* находится перед и вне всех трех функций, оно для них действительно, и на весь *файл 2* достаточно этих описаний.

Если мы имеем дело с большими программами, можно иметь одну-единственную копию внешних описаний для всей программы и вставлять эту копию в каждый файл при его трансляции с помощью конструкции #include, о которой еще пойдет речь в этой главе.

Вернемся теперь к реализации функции getop — функции, вызывающей следующий операнд или операцию. Основная задача проста: пропускать пробелы и символы табуляции или перехода на новую строку. Если следующий символ не цифра или десятичная точка, то он не выдается. В противном же случае набирается строка цифр, могущая включать десятичную точку, и выдается NUMBER — сигнал о том, что введено число.

Программа значительно усложняется из-за того, что нужно должным образом обрабатывать ситуации, когда приходит слишком длинное число. Функция getop читает цифры (возможна и десятичная точка) до тех пор, пока не обнаружится, что больше накапливать цифры уже нельзя. Если переполнения не было, то выдаются NUMBER и строка цифр. Если же число слишком длинное, то getop уничтожает остаток входной строки, так что пользователь может просто повторить ввод ошибочной строки; в качестве сигнала переполнения выдается TOOBIG:

```
getop(s, lim)   /* выборка следующего операнда или операции */
char s[];
int lim;
{
     int i, c;

     while ((c = getch()) == ' ' || c == '\t' || c == '\n')
          ;
     if (c != '.' && (c < '0' || c > '9'))
          return(c);
     s[0] = c;
     for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
          if (i < lim)
               s[i] = c;
     if (c == '.') {      /* дробная часть */
          if (i < lim)
               s[i] = c;
          for (i++; (c=getchar()) >= '0' && c <= '9'; i++)
               if (i < lim)
                    s[i] = c;
     }
     if (i < lim) { /* число правильное   */
          ungetch(c);
          s[i] = '\0';
          return(NUMBER);
     } else {   /* число велико, пропуск до конца строки */
          while (c != '\n' && c != EOF)
               c = getchar();
          s[lim-1] = '\0';
          return(TOOBIG);
     }
}
```

А что такое getch и ungetch? В программах, читающих входную информацию, часто встречается ситуация, когда невозможно определить, считано ли то, что нужно, если не считать несколько больше. Примером служит, скажем, процесс накопления символов, образующих число: до тех пор пока не появится «не цифра», число не окончено. Но в этот момент программа прочитала на один символ больше, и этот символ она не готова обрабатывать.

Проблему можно было бы решить, если бы была возможность «не читать» ненужный символ. Тогда каждый раз, когда программа прочитает на один символ больше, она могла бы вернуть его на вход. Последующие части программы могли бы в этом случае вести себя так, как будто ничего и не читалось. Такой возврат символа, к счастью, легко смоделировать, написав пару дополняющих друг друга функций. Функция getch поставляет следующий символ, подлежащий рассмотрению, а ungetch возвращает символ обратно на вход так, что следующее обращение к getch опять выдаст его же.

Совместная их работа проста. Ungetch сохраняет возвращаемые символы в некотором общем буфере — символьном массиве. Getch читает информацию из буфера, если она там есть, или же обращается к getchar, если буфер пустой. Кроме этого, должна быть и индексирующая переменная, где хранится положение в буфере текущего символа.

Так как буфер и индекс разделяются getch и ungetch и должны сохранять свои значения между обращениями, то они должны быть по отношению к функциям внешними. Таким образом, мы можем описать getch и ungetch и их разделяемые переменные так:

```
#define   BUFSIZE   100

char buf[BUFSIZE];   /* буфер для возврата */
int  bufp = 0; /* следующая свободная позиция в buf */

getch()    /* выдача (возможно возвращенного) символа */
{
     return((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch(c)      /* возврат выданного символа */
int c;
{
     if (bufp > BUFSIZE)
          printf("ungetch: too many characters\n");
     else
          buf[bufp++] = c;
}
```

Для «возврата» мы использовали массив, а не один символ, поскольку такая общность позднее может оказаться удобной.

Упражнение 5.4. Напишите подпрограмму ungets(s), возвращающую на вход целую строку. Нужно ли unqets знать о buf и bufp или можно воспользоваться только ungetch?

Упражнение 5.5. Предположим, что никогда не возвращается более одного символа. Модифицируйте должным образом getch и ungetch.

Упражнение 5.6. Наши программы не могут возвращать EOF и быть переносимыми. Посмотрите, что в них нужно изменить, чтобы можно было возвращать EOF, и напишите новые программы.

5.6. СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ

Статические переменные относятся к третьему виду памяти, дополняющей внешние и автоматические, о которых мы уже говорили.

Статическая переменная может быть внутренней или внешней. Внутренние статические переменные локальны по отношению к отдельной функции, так же как и автоматические, но в отличие от последних они продолжают существовать, а не возникают и уничтожаются при каждой активации функции. Это значит, что внутренние статические переменные являются собственной, постоянной памятью для функции. Символьные строки, появляющиеся в функциях, например, как аргументы для printf, представляют собой внутренние статические объекты.

Внешние статические переменные известны внутри оставшейся части *исходного файла* после того, как они в нем описаны, однако в других файлах они не известны. Таким образом, внешние статические объекты позволяют «прятать» имена вроде buf и bufp в комбинации функций getch и ungetch. Такие имена должны быть внешними, поскольку они общие для этих функций, однако для пользователей функций они должны оставаться невидимыми, и поэтому конфликтов по именам не возникает. Если эти две переменные и две функции собраны в одном файле:

```
static char    buf[BUFSIZE];  /*     буфер для ungetch */
static int     bufp = 0; /*          очередная позиция в buf */

getch() { ... }

ungetch(c) { ... }
```

тогда никакая другая подпрограмма не сможет «добраться» до buf и bufp; фактически эти имена не будут конфликтовать с аналогичными именами в других файлах той же программы.

Статичность памяти (внутренней или внешней) определяется добавлением перед обычным описанием слова static. Переменная внешняя, если она определена вне любой из функций, и внутренняя, если она определена внутри функции.

Функции обычно являются внешними объектами; их имена известны везде (они глобальные). Однако можно описать функцию как статическую: это приведет к тому, что ее имя будет неизвестно вне файла, где она описана.

В языке Си понятие «статическая» связано не только с постоянством, но и со степенью доступности, можно сказать, «собственности». Внутренние статические объекты известны только внутри одной функции; внешние статические объекты (переменные или функции) известны только внутри исходного файла, в котором они появляются. Их имена никак не путаются с именами переменных или функций из других файлов.

Внешние статические переменные и функции обеспечивают способ объединения данных и манипулирующих ими подпрограмм так, что другие подпрограммы и данные даже в случае небрежности не могут «конфликтовать» с ними. Например, getch и ungetch образуют «модуль» для ввода и возврата символов: buf и bufp должны быть статическими и поэтому недоступными извне. Аналогично push, pop и clear образуют модуль для работы со стеком: статическими внешними должны быть val и sp.

5.7. РЕГИСТРОВЫЕ ПЕРЕМЕННЫЕ

Регистры относятся к четвертому и последнему классу памяти. Описание register дает знать транслятору, что переменная, о которой идет речь, будет интенсивно использоваться. Если возможно, регистровые переменные помещаются в регистры машины; в результате это может привести к более быстрой и короткой программе.

Описание регистров выглядит так:

```
register int   x;
register char  c;
```

слово int можно опускать. Описание register применимо лишь к автоматическим переменным и к формальным параметрам функции. В последнем случае описание выглядит так:

```
f(c, n)
register int c, n;
{
        register int i;
        ...
}
```

На практике на регистровые переменные накладываются некоторые ограничения, отражающие реальные возможности конкретной машины. В каждой функции на регистрах можно хранить небольшое число переменных только некоторых типов. В случае избыточных или недопустимых описаний слово register просто игнорируется. Кроме того, для регистровых переменных невозможно взять адрес (об этом речь пойдет в гл. 5). Специфические ограничения меняются от машины к машине, например на PDP-11 в функции имеют силу только три описания регистров, а типы должны быть int, char или ссылка.

5.8. БЛОЧНАЯ СТРУКТУРА

Язык Си не относится к языкам с блочной структурой, в смысле ПЛ/1 или Алгола, здесь функции не могут определяться внутри других функций.

С другой стороны, переменные можно определять, как и в блочных языках. Описания переменных (включая инициацию) могут идти следом за левой фигурной скобкой, вводящей любой составной оператор, а не только в начале функции. Описанные таким образом переменные «затеняют» любые переменные с идентичными именами во внешних блоках и существуют до тех пор, пока не придет правая фигурная скобка. Например, в

```
if (n > 0) {
        int i;      /* описание нового i  */
        for (i = 0; i < n; i++)
                ...
}
```

областью действия переменной i является «истинная» ветвь оператора if; это i не имеет никакого отношения к другим i в программе.

Блочная структура применима и ко внешним переменным. Если у нас есть такие описания:

```
int x;

f()
{
        double x;
        ...
}
```

то внутри f вхождение x относится ко внутренней переменной удвоенной точности; вне f вхождение x относится ко внешней целой переменной. Это же справедливо и для имен формальных параметров:

```
int z;

f(z)
double z;
{
        ...
}
```

Внутри функции f z относится к формальному параметру, а не ко внешней переменной.

5.9. ИНИЦИАЦИЯ

Об инициации мы уже упоминали много раз, но всегда в связи с другими темами. Теперь мы суммируем некоторые из тех правил, о которых уже говорилось при обсуждении различных классов памяти.

Если явная инициация отсутствует, гарантируется, что внешние и статические переменные будут иметь значение нуль; автоматические и регистровые переменные будут иметь неопределенное значение (мусор).

Простые переменные (не массивы или записи) можно инициировать при их описании, если следом за именем поставить знак равенства и константное выражение:

```
int  x = 1;
char squote = '\'';
long day = 60 * 24; /* минут в дне  */
```

Внешние и статические переменные инициируются лишь один раз, концептуально, во время трансляции. Для автоматических и регистровых переменных инициация происходит при каждом входе в функцию или блок.

В случае автоматических и регистровых переменных инициация не ограничивается константами; фактически можно использовать любые допустимые выражения, включающие определенные ранее значения, даже обращения к функциям. Например, инициацию в программе поиска делением пополам, приведенной в гл. 4, можно было бы записать так:

```
binary(x, v, n)
int x, v[], n;
{
     int low = 0;
     int high = n - 1;
     int mid;
     ...
}
```

вместо

```
binary(x, v, n)
int x, v[], n;
{
     int low, high, mid;

     low = 0;
     high = n - 1;
     ...

}
```

Фактически инициация автоматических переменных — просто сокращенная запись операторов присваивания. Какой формой пользоваться, это дело вкуса. Как правило, мы будем использовать явные присваивания, поскольку инициация в описаниях «смотрится» хуже.

Автоматические массивы инициировать нельзя. Внешние же и статические массивы можно инициировать, поместив в описании список инициирующих значений, заключенных в фигурные скобки и разделенных запятыми. Например, программу подсчета символов из гл. 2, которая начиналась так:

```
main()     /* подсчет цифр, пустых символов и пр. */
{
     int c, i, nwhite, nother;
     int ndigit[10];

     nwhite = nother = 0;
     for (i = 0; i < 10; i++)
          ndigit[i] = 0;

}
```

можно было бы переписать так:

```
int  nwhite = 0;
int  nother = 0;
int  ndigit[10] ={ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

```
main()      /* подсчет цифр, пустых символов и пр. */
{
        int c, i;
        ...
}
```

В этой инициации фактически нет необходимости, так как все переменные и элементы получают нулевые значения. Однако лучше это делать в явном виде. Если инициирующих значений меньше, чем указанный размер, остальные элементы получают нулевые значения. Если же значений слишком много — это ошибка. К сожалению, не существует способа задать повторение инициирующих значений или инициировать элементы в середине массива, не задавая значений предыдущих элементов.

Для символьных массивов существует специальный способ инициации: вместо нотации с помощью скобок и запятых можно использовать строку

```
char pattern[] = "the";
```

Это сокращение более длинного, но эквивалентного описания

```
char pattern[] = { 't', 'h', 'e', '\0' };
```

Если размер массива любого типа пропущен, то транслятор определяет его длину, считая инициирующие значения. В данном конкретном случае размер равен 4 (три символа плюс закрывающий\0).

5.10. РЕКУРСИЯ

В языке Си функцию можно использовать рекурсивно, т. е. функция может обращаться *сама к себе* непосредственно или косвенно. Традиционный пример — печать чисел как последовательности символов. Как упоминалось ранее, цифры представления формируются в неудобном порядке: цифры младших разрядов числа получаются прежде старших, а печататься они должны в обратном порядке.

Задачу можно решать двумя способами. Первый — сохранять цифры по мере их формирования в массиве и печатать их оттуда в нужном порядке, как это делалось в программе itoa из гл. 4. Наш первый вариант программы printd построен по этому принципу:

```
printd(n) /* десятичная печать n */
int n;
{
        char s[10];
        int i;

        if (n < 0) {
                putchar('-');
                n = -n;
        }
        i = 0;
        do {
                s[i++] = n % 10 + '0';    /* следующий символ */
        } while ((n /= 10) > 0); /* убрать его */
        while (--i >= 0)
                putchar(s[i]);
}
```

Возможен и рекурсивный способ решения: при каждом обращении к printd она сначала обращается к себе, чтобы напечатать начальные цифры, а затем печатает замыкающую цифру:

```
printd(n) /* десятичная печать n (рекурсивная) */
int n;
{
     int i;

     if (n < 0) {
          putchar('-');
          n = -n;
     }
     if ((i = n/10) != 0)
          printd(i);
     putchar(n % 10 + '0');
}
```

При рекурсивном обращении функции к самой себе при каждом вызове создается новое множество всех автоматических переменных, и оно совершенно не зависит от предыдущего. Таким образом, при обращении printd(123) в первой активации функции n = 123. Во вторую printd передается 12, а после возврата из нее печатается 3. Аналогично вторая printd передает третьей 1 (та печатает ее), а затем сама печатает 2.

Рекурсии, как правило, не предусматривают никаких механизмов защиты памяти, так что иногда стек обрабатываемых величин может переполняться. И быстрее рекурсивные функции не работают. Однако рекурсивные программы более компактные, и их часто гораздо легче писать и понимать. Особенно удобны рекурсии при обработке данных с рекурсивно определенной структурой, например деревьев. Прекрасный пример такой работы приводится в гл. 7.

Упражнение 5.7. Используйте программу printd для написания рекурсивной версии itoa; т. е. рекурсивной программы для преобразования целого числа в строку.

Упражнение 5.8. Напишите рекурсивную версию функции reverse(s) «перевертывающей» строку s.

5.11. ПРЕПРОЦЕССОР ДЛЯ СИ

Предусмотрены и некоторые расширения языка, которые выполняются с помощью простого макропроцессора. Наиболее часто употребляемое из этих расширений — возможность определить (#define) нечто; можно еще и включать в программу во время трансляции содержимое других файлов.

Включение файлов

Возможность включения файлов предусмотрена как средство для облегчения обработки групп #define и описаний. Любая строка текста, выглядящая как

```
#include "имя файла"
```

заменяется содержимым файла с указанным именем. Кавычки здесь обязательны. Часто в начале каждого исходного файла есть одна-две строки такого вида; с их помощью вводятся общие операторы #define и описания глобальных переменных. Конструкции #include могут «вкладываться» одна в одну.

Конструкция #include представляет собою наиболее приемлемый способ собрать вместе описания для больших программ. Этот способ гарантирует, что все исходные файлы будут снабжены одинаковыми определениями и описаниями переменных и тем самым будет исключен один из источников ошибок. Конечно, если включаемый файл будет изменен, то все зависящие от него файлы должны быть перетранслированы.

Макроподстановка

Определение вида

```
#define   YES  1
```

вызывает макроподстановку наипростейшего вида — замену имени на строку символов. Имена в конструкции #define имеют тот же вид, что и идентификаторы языка Си. Замещающий текст обычно представляет собою остаток строки; длинное определение можно продолжить, если в конце продолжаемой строки поставить \. «Область действия» определяемых в #define имен простирается от точки определения до конца исходного файла. Имена могут переопределяться, и определения могут использовать предыдущие определения. Внутри строк, заключенных в кавычки, подстановка не происходит, так что, например, если YES—определенное имя, то в printf("YES") подстановки не будет.

Так как #define реализуется с помощью отдельного просмотра, а не как часть собственно трансляции, здесь нет сколь-либо существенных грамматических ограничений. Любители Алгола могут, например, написать:

```
#define then
#define begin  {
#define end    ;}
```

и после этого писать такие, например, программы:

```
if (i > 0) then
     begin
          a = 1;
          b = 2
     end
```

Можно также определить и макроподстановку с аргументами, так что заменяющий текст будет зависеть от вида макровызова. В качестве примера определим такую макроподстановку:

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

После этого строка

```
x = max(p+q, r+s);
```

будет заменена на строку

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Это позволяет ввести функцию max, которая вставляется прямо в программу, без обращения к подпрограмме. Поскольку аргумент трактуется буквальн-

но, такое макроопределение работает для любых типов данных и нет необходимости в различных видах max для данных различных типов, как это пришлось бы делать для функций.

Конечно, если вы посмотрите на полученное выше расширение для max, то обратите внимание на некоторые промахи. Дважды вычисляются выражения: это плохо, если они содержат побочные эффекты вроде обращения к функциям или операции увеличения. Надо тщательно расставлять скобки, чтобы быть уверенными в нужном порядке вычислений. Проанализируйте макроподстановку по такому определению:

```
#define  square(x)  x * x
```

если она вызывается следующим образом: square(z + 1).) Кроме того, существуют еще и чисто лексические сложности: между именем и левой скобкой, вводящей список аргументов, не может быть никаких пробелов.

Тем не менее макроподстановки крайне полезны. Практическим их примером является стандартная библиотека ввода-вывода, описываемая в гл. 8. В этой библиотеке getchar и putchar определяются как макроподстановки, что позволяет избежать интенсивных обращений к функциям при работе с символами. (Очевидно, для putchar требуется аргумент.)

Другие возможности макропроцессора описываются в приложении 1.

Упражнение 5.9. Определите макроподстановку swap(x, y), меняющую местами два целых аргумента. (В этом вам поможет блочная структура.)

Глава 6. ССЫЛКИ И МАССИВЫ

Ссылка представляет собою переменную, содержащую адрес другой переменной. Ссылки в языке Си используются очень интенсивно отчасти из-за того, что иногда некоторые вычисления можно выразить только с их помощью, а отчасти потому, что с ними получаются более компактные и эффективные программы, чем если бы мы пользовались другими средствами.

Ссылки и операторы перехода многие считают великолепным средством для порождения программ, которые невозможно понять. Это действительно верно, если ими пользоваться не аккуратно: ведь можно создавать ссылки, указывающие на самые неожиданные точки. Однако при некоторой дисциплине с помощью ссылок можно добиться и ясности, и простоты. Именно об этом мы и попытаемся рассказать.

6.1. ССЫЛКИ И АДРЕСА

Так как ссылка содержит адрес некоторого объекта, то возможно «косвенное» (через эту ссылку) обращение к такому объекту. Предположим, x — переменная типа int, а px — ссылка, созданная каким-то, хотя и неизвестным еще, способом. Унарная операция & дает *адрес* объекта, поэтому оператор

```
px = &x;
```

присваивает адрес x переменной px; говорят, что теперь px «ссылается на» x. Операцию & можно применять только к переменным и элементам массива; конструкции вида &(x+1) или &3 недопустимы.

Унарная операция * трактует свой операнд как адрес некоторого объекта и использует этот адрес для выборки содержимого. Таким образом, если y типа int, то

```
y = *px;
```

присваивает y содержимое того, на что указывает px. Поэтому

```
px = &x;
y = *px;
```

присваивает y то же самое, что и

```
y = x;
```

Переменные, участвующие в этих действиях, также необходимо описывать:

```
int  x, y;
int  *px;
```

Описание x и y такое же, как уже встречалось. Описание же ссылки px новое:

```
int  *px;
```

Оно мнемонично и говорит, что комбинация *px суть типа int, т. е. если px встречается в контексте *px, это эквивалентно переменной типа int. Фактически синтаксис описания переменной напоминает синтаксис выражений, в которых данная переменная может появляться. Такая форма полезна в тех случаях, когда описание сложное. Например:

```
double atof(), *dp;
```

говорит, что выражения atof( ) и *dp дают значения типа double.

Обратите также внимание на констатацию в описании того факта, что ссылка указывает на объект определенного вида.

Ссылки могут встречаться в выражениях. Например, если px — ссылка на целое x, то *px может встречаться в любом месте, где может встречаться x. Оператор

```
y = *px + 1
```

присваивает y значение x, увеличенное на 1;

```
printf("%d\n", *px)
```

печатает текущее значение x, а

```
d = sqrt((double) *px)
```

сохраняет в d корень квадратный из x, который, прежде чем попадет в sqrt как аргумент, преобразуется к типу double (см. гл. 3). В выражениях, подобных

```
y = *px + 1
```

унарные операции * и & более приоритетны, чем арифметические операции, поэтому в таком выражении берется то, на что указывает px, к нему добавляется 1 и результат присваивается y. Вскоре мы объясним, что означает y:

```
y = *(px + 1)
```

Ссылки могут встречаться и в левой части присваивания. Если px указывает на x, то

```
*px = 0
```

присваивает нуль переменной x, а

```
*px += 1
```

увеличивает x, как и

```
(*px)++
```

В последнем примере скобки необходимы, без них такое выражение увеличивало бы px, а не то, на что она (ссылка) указывает. Это произошло бы из-за того, что унарные операции вроде * и ++ вычисляются справа налево.

И наконец, так как ссылки суть переменные, то ими можно манипулировать, как и обычными переменными. Если py другая ссылка на int, то оператор

```
py = px
```

копирует содержимое px в py, т. е. py начинает указывать на то же, на что указывает px.

6.2. ССЫЛКИ И АРГУМЕНТЫ ФУНКЦИИ

Так как в языке аргументы функции передаются значением, то нет прямого способа в вызванной функции изменить некоторую переменную в вызвавшей функции. Что же делать, если фактически необходимо изменить первоначальный аргумент? Например, подпрограмме сортировки нужно поменять местами два элемента, обращаясь к функции swap. Недостаточно написать

```
swap(a, b);
```

где функция swap определена так:

```
swap(x, y)      /* НЕВЕРНО */
int x, y;
{
     int temp;

     temp = x;
     x = y;
     y = temp;
}
```

Так как передаются значения, то swap *не может* воздействовать на аргументы a и b в обратившейся к ней программе.

К счастью, есть способ добиться желаемого эффекта. Вызывающая программа в этом случае передает ссылки на величины, которые нужно поменять местами:

```
swap(&a, &b);
```

Так как операция & дает адрес переменной, то &a — ссылка на a. В самой функции swap аргументы описываются как ссылки, и доступ к фактическим операндам идет через них.

```
swap(px, py)    /* поменять *px и *py */
int *px, *py;
{
     int temp;

     temp = *px;
     *px = *py;
     *py = temp;
}
```

Часто используется аргумент-ссылка и в функциях, возвращающих более одного значения. (Можно сказать, что swap также возвращает два значения: новые значения своих аргументов.) Рассмотрим, например, функцию getint, выделяющую из входного потока символов целое число, записанное в свободном формате, и преобразующую это представление в значение типа int. При одном обращении к getint выдается одно целое значение; если вводить больше нечего, то дается сигнал о конце файла. Эти значения нужно выдавать как отдельные объекты. Какое бы значение ни было выбрано для EOF, всегда может встретиться такое же целое.

Наше решение основывается на функции ввода scanf (см. гл. 8) и заключается в том, что getint как значение функции возвращает либо

EOF, если обнаружен конец файла, либо некоторое значение, сигнализирующее о приходе обычного целого числа. Найденное целое значение выдается через аргумент, который должен быть ссылкой на целое число. Такая организация отделяет состояние конца файла от числовых значений.

Приведенный ниже цикл заполняет целыми числами массив, обращаясь к getint:

```
int n, v, array[SIZE];

for (n = 0; n < SIZE && getint(&v) != EOF; n++)
     array[n] = v;
```

Каждое обращение посылает в v очередное целое число, обнаруженное на входе. Обратите внимание, сколь важно писать &v вместо v на месте аргумента getint. Если использовать просто v, то это скорее всего приведет к ошибочной адресации, так как getint рассчитывает, что она работает с правильной ссылкой.

Сама функция getint есть некоторая модификация ранее описанной функции atoi:

```
getint(pn)      /* взять очередное целое */
int *pn;
{
     int c, sign;

     while ((c = getch()) == ' ' || c == '\n' || c == '\t')
          ;     /* пропуск пустых символов */
     sign = 1;
     if (c == '+' || c == '-') {    /* запись знака */
          sign = (c=='+') ? 1 : -1;
          c = getch();
     }
     for (*pn = 0; c >= '0' && c <= '9'; c = getch())
          *pn = 10 * *pn + c - '0';
     *pn *= sign;
     if (c != EOF)
          ungetch(c);
     return(c);
     }
```

Внутри getint *pn используется как обычная целая переменная. Здесь же мы используем и getch, и ungetch (см. гл. 5), так что один лишний символ, который мы вынуждены считывать, можно вернуть опять на вход.

Упражнение 6.1. Напишите getfloat, аналог getint для плавающей точки. Какой тип будет возвращать getfloat в качестве значения функции?

6.3. ССЫЛКИ И МАССИВЫ

В нашем языке существует сильная взаимосвязь между ссылками и массивами, настолько сильная, что ссылки и массивы фактически надо рассматривать одновременно. Любое действие, которое достигается индексированием массива, может быть выполнено и с помощью ссылок. Вариант со ссылками в общем-то будет быстрее, но он, по крайней мере для начинающих, несколько тяжеловат для понимания.

Описание

```
int a[10]
```

определяет массив a размером в 10 элементов, т. е. это блок из 10 последовательных объектов, именуемых a[0], a[1], ..., a[9]. Запись *a* [i] обозначает элемент в i-й позиции от начала. Если pa—ссылка на целое значение, описанная как

```
int *pa
```

то присваивание:

```
pa = &a[0]
```

устанавливает в pa ссылку на нулевой элемент a; т. е. pa содержит адрес a[0]. Теперь присваивание

```
x = *pa
```

копирует содержимое a[0] в x.

Если содержимое pa указывает на отдельный элемент массива a, то *по определению* pa + 1 указывает на следующий элемент, и, вообще, pa — i указывает на i-й элемент перед pa, а pa + i на i-й элемент после. Таким образом, если pa указывает на a[0], то

```
*(pa+1)
```

относится к содержимому a[1], pa + i есть адрес a[i], a *(pa+i) есть содержимое a[i].

Эти замечания справедливы вне зависимости от типа переменных в массиве a. Определение операции «добавление 1 к ссылке» и другой ссылочной арифметики подразумевает масштабирование, связанное с размером памяти для объекта, на который указывает ссылка. Таким образом, в pa + i значение i, прежде чем будет добавлено к pa, будет умножено на размер объекта, на который указывает pa.

Очевидно, что между индексированием и ссылочной арифметикой связь очень тесная. Фактически любое упоминание массива приводится транслятором к ссылке на начало этого массива, т. е. имя массива есть ссылочное выражение. Это приводит к небольшому числу полезных следствий. Так как имя массива есть синоним для местоположения нулевого элемента, то присваивание

```
pa = &a[0]
```

можно записать и в таком виде:

```
pa = a
```

Не удивительно теперь, по крайней мере на первый взгляд, что значение a[i] можно записать как *(a+i). Вычисляя a[i], транслятор сразу же переводит его в *(a+i); эти две формы полностью эквивалентны. Применяя операцию & к обеим частям этого равенства, получаем, что &a[i] и a + i также идентичны: a + i — адрес i-го элемента относительно a. С другой стороны, если pa — ссылка, то ее можно использовать с индексом: pa[i] идентично *(pa+i). Короче, любой массив и индексное выражение можно записать как ссылку и смещение и, наоборот, причем это можно делать даже в одном операторе.

Однако между именем массива и ссылкой есть одно различие, о котором следует всегда помнить. Ссылка есть переменная, так что pa=a и pa++ суть осмысленные операции. Имя же массива — константа, а не переменная, поэтому конструкции вроде a=pa или a++, или p=&a недопустимы.

Если имя массива передается функции, то оно передается как местоположение начала массива. Внутри вызванной функции этот аргумент — переменная, такая же, как любые другие переменные. Поэтому аргумент, связанный с именем массива, фактически ссылка, т. е. переменная, содержащая адрес. Мы можем воспользоваться этим фактом и написать новую версию функции strlen, вычисляющей длину строки.

```
strlen(s) /* подсчет длины строки s */
char *s;
{
        int n;

        for (n = 0; *s != '\0'; s++)
                n++;
        return(n);
}
```

Увеличение s, бесспорно, допустимо, так как это ссылочная переменная. Операция s++ никак не воздействует на строку символов, с которой обратились к strlen, а просто увеличивает собственную для strlen копию адреса.

В определении функции формальные параметры

char s[ ] и char *s

совершенно эквивалентны. Какое описание использовать, во многом определяется тем, как будут записываться в функции соответствующие выражения. Если имя массива передается функции, то, составляя функцию, можно исходить из того, что она работает с массивом, а можно исходить и из того, что она работает со ссылкой. Можно даже, если это удобно, использовать вместе и то и другое толкование.

Функции можно передавать и часть массива, если задать ссылку на начало подмассива. Например, если a — массив, то

f(&a[2])

и

f(a+2)

передают функции f адрес элемента a[2], так как и &a[2], и a+2 суть ссылочные выражения и оба указывают на третий элемент в массиве a. Внутри f описание аргумента можно записать и так:

```
f(arr)
int *arr;
{

}
```

и так:

```
f(arr)
int arr[];
{

}
```

Тот факт, что аргумент в действительности указывает на часть большого массива, на f не сказывается.

6.4. АДРЕСНАЯ АРИФМЕТИКА

Если p — некоторая ссылка, то p++ увеличивает p, теперь она указывает на следующий элемент вне зависимости от типа объекта, о которых идет речь. Оператор p+=i передвигает (увеличивает) ссылку на i элементов относительно текущего. Эти и подобные конструкции есть простейшие и наиболее общие виды адресной или ссылочной арифметики.

Язык Си последователен и регулярен в своем подходе к адресной арифметике; целостность и взаимосвязь ссылок, массивов и адресной арифметики представляют собой одно из основных достоинств языка. Проиллюстрируем некоторые из его свойств на примере «рудиментарного»

распределителя памяти. Он может оказаться полезным именно в силу своей простоты. Есть две подпрограммы: alloc(n) выдает ссылку p на n последовательных «символьных позиций»; обратившийся к alloc может использовать их для хранения символов. Функция free(p) освобождает захваченную таким образом память, и ее можно позже повторно использовать. Рудиментарность заключается в том, что освобождать память нужно в порядке, противоположном тому, в котором она выделялась. Это означает, что alloc и free управляют памятью по принципу стека или списка с дисциплиной «последний пришел — первый вышел» (LIFO). В стандартной библиотеке языка есть аналогичные функции, не имеющие таких ограничений. В гл. 8 мы познакомимся и с этими улучшенными версиями. Однако во многих приложениях реально нужна некоторая тривиальная функция alloc для выделения памяти на непредсказуемое время, причем памяти некоторого непредсказуемого размера.

Самая простая реализация заключается в том, чтобы alloc брала этот кусок из большого массива символов. Мы будем называть его allocbuf. Это собственный массив функций alloc и free. Так как они работают со ссылками, а не с индексами массива, никаким другим программам нет необходимости знать имя этого массива, который можно описать как внешний статический. Это означает, что он локален в исходном файле, содержащем alloc и free, и невидим вне этого файла. В практической реализации такой массив может даже и не иметь имени, его можно получить от операционной системы как ссылку на некоторый безымянный блок памяти.

Кроме того, нужна еще информация, указывающая, какая часть allocbuf уже использована. Мы для этого возьмем ссылку (по имени allocp) на очередной свободный элемент. Если к alloc обращаются с просьбой выделить n символов, она проверяет, осталось ли в allocbuf достаточно места для этого массива. Если осталось, то функция выдает текущее значение allocp (т. е. начало свободного блока), а затем увеличивает его на n, после этого allocp указывает снова на очередной свободный блок. Функция free(p) просто присваивает значение p ссылке allocp, если p указывает внутрь allocbuf.

```
#define   NULL 0     /* значение, сигнализирующее об ошибке */
#define   ALLOCSIZE 1000 /* объем доступной памяти */

static char allocbuf[ALLOCSIZE];  /* память для alloc */
static char *allocp = allocbuf;   /* очередное свободное место  */

char *alloc(n) /* выдача ссылки на n символов */
int n;
{
     if (allocp + n <= allocbuf + ALLOCSIZE) { /* есть  */
          allocp += n;
          return(allocp - n); /* старое  */
     } else          /* места нет */
          return(NULL);
}

free(p)    /* освободить память по  p */
char *p;
{
     if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
          allocp = p;
}
```

Теперь несколько пояснений. В общем случае ссылку можно инициировать, так же как и любую другую переменную, хотя обычно осмысленным является лишь значение NULL (о нем еще пойдет речь) или же значения выражений, содержащих адреса предварительно определенных данных соответствующих типов. Описание

```
static char *allocp = allocbuf;
```

определяет allocp как ссылку на символ и инициирует ее значением, указывающим на allocbuf как на очередное свободное место в начале работы программы. Это же можно было записать и так:

```
static char *allocp = &allocbuf[0];
```

поскольку имя массива *есть* адрес его нулевого элемента, такое использование более естественно.

Проверка

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

контролирует, хватает ли места для удовлетворения требования на n символов. Если хватает, то новое значение allocp по крайней мере на единицу отличается от конца allocbuf. Если требование удовлетворить можно, то alloc выдает обычную ссылку (обратите внимание на описание самой функции). Если же нельзя, то нужно выдать некоторый сигнал о том, что места не осталось. Язык гарантирует, что ни одна ссылка, указывающая на какие-то данные, не будет иметь нулевое значение; поэтому такое значение можно использовать как сигнал о необычайном событии, в нашем случае — об отсутствии памяти. Вместо нуля мы будем писать NULL, дабы подчеркнуть, что это некоторое особое значение для ссылки. Обычно целое число нельзя осмысленно присваивать ссылке; нуль — особый случай.

Проверки вида

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

и

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

демонстрируют некоторые важные особенности ссылочной арифметики. Первое, при некоторых обстоятельствах ссылки можно сравнивать. Если p и q указывают на элементы одного и того же массива, то отношения $<, >, =$ и т. д. работают правильно. Например $p<q$ дает ответ «истина», если p указывает на более «ранний» элемент массива, чем тот, на который указывает q. Отношения $==$ и $!=$ также работают должным образом. Любую ссылку можно осмысленно сравнивать на равенство или неравенство со значением NULL. Однако все летит кувырком, если сравниваются или участвуют в операциях ссылки на различные массивы. Если вы это сделаете, то на всех машинах вы получите очевидную чепуху. Если же вы будете поступать более изощренным образом, то на одной машине ваша программа будет работать, но таинственно перестанет это делать на другой.

Второе, вы уже заметили, что ссылки и целые числа можно складывать или вычитать. Конструкция $p + n$ указывает на n-й объект относительно точки, на которую указывает p. Это справедливо, на какого бы типа объекты ни указывала ссылка p; транслятор будет масштабировать p в соответствии с типом, определяемым из описания для p. Например, на PDP-11 масштабный множитель для типа char равен 1, для int и short 2, для long и float 4 и 8 для двойной точности.

Возможно и вычитание ссылок; если p и q ссылаются на элементы одного массива, то p — q есть число элементов между p и q. Этот факт можно использовать для составления еще одной версии программы strlen:

```
strlen(s) /* подсчет длины строки s */
char *s;
{
    char *p = s;

    while (*p != '\0')
        p++;
    return(p-s);
}
```

Здесь в описании p инициируется так, что она указывает на s, т. е. на первый символ. В цикле по очереди проверяется каждый символ, пока в конце не будет обнаружен \0. Так как значение \0 есть нуль, а цикл с while проверяет, не нуль ли выражение, это позволяет опустить явную проверку, и поэтому часто такие циклы записываются как

```
while (*p)
    p++;
```

Поскольку p указывает на символ, то p++ перестраивает каждый раз p на следующий символ, а p — s суть число пройденных символов, т. е. длина строки. Ссылочная арифметика осмысленна; если бы мы имели дело с числами типа float, занимающими больше памяти, чем символы, и p указывала бы на float, то операция p++ перестраивала бы ссылку на следующее число с плавающей запятой. Поэтому мы можем написать другую версию alloc для работы, скажем с float, просто заменив везде в alloc и free слово char на float. Везде при работе со ссылками автоматически будут учтены размеры объектов и связанные с ними ссылки, и ничего больше изменять не требуется.

6.5. СИМВОЛЬНЫЕ ССЫЛКИ И ФУНКЦИИ

Любая *строковая константа* вроде такой:

```
"I am a string"
```

представляет собой массив символов. Во внутреннем представлении транслятор закончит такой массив символом \0, так что программа может обнаружить этот концевой символ. Поэтому строка занимает в памяти на один символ больше, чем указано между двойными кавычками.

Строковые константы чаще всего появляются в качестве аргументов функции. Например:

```
printf("hello, world\n");
```

Если такая строка символов встречается в программе, то обращение к ней идет через ссылку: printf получает ссылку на символьный массив.

Символьные массивы, конечно же, употребляются не только как аргументы функций. Если описать сообщение как

```
char *message;
```

то оператор

```
message = "now is the time";
```

присваивает переменной message ссылку на фактическую строку. Сама строка не копируется, передается только ссылка. В языке нет каких-либо операций для работы со строками символов целиком, как единым целым.

Для иллюстрации некоторых из аспектов работы со ссылками и массивами давайте рассмотрим две полезные подпрограммы из стандартной библиотеки ввода-вывода, о которой речь пойдет в гл. 8.

Первая функция strcpy(s, t) копирует строку t в строку s. Порядок аргументов аналогичен порядку в операторе присваивания, если бы его можно было написать так:

s = t

т. е. s присвоить значение t. Сначала приведем версию, основанную на работе с массивами:

```
strcpy(s, t)    /* копирование t в s; */
char s[], t[];
{
     int i;

     i = 0;
     while ((s[i] = t[i]) != '\0')
          i++;
}
```

Для сравнения ниже приводится версия с использованием ссылок.

```
strcpy(s, t)    /* копирование t в s; вариант 1 со ссылками */
char *s, *t;
{
     while ((*s = *t) != '\0') {
          s++;
          t++;
     }
}
```

Так как аргументы передаются как значения, то в strcpy s и t можно использовать наиболее удобным образом. Здесь они рассматриваются как инициированные должным способом ссылки, которые продвигаются вдоль массивов, за шаг на один символ, до тех пор, пока не встретится \0; в этот момент копирование t в s заканчивается.

С практической точки зрения приведенную программу не стоит использовать: ее можно написать и по-другому.

```
strcpy(s, t)    /* копирование t в s; вариант 2 со ссылками */
char *s, *t;
{
     while ((*s++ = *t++) != '\0')
          ;
}
```

Увеличение s и t переносится в проверку. Значение *t++ есть символ, на который указывает t до ее увеличения. Постфиксная операция ++ не изменяет t до тех пор, пока не будет получен требуемый символ. Аналогично символ сохраняется в старой s позиции (т. е. позиции, определяемой s до ее увеличения). Этот же символ будет и значением, сравниваемым с \0 при управлении циклом.

Продолжая сокращать программу, можно обратить внимание и на то, что сравнение с \0 излишне. Поэтому такую функцию записывают следующим образом:

```
strcpy(s, t)    /* копирование t в s; вариант 3 со ссылками */
char *s, *t;
{
     while (*s++ = *t++)
          ;
}
```

Хотя на первый взгляд она и может показаться некоторой криптограммой, но записывать ее так значительно удобнее, и, если нет других причин, такую идиоматическую запись следует освоить, вы с ней будете часто встречаться в программах на языке Си.

Вторая программа, strcmp(s, t), сравнивает символьные строки s и t и выдает отрицательное, нулевое или положительное число в зависимости от того, будет ли s лексикографически меньше, равно или больше t. Возвращаемое значение получается путем вычитания первых несовпадающих символов из s и t.

```
strcmp(s, t)  /* return <0 if s<t, 0 if s==t, >0 if s>t */
char s[], t[];
{
     int i;

     i = 0;
     while (s[i] == t[i])
          if (s[i++] == '\0')
               return(0);
     return(s[i] - t[i]);
}
```

Вариант со ссылками таков:

```
strcmp(s, t)  /* return <0 if s<t, 0 if s==t, >0 if s>t */
char *s, *t;
{
     for ( ; *s == *t; s++, t++)
          if (*s == '\0')
               return(0);
     return(*s - *t);
}
```

Так как операции ++ и -- префиксные или постфиксные, то хотя и редко, но встречаются и другие комбинации * и ++ или --. Например, в *++p увеличивается *до того*, как будет получен (считан) символ, на который указывает p. Аналогично в *--p сначала будет уменьшена ссылка p.

Упражнение 6.2. Напишите вариант программы strcat, из гл. 2, с использованием ссылок. Программа strcat(s, t) копирует строку t в конец строки s.

Упражнение 6.3. Напишите макроопределение для strcpy.

Упражнение 6.4. Перепишите в варианте со ссылками уместные в этом случае программы из предшествующих глав. Хорошо бы переписать getline (гл. 2 и 5), atoi, itoa и их варианты (гл. 3, 4 и 5), reverse (гл. 4) и index, getop (гл. 5).

6.6. ССЫЛКИ — НЕ ЦЕЛЫЕ ЗНАЧЕНИЯ

В «ископаемых» программах на языке Си можно было обратить внимание на довольно небрежное отношение к копированию ссылок. В общем-то будет справедливым сказать, что для большинства машин ссылки можно присваивать целым переменным, а затем вновь пересылать их в ссылки; при этом ничего не меняется: не выполняется никакого масштабирования или преобразования и не теряется ни одного разряда. К сожалению, это привело к тому, что в обиход вошло свободное обхождение с подпрограммами, выдающими ссылки, которые затем просто передаются в другие программы, при этом часто необходимые описания опускаются. Разберем, например, функцию strsave(s), копирующую строку s на свободное место, полученное при обращении к alloc, и выдающую ссылку на эту копию. Эту программу правильно было бы писать так:

```
char *strsave(s)     /* сохранение s */
char *s;
{
     char *p, *alloc();

     if ((p = alloc(strlen(s)+1)) != NULL)
          strcpy(p, s);
     return(p);
}
```

На практике же имеется тенденция пропускать описания:

```
strsave(s)      /* сохранение s */
{
     char *p;

     if ((p = alloc(strlen(s)+1)) != NULL)
          strcpy(p, s);
     return(p);
}
```

На многих машинах такая программа работать будет так, как подразумеваемый тип функции и аргументов — int, а целые и ссылки можно без осложнений присваивать друг другу. Однако употребление программ такого типа рискованно, ибо они зависят от реализации и от архитектуры машины и ваш конкретный транслятор может реализовать их по-другому. Мудрее будет сохранять все описания. (Программа lint предупреждает о таких конструкциях, которые могут возникнуть из-за невнимательности.)

6.7. МНОГОМЕРНЫЕ МАССИВЫ

В языке предусмотрена и работа с прямоугольными многомерными массивами, хотя на практике они используются значительно реже, чем массивы ссылок. В этом разделе мы покажем некоторые из их свойств.

Рассмотрим задачу о преобразовании даты в виде дня и месяца в день года и обратно. Например, 1 марта — 60-й день невисокосного года и 61-й

день високосного. Для такого преобразования определим две функции: day_of_year, преобразующую месяц и день в день года, и month_day, переводящую день года в месяц и день. Так как последняя функция возвращает два значения, то соответствующие аргументы — ссылки:

```
month_day(1977, 60, &m, &d)
```

присваивает значение 3 переменной m и 1—d(1-е марта).

Этим функциям нужна одна и та же информация: таблица чисел, задающая число дней в месяцах («в сентябре тридцать дней...»). Так как число дней в месяце зависит от того, високосный или нет год, то легче разделить эту таблицу на две строки двумерного массива, а не пытаться «следить за февралем». Массивы таблицы и функции преобразования описываются так:

```
static int day_tab[2][13] ={
     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
     {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

day_of_year(year, month, day) /* номер дня в году */
int year, month, day;         /* по месяцу и числу */
{
     int i, leap;

     leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
     for (i = 1; i < month; i++)
          day += day_tab[leap][i];
     return(day);
}

month_day(year, yearday, pmonth, pday) /* месяц и число по */
int year, yearday, *pmonth, *pday; /* номер у дня в году */
{
     int i, leap;

     leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
     for (i = 1; yearday > day_tab[leap][i]; i++)
          yearday -= day_tab[leap][i];
     *pmonth = i;
     *pday = yearday;
}
```

Массив day_tab внешний для day_of_year и month_day, поэтому они обе могут им пользоваться. Это первый двумерный массив, с которым мы встречаемся. По определению в языке двумерный массив есть одномерный массив, элементы которого суть массивы. Следовательно, индексы записываются так:

```
day_tab[i][j]
```

а не

```
day_tab[i, j]
```

как в большинстве языков. За исключением этой особенности, двумерные массивы можно трактовать точно так же, как и в других языках. Элементы их хранятся по строкам, т. е. если проходить по элементам в порядке их хранения, то быстрее всего изменяется самый правый индекс.

Инициируется массив с помощью списка начальных значений, заключенного в фигурные скобки; каждая строка двумерного массива инициируется соответствующим подсписком. Массив day_tab мы начинаем с нулевого столбца, чтобы месяц изменялся от 1 до 12, что естественно, а не от 0 до 11. Так как место в этой программе экономить ни к чему, то легче сделать так, а не уменьшать индекс на единицу.

Если двумерный массив нужно передавать функции, то описание аргумента в этой функции *должно* включать в себя размер строки массива, а размер столбцов неважен, так как массив, как и раньше, передается с помощью ссылки. В нашем конкретном случае ссылка указывает на объект, представляющий собою массив из 13 целых чисел. Таким образом, если массив day_tab передать в функцию f, то описание этой функции должно выглядеть так:

```
f(day_tab)
int day_tab[2][13];
{

}
```

Описание аргумента в f можно сделать и таким:

```
int day_tab[][13];
```

так как размер столбцов не имеет значения, а можно было бы описать его и так:

```
int (*day_tab)[13];
```

Такое описание говорит, что аргумент есть ссылка на массив из 13 целых. Скобки здесь обязательны, поскольку индексные скобки [ ] имеют больший приоритет, чем *. Без скобок описание

```
int *day_tab[13];
```

относится, как будет пояснено в следующем разделе, к 13 ссылкам на целые числа.

6.8 МАССИВЫ ССЫЛОК; ССЫЛКИ НА ССЫЛКИ

Так как ссылки представляют собою переменные, можно ожидать, что будут использоваться и массивы ссылок. И это действительно так. Проиллюстрировать этот прием можно, написав программу для сортировки в алфавитном порядке множества строк текста; это будет упрощенный до предела вариант обслуживающей программы sort из системы UNIX.

В гл. 4 мы уже описывали функцию сортировки shell для массива целых чисел. Здесь мы используем тот же алгоритм, но теперь мы имеем дело со строками текста, причем все они различны по длине и в отличие от целых их нельзя сравнивать и пересылать одной операцией. Поэтому нужно такое представление данных, которое позволяло бы удобно и эффективно работать с текстовыми строками переменного размера.

Здесь самое время воспользоваться массивом ссылок. Если хранимые строки располагаются плотно одна за другой в одном длинном символьном массиве (полученном, скажем, с помощью alloc), то до каждой строки можно добираться с помощью ссылки на ее первый символ. Сами ссылки можно хранить в некотором массиве. Сравнить две строки можно с помощью функции strcmp, передавая ей ссылки на них.

Если две строки нужно поменять местами из-за неверного их расположения, то переставляются *ссылки* в массиве ссылок, а не сами текстовые строки. Такое решение позволяет отказаться от проблемы близнецов при усложненном управлении памятью и осложнений, связанных с переносом фактических строк.

Процесс сортировки включает три этапа:

чтение всех строк на входе;
их сортировку;
печать их в установленном порядке.

Как обычно, лучше всего разделить программу на функции, соответствующие этому естественному делению, и основную программу, ими управляющую.

Отложим на некоторое время этап сортировки и остановимся на структуре данных и вводе и выводе. Программа ввода должна собирать и сохранять символы каждой строки, а также строить массив ссылок к этим строкам. Кроме того, она считает число строк, так как эта информация нужна для сортировки и печати. Поскольку программа ввода может работать только с некоторым конечным числом входных строк, то в этом случае, если их будет слишком много, она может выдавать некоторый «запрещенный» счетчик строк, скажем — 1. Программа вывода только печатает строки в том порядке, как они указаны в массиве ссылок.

```
#define NULL   0
#define LINES  100   /* максимальное число хранимых строк */

main()     /* сортировка входных строк */
{
     char *lineptr[LINES]; /* ссылки на строки текста  */
     int  nlines;          /* число прочитанных строк */

     if ((nlines = readlines(lineptr, LINES)) >= 0) {
          sort(lineptr, nlines);
          writelines(lineptr, nlines);
     }
     else
          printf("input too big to sort\n");
}
#define MAXLEN 1000
readlines(lineptr, maxlines)  /* чтение строк  */
char *lineptr[];              /* для сортировки */
int maxlines;
{
     int  len, nlines;
     char *p, *alloc(), line[MAXLEN];

     nlines = 0;
     while ((len = getline(line, MAXLEN)) > 0)
          if (nlines >= maxlines)
               return(-1);
          else if ((p = alloc(len)) == NULL)
               return(-1);
          else {
               line[len-1] = '\0'; /* новая строка */
               strcpy(p, line);
               lineptr[nlines++] = p;
          }
     return(nlines);
}
```

Переход на новую строку в конце каждой строки исключается, и поэтому он не влияет на порядок сортировки.

```
writelines(lineptr, nlines)    /* выдача строк */
char *lineptr[];
int nlines;
{
     int i;

     for (i = 0; i < nlines; i++)
          printf("%s\n", lineptr[i]);
}
```

Основное новшество это описание

```
char *lineptr[LINES];
```

оно говорит, что lineptr — это массив из элементов, представляющих собою ссылки на величины типа char. Всего таких элементов LINES. Таким образом, lineptr[i] —ссылка на символ, а *lineptr[i] относится к самому символу.

Так как lineptr — это массив, который передается в функцию writelines, его можно трактовать как ссылку, в точности аналогично тому, как мы делали это в предыдущих примерах. И функцию можно переписать таким образом:

```
writelines(lineptr, nlines)    /* выдача строк */
char *lineptr[];
int nlines;
{
     while (--nlines >= 0)
          printf("%s\n", *lineptr++);
}
```

В самом начале *lineptr указывает на первую строку, каждое последующее увеличение ссылки переводит нас на следующую строку, пока не будет исчерпана переменная nlines.

Разобравшись с вводом и выводом, мы можем перейти к сортировке. Сортировка shell из гл. 3 нуждается лишь в небольших изменениях: нужно модифицировать описания и выполнять сравнения с помощью отдельной функции. Основной алгоритм остается тем же самым, что позволяет надеяться на его правильную работу:

```
Sort(v, n)      /* сортировка строк   v[0] ... v[n-1] */
Char *v[];           /* в возрастающем порядке */
Int n;
{
     int  gap, i, j;
     char *temp;

     for (gap = n/2; gap > 0; gap /= 2)
          for (i = gap; i < n; i++)
               for (j = i-gap; j >= 0; j -= gap) {
                    if (strcmp(v[j], v[j+gap]) <= 0)
                         break;
                    temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
               }
}
```

Так как любой отдельный элемент v (двойник lineptr) есть ссылка на символ, а temp того же типа, их можно присваивать друг другу.

Мы написали эту программу довольно прямолинейно, но старались, чтобы она работала быстро. Ее можно сделать и более быстрой, копируя, например, приходящие строки непосредственно в массив, организованный внутри функции readlines, вместо того, чтобы копировать их в line, а затем в некоторое место, получаемое при обращении к alloc. Однако первый набросок программы разумнее делать более легким для понимания, а об эффективности уже заботиться позже.

Способ, с помощью которого можно сделать эту программу более быстрой, скорее всего не связан с лишними копированиями входных строк. Замена «пузырькового» метода сортировки на какой-нибудь лучший, вроде Quicksort, с большей вероятностью приведет к улучшению.

Во второй главе мы уже говорили, что, поскольку циклы с while и for проверяют условие окончания даже *перед первым* выполнением тела цикла, они позволяют быть уверенными, что программа будет работать и «на границах», в частности если не будет вообще входных строк. В этом можно убедиться, просматривая функции программы сортировки и следя за тем, что произойдет, если входных строк не будет.

Упражнение 6.5. Перепишите readlines так, чтобы строки создавались в массиве, выделенном для этого в main, без обращения к alloc за памятью. Насколько быстрее ваша версия?

6.9. ИНИЦИАЦИЯ МАССИВОВ ССЫЛОК

Предположим, нам надо написать функцию month_name(n), дающую ссылку на строку символов, содержащую название n-го месяца. Это идеальная ситуация для использования внутреннего статического массива. Функция month_name имеет свой собственный массив символьных строк и выдает ссылку на требуемую строку. Цель данного раздела—показать, как инициируется массив названий.

Синтаксис соответствующей конструкции похож на те инициации, с которыми мы уже встречались.

```
char *month_name(n) /* выдать название n-го месяца */
int n;
{
    static char *name[] ={
        "illegal month",
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    };

    return((n < 1 || n > 12) ? name[0] : name[n]);
}
```

Описание для переменной name (name есть массив символьных ссылок) то же самое, что и для lineptr в примере с сортировкой. Инициатор —

просто список символьных строк, каждая сопоставляется соответствующему элементу в массиве. Более же точно, символы i-й строки помещаются в некоторое место, а ссылка на это место хранится в name[i]. Так как размер массива name не указан, транслятор сам считает инициирующие значения и подставляет нужный размер.

6.10. ССЫЛКИ И МНОГОМЕРНЫЕ МАССИВЫ

При первом знакомстве с языком Си иногда возникают неясности с толкованием различия между двумерным массивом и массивом ссылок, таким, как name в только что приведенном примере. Если есть описания:

```
int a[10][10];
int *b[10];
```

то a и b используются аналогичным образом, т. е. a [5][5] и b [5][5] представляют собою допустимые обращения к единственному целому значению. Однако a — это действительно массив, для него выделены все 100 ячеек памяти, и для определения нужного элемента выполняются обычные индексные вычисления, учитывающие «прямоугольность» массива. В то же время при описании b выделяются в памяти только 10 ссылок, и каждую из них нужно установить так, чтобы она указывала на массив целых величин. Если предположить, что каждая ссылка указывает на массив из десяти элементов, то всего будет занято 100 ячеек памяти под эти массивы плюс еще десять ячеек на ссылки. Таким образом, массивы ссылок* занимают чуть больше памяти и могут требовать явной инициации. Но такие массивы обладают двумя преимуществами: доступ к элементам идет косвенно через ссылку и не требует умножения и сложения, кроме того, строки массива (векторы) могут быть различного размера. Например, каждый элемент b не обязательно указывает на вектор из десяти элементов, иногда в нем будет два элемента, иногда—двенадцать, а иногда элементов вообще нет.

Хотя мы и обсуждали эту проблему в терминах целых значений, но чаще всего используются массивы ссылок в ситуациях, подобных функции month_name: для хранения символьных строк различного размера.

Упражнение 6.6. Перепишите программы day_of_year и month_day, используя вместо индексации ссылки.

6.11. ВНЕШНИЕ АРГУМЕНТЫ

Язык Си обычно используется в окружении, где существует возможность передавать программе в начале ее выполнения некоторые аргументы или параметры из задания, написанного на языке управления заданиями. Когда, начиная вычисления, мы обращаемся к программе main, ей передаются два аргумента. Первый, его удобно называть argc,—число командных аргументов при обращении к программе, а второй (argv)— ссылка на массив символьных строк, содержащих эти аргументы; в одной строке один аргумент. Работа с такими символьными строками обычно идет с помощью многоуровневых ссылок.

Самой простой иллюстрацией того, какие в этом случае должны быть описания и как ими пользоваться, представляется программа echo. Она просто «возвращает» командные аргументы, печатая их в строку, разделяя пробелами. Таким образом, если дается команда

```
echo hello, world
```

---

*Точнее было бы сказать, массивы со «ссылками». — *Примеч. пер.*

то результатом будет печать

```
hello, world
```

По соглашению argv[0] — имя, под которым вызывалась программа, так что argc по крайней мере равно 1. В нашем же примере argc=3, а argv[0], argv[1], argv[2] — соответственно строки «echo», «hello», «word». Первый фактический аргумент находится в argv[1], а последний — в argv[argc-1]. Если argc равно 1, то после имени программы командных аргументов нет. Теперь сама программа echo. Выглядит она так:

```
main(argc, argv)     /* вернуть аргументы; 1-я версия */
int argc;
char *argv[];
{
     int i;

     for (i = 1; i < argc; i++)
          printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

Так как argv — ссылка на массив ссылок, то есть несколько способов написать такую программу, манипулируя массивами с помощью ссылок, а не индексов. Вот еще два варианта:

```
main(argc, argv)     /* вернуть аргументы; 2-я версия */
int argc;
char *argv[];
{
     while (--argc > 0)
          printf("%s%c", *++argv, (argc > 1) ? ' ' : '\n');
}
```

Поскольку argv ссылка на начало массива ссылок-аргументов, то, увеличивая его на 1 (++argv), мы передвигаемся от argv[0] в точку argv[1]. И каждое последующее увеличение передвигает нас к следующему аргументу; *argv в этом случае — ссылка на этот аргумент. В то же время мы уменьшаем argc. Как только она станет равной нулю, аргументов, которые нужно печатать, больше не останется.

```
main(argc, argv)     /* вернуть аргументы; 3-я версия */
int argc;
char *argv[];
{
     while (--argc > 0)
          printf((argc > 1) ? "%s " : "%s\n", *++argv);
}
```

В этом варианте демонстрируется, что форматный аргумент для printf может, как любой другой, быть выражением. Хотя такое представление встречается не очень часто, его стоит запомнить.

В качестве второго примера, давайте проведем некоторое улучшение в программе опознания образа из гл. 5. Если помните, мы искомый образ «втянули» глубоко в программу, и это было, очевидно, плохое решение. Теперь, следуя за обслуживающей программой grep из UNIX, давайте изменим программу так, чтобы опознаваемый образ задавался первым аргументом из командной строки.

```
#define   MAXLINE   1000

main(argc, argv)   /* поиск образа из первого аргумента */
int argc;
char *argv[];
{
     char line[MAXLINE];

     if (argc != 2)
          printf("Usage: find pattern\n");
     else
          while (getline(line, MAXLINE) > 0)
               if (index(line, argv[1]) >= 0)
                    printf("%s", line);
}
```

Теперь для иллюстрации дальнейших приемов работы со ссылками можно развить такую основную модель работы. Предположим, мы хотим допустить использование двух (если нужно) аргументов. Один говорит, что нужно печатать все строки, кроме совпадающих с образом, а второй — что перед каждой строкой нужно указывать ее номер.

По общераспространенному для языка Си соглашению аргумент, начинающийся со знака минус, вводит признак вариантности или параметр. Если мы выберем —x («кроме») как сигнал об обратном действии, а —n («номер») —как требование нумерации строк, то команда

```
find  -x  -n  the
```

при условии, что на вход поступают такие строки:

```
now is the time
for all good men
to come to the aid
of their party.
```

приведет к такой выдаче:

```
2: for all good men
```

Следовало бы предусмотреть, чтобы возможные аргументы могли идти в любом порядке и оставшаяся часть программы была бы нечувствительной к числу фактически имеющихся аргументов. В частности, при обращении к index нельзя обращаться к argv[2], если был признак одного аргумента, и к argv[1], если его не было. Более того, для пользователя было бы удобно, если бы было можно «соединить» вариантные аргументы и писать, например, так:

```
find -nx the
```

Теперь сама программа:

```
#define   MAXLINE   1000

main(argc, argv)   /* поиск образа из первого аргумента */
int argc;
char *argv[];
{
     char line[MAXLINE], *s;
     long lineno = 0;
     int  except = 0, number = 0;
```

```
        while (--argc > 0 && (*++argv)[0] == '-')
             for (s = argv[0]+1; *s != '\0'; s++)
                  switch (*s) {
                  case 'x':
                       except = 1;
                       break;
                  case 'n':
                       number = 1;
                       break;
                  default:
                       printf("find: illegal option %c\n", *s)
                       argc = 0;
                       break;
                  }
        if (argc != 1)
             printf("Usage: find -x -n pattern\n");
        else
             while (getline(line, MAXLINE) > 0) {
                  lineno++;
                  if ((index(line, *argv) >= 0) != except) {
                       if (number)
                            printf("%ld: ", lineno);
                       printf("%s", line);
                  }
             }
   }
```

Перед каждым вариантным аргументом argv увеличивается, а argc — уменьшается. Если нет ошибок, то в конце цикла argc должно равняться 1, а *argv — указывать на образ. Обратите внимание, что *++argv есть ссылка на строку аргумента, а (*++argv)[0] — ее первый символ. Скобки необходимы; если их не поставить, то получится *++(argv[0]) — нечто другое (и неверное). Допустимой формой того же выражения будет и **++argv.

Упражнение 6.7. Напишите программу add, вычисляющую выражение, записанное в командной строке в обратной польской нотации. Например,

```
add 2 3 4 + *
```

вычисляется как $2 \times (3+4)$.

Упражнение 6.8. Модифицируйте программы entab и detab из гл. 2, чтобы они воспринимали в качестве аргументов список «табулирующих стопов». Если аргументов нет, то устанавливается нормальная табуляция.

Упражнение 6.9. Усовершенствуйте entab и detab так, чтобы они воспринимали сокращения такого типа:

```
entab m +n
```

означающие, что табуляция идет на каждых n столбцах, начиная со столбца m. Выберите удобное (для пользователя) соглашение о подразумеваемом поведении.

Упражнение 6.10. Напишите программу tail, печатающую последние n строк входной информации. По умолчанию n равно 10, но его можно изменить вариантным аргументом, поэтому

```
tail -n
```

печатает последние n строк. Программа должна работать рационально независимо от того, сколь бессмысленна входная информация или значение n. Напишите программу так, чтобы она наилучшим образом использовала бы доступную память: строки нужно хранить так, как в sort, а не в двумерном массиве фиксированного размера.

6.12. ССЫЛКИ НА ФУНКЦИИ

В Си сама функция не может быть значением переменной, но можно определить *ссылку на функцию*. С этой ссылкой можно уже манипулировать как с переменной: передавать ее функциям, помещать в массивы и т. п. Возможности такой работы мы проиллюстрируем на примере модификации ранее описанной процедуры сортировки. Введем сюда вариантный аргумент — n; если он задан, то входные строки сортируются не в лексикографическом порядке, а в числовом.

Сортировки часто состоят из трех частей — *сравнения*, проверяющего упорядоченность любой пары объектов; *смены*, изменяющей их расположение на обратное, и самого *алгоритма сортировки*, выполняющего сравнения и смены до тех пор, пока не будет установлен требуемый порядок. Алгоритм сортировки не зависит от операции сравнения и смены, поэтому, передавая в этот алгоритм различные функции сравнения и смены, мы можем настраивать программу на различные критерии сортировки. Именно этот подход мы и применим в нашей новой программе сортировки.

Лексикографическое сравнение двух строк, как и раньше, делается с помощью функции strcmp, а смена — с помощью функции swap. Для сравнения двух строк на основе их численного значения и выдачи некоторого кода условия, аналогичного strcmp, нам потребуется подпрограмма numcmp. Все эти три функции описываются в программе main, и ссылки на них передаются в sort. А функция sort обращается, в свою очередь, к этим функциям уже через ссылки. Мы пропускаем обработку ошибок в аргументах и сосредоточиваемся на главном:

```
#define LINES 100  /* максимальное число сортируемых строк */

main(argc, argv)     /* сортировка строк */
int argc;
char *argv[];
{
     char *lineptr[LINES];   /* ссылки на строки */
     int  nlines;            /* число прочитанных строк */
     int  strcmp(), numcmp(); /* функции сравнения */
     int  swap();    /* функции обмена */
     int  numeric = 0;  /* 1 если сортировка числовая */

     if (argc>1 && argv[1][0] == '-' && argv[1][1] == 'n')
          numeric = 1;
     if ((nlines = readlines(lineptr, LINES)) >= 0) {
          if (numeric)
               sort(lineptr, nlines, numcmp, swap);
          else
               sort(lineptr, nlines, strcmp, swap);
          writelines(lineptr, nlines);
     } else
          printf("input too big to sort\n");
}
```

Здесь strcmp, numcmp и swap — адреса функций; так как известно, что они должны быть функциями, в операции & необходимости нет, точно так же, как в этом не было ранее необходимости и в случае имен массивов. Транслятор ориентирован на передачу адресов функций.

Теперь модифицируем функцию sort:

```
sort(v, n, comp, exch)    /* сортировка строк v[0]...v[n-1] */
char *v[];                /* в возрастающем порядке */
int n;
int (*comp)(), (*exch)();
{
     int gap, i, j;

     for (gap = n/2; gap > 0; gap /= 2)
          for (i = gap; i < n; i++)
               for (j = i-gap; j >= 0; j -= gap) {
                    if ((*comp)(v[j], v[j+gap]) <= 0)
                         break;
                    (*exch)(&v[j], &v[j+gap]);
               }
}
```

Обратите особое внимание на описания:

```
int (*comp)()
```

говорит, что comp — ссылка на функцию, возвращающую целое значение. Первая пара скобок необходима, без них

```
int *comp()
```

означало бы, что comp — функция, возвращающая ссылку на целое значение, а это совершенно разные вещи.

Использование comp — в строке программы

```
if ((*comp)(v[j], v[j+gap]) <= 0
```

согласовано с описанием: comp — ссылка на функцию, *comp — сама функция, а

```
(*comp)(v[j], v[j+gap])
```

обращение к этой функции. Все скобки необходимы, ибо они нужным образом выделяют компоненты обращения.

Функцию strcmp, сравнивающую две строки, мы уже приводили. Приведем теперь функцию numcmp для сравнения двух строк по начальным числовым значениям.

```
numcmp(s1, s2) /* числовое сравнение s1 и s2 */
char *s1, *s2;
{
     double atof(), v1, v2;

     v1 = atof(s1);
     v2 = atof(s2);
     if (v1 < v2)
          return(-1);
```

```
        else if (v1 > v2)
                return(1);
        else
                return(0);
}
```

И наконец, последний шаг: вводим функцию swap, меняющую местами две ссылки. Это просто небольшая переделка программы, приведенной в данной главе.

```
swap(px, py)    /* поменять *px и *py */
char *px[], *py[];
{
        char *temp;

        temp = *px;
        *px = *py;
        *py = temp;
}
```

Существуют различные возможности, которые можно вводить в программу сортировки; некоторые из них мы приводим в качестве упражнений.

Упражнение 6.11. Модифицируйте программу sort для работы с вариантом —r, указывающим, что сортировку нужно вести в обратном порядке (реверсировать). Вариант —r, конечно же, должен работать вместе с —n.

Упражнение 6.12. Добавьте вариант —f для объединения верхнего и нижнего регистров, чтобы во время сортировки регистр не учитывался и все данные сортировались бы вместе. Проще — буквы a и A «стоят» рядом, а не разделены целым алфавитным регистром.

Упражнение 6.13. Добавьте вариант —d («словарный порядок»). В этом случае сравниваются только буквы, числа и пробелы. Работает этот вариант вместе с вариантом —f.

Упражнение 6.14. Добавьте возможность работы с полями, т. е. возможность проведения сортировки по некоторым полям внутри строки. Для каждого поля существует свой набор вариантов. (Предметный указатель в этой книге был отсортирован в варианте —df для самих категорий и в варианте —n для номеров страниц.)

Глава 7. ЗАПИСИ

Запись — это объединение одной или более переменных, возможно, различных типов, в одну группу, имеющую для простоты работы единственное имя. (Записи в некоторых языках, скажем в знаменитом ПЛ/1, называются «структурами».)

Традиционным примером записи является запись в платежной ведомости: каждый служащий описывается с помощью множества таких атрибутов, как имя, адрес, номер страхового полиса, заработная плата и т. д. Некоторые из них, в свою очередь, могут быть записями: по несколько компонент имеют имя, адрес и даже заработная плата.

Записи предназначены для организации сложных данных, в частности, в больших программах, ибо во многих ситуациях они позволяют группировать в единое целое родственные между собой переменные и работать с ними как с единым целым, а не с отдельными составляющими. В данной главе мы попытаемся показать, как используются записи. Приводимые нами программы будут побольше, чем прежние, но их размер не будет превышать разумных пределов.

7.1. ОСНОВЫ

Вспомним программы преобразования дат из гл. 6. Дата состоит из нескольких частей, таких, как день, месяц, год и, может быть, еще день года и название месяца. Все эти пять переменных можно поместить в единственную запись вроде такой:

```
struct date {
        int  day;
        int  month;
        int  year;
        int  yearday;
        char mon_name[4];
};
```

Служебное слово struct начинает описание записи, состоящее из списка описаний, заключенных в фигурные скобки. Следом за словом struct идет имя (в данном случае — date), его мы называем *типом* этой *записи*. Тип записи именует такую структуру записи и в дальнейшем может использоваться для сокращения деталей описаний. Переменные, упоминающиеся в записи, называются *элементами*. Элемент записи, тип записи и обычные переменные (не элементы записи) могут иметь одно и то же имя; никаких конфликтов не возникает, так как по контексту всегда можно понять, о чем идет речь. Однако в целях стилевого единства обычно одинаково именуют связанные между собой объекты.

Следом за правой фигурной скобкой, заканчивающей список элементов, может следовать список переменных, так же как и в случае основных типов. Т. е. описание

```
struct { ... } x, y, z;
```

синтаксически аналогично таким описаниям:

```
int x, y, z;
```

в том смысле, что и тот и другой оператор описывают x, y и z как переменные поименованного типа и приводят к выделению для этих переменных памяти.

Описание записи без последующего списка переменных не выделяет никакой памяти; оно просто описывает *шаблон* или форму записи. Если такое описание вводит еще и имя типа, то это имя можно позже использовать при определении фактических экземпляров этой записи. Например, имея указанное выше описание даты, мы можем пользоваться такими описаниями, как

```
struct date d;
```

для определения переменной со структурой, аналогичной структуре*, введенной под именем date. Внешние или статические записи можно инициировать, помещая следом за определением список начальных значений компонент:

```
struct date d ={ 4, 7, 1776, 186, "Jul" };
```

Элементы отдельной записи в выражениях обозначаются с помощью такой конструкции:

имя-записи. элемент

Операция выделения элемента . связывает имя записи и имя элемента. Например, указывает ли переменная d на високосный год или нет, определяется так:

```
leap = d.year % 4 == 0 && d.year % 100 != 0
          || d.year % 400 == 0;
```

Сравнение названия месяца делается так:

```
if (strcmp(d.mon_name, "Aug") == 0) ...
```

Преобразование первого символа названия месяца к нижнему регистру:

```
d.mon_name[0] = lower(d.mon_name[0]);
```

Записи можно вкладывать одна в другую, запись о заработной плате фактически могла быть и такой:

---

* Здесь structure начинает путаться со служебным термином "структура чего-то" (строение).—*Примеч. пер.*

```
struct person {
     char name[NAMESIZE];
     char address[ADRSIZE];
     long zipcode;
     long ss_number;
     double salary;
     struct date birthdate;
     struct date hiredate;
};
```

Запись типа person содержит две даты. Если мы теперь опишем emp как

```
struct person emp;
```

то

```
emp.birthdate.month
```

относится к месяцу рождения. Операции выделения элемента «.» выполняются слева направо.

7.2. ЗАПИСИ И ФУНКЦИИ

С записями в языке Си связано несколько ограничений. Наиболее существенное заключается в том, что все, что можно делать с записью,— это взять ее адрес с помощью операции & и обращаться к ее элементам, т. е. записи нельзя копировать или присваивать как целое, их нельзя передавать в функцию или получать оттуда. (В последующих версиях такие ограничения будут сняты.) Однако на ссылки на записи эти ограничения не распространяются, и поэтому записи довольно удобно сочетаются с функциями. И наконец, автоматические записи подобно автоматическим массивам нельзя инициировать, хотя инициация внешних или статических записей вполне допустима.

Для того чтобы разобраться в этих особенностях, перепишем функции преобразования даты из предыдущей главы с учетом использования записей. Поскольку правила запрещают непосредственно передавать записи функциям, мы должны либо передавать по отдельности компоненты, либо передавать ссылку на запись как на единое целое. Первый вариант мы уже разбирали в гл. 6, при этом обращение выглядит так:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

Другой вариант — передать ссылку. Если мы опишем hiredate как

```
struct date hiredate;
```

и перепишем day_of_year, то сможем обращаться:

```
hiredate.yearday = day_of_year(&hiredate);
```

при этом в day_of_year передается ссылка на hiredate. Саму функцию

нужно изменить, так как ее параметр теперь ссылка, а не список переменных:

```
day_of_year(pd)   /* выдать номер дня по месяцу и числу */
struct date *pd;
{
     int i, day, leap;

     day = pd->day;
     leap = pd->year % 4 == 0 && pd->year % 100 != 0
               || pd->year % 400 == 0;
     for (i = 1; i < pd->month; i++)
          day += day_tab[leap][i];
     return(day);
}
```

Описание

```
struct date *pd;
```

говорит, что pd — ссылка на запись типа date. Обозначения вида

```
pd->year
```

для нас новые. Если p — ссылка на запись, тогда

p — > *элемент структуры*

относится к этому конкретному элементу. (Операция — > записывается как минус, за которым следует >.)

Поскольку pd ссылается на запись, к элементу year можно было бы обращаться и так:

```
(*pd).year
```

Однако так как ссылки на записи используются очень часто, то в качестве удобного сокращения предложена операция — >. В выражении (*pd).year скобки обязательны из-за того, что приоритет операции выделения элемента «.» выше, чем у операции «*». Поскольку операции — > и . выполняются слева направо, выражения:

```
p->q->memb
emp.birthdate.month
```

означают:

```
(p->q)->memb
(emp.birthdate).month
```

Для полноты приводим и другую функцию, month_day, переписанную с использованием записей:

```
month_day(pd)   /* выдать месяц и число по номеру дня */
struct date *pd;
{
     int i, leap;

     leap = pd->year % 4 == 0 && pd->year % 100 != 0
               || pd->year % 400 == 0;
     pd->day = pd->yearday;
     for (i = 1; pd->day > day_tab[leap][i]; i++)
          pd->day -= day_tab[leap][i];
     pd->month = i;
}
```

Операции над записями —> и . вместе с () для списка параметров и [] для индексации находятся на вершине иерархии приоритетов и поэтому сильно связаны друг с другом. Например, если есть такое описание:

```
struct {
     int  x;
     int  *y;
} *p;
```

то выражение ++p—>x увеличивает x, а не p, так как неявные скобки заставляют рассматривать это выражение как ++(p—>x). Для изменения порядка действий можно использовать скобки (явные): (++p—>x увеличивает p до обращения к x, а (p++)—>x увеличивает p после обращения. В последнем случае скобки не обязательны. Почему?

Аналогично *p—>y обращается туда, куда указывает y; *p—>y++ увеличивает y после обращения к тому, на что он указывает (точно так же, как и *s++); (*p—>y)++ увеличивает то, на что указывает y, а *p++—>y увеличивает p после обращения к тому, на что указывает y.

7.3. МАССИВЫ ЗАПИСЕЙ

Записи особенно подходят для работы с массивами «родственных» переменных. Рассмотрим, например, программу для подсчета вхождений каждого из служебных слов* языка Си. Нам потребуется массив символьных строк для хранения имен и массив целых чисел — счетчиков. Можно, конечно, использовать два параллельных массива:

```
char *keyword[NKEYS];
int  keycount[NKEYS];
```

Однако сам факт, что массивы параллельны, подсказывает возможность и другой их организации. Фактически каждое служебное слово вводит пару объектов:

```
char *keyword;
int  keycount;
```

---

*Чтобы не переделывать программу, мы будем считать, что программа подсчитывает вхождения английских служебных слов. — *Примеч. пер.*

и есть массив таких пар. Описание записи

```
struct key {
     char *keyword;
     int  keycount;
} keytab[NKEYS];
```

определяет массив keytab записей такого типа и выделяет для них память. Каждый элемент массива — запись. Можно было бы воспользоваться и такими описаниями:

```
struct key {
     char *keyword;
     int  keycount;
};

struct key keytab[NKEYS];
```

Поскольку запись keytab фактически содержит множество постоянных имен, то лучше ее в момент определения раз и навсегда инициировать. Инициация записей совершенно аналогична тому, с чем мы уже имели дело,—за определением следует в фигурных скобках список начальных значений:

```
struct key {
     char *keyword;
     int  keycount;
} keytab[] ={
     "break", 0,
     "case", 0,
     "char", 0,
     "continue", 0,
     "default", 0,
     /* ... */
     "unsigned", 0,
     "while", 0
};
```

Перечисленные парами начальные значения соответствуют элементам записи. Можно было бы быть более пунктуальными и заключать в скобки начальные значения для каждой строчки или записи:

```
{ "break", 0 },
{ "case", 0 },
...
```

Однако, если начальные значения — простые переменные или символьные строки, и причем они все присутствуют, то во внутренних фигурных скобках нет нужды. Как обычно, транслятор, если есть инициация, подсчитывает число элементов в массиве keytab, и поэтому можно писать «пустой» размер — [].

Программа подсчета служебных слов начинается с определения keytab. Главная программа читает входную информацию, обращаясь каждый раз к функции getword, а та уже читает по одному слову. Каждое слово ищется в keytab с помощью варианта функции поиска деления пополам, описанного в гл. 4. Чтобы она могла работать,

список служебных слов задается уже упорядоченным (в возрастающем порядке):

```
#define   MAXWORD   20

main()    /* подсчет служебных слов Си */
{
     int  n, t;
     char word[MAXWORD];

     while ((t = getword(word, MAXWORD)) != EOF)
          if (t == LETTER)
               if ((n = binary(word, keytab, NKEYS)) >= 0)
                    keytab[n].keycount++;
     for (n = 0; n < NKEYS; n++)
          if (keytab[n].keycount > 0)
               printf("%4d %s\n",
                  keytab[n].keycount, keytab[n].keyword);
}

binary(word, tab, n)  /* поиск слова в  tab[0]...tab[n-1] */
char *word;
struct key tab[];
int n;
{
     int low, high, mid, cond;

     low = 0;
     high = n - 1;
     while (low <= high) {
          mid = (low+high) / 2;
          if ((cond = strcmp(word, tab[mid].keyword)) < 0)
               high = mid - 1;
          else if (cond > 0)
               low = mid + 1;
          else
               return(mid);
     }
     return(-1);
}
```

К функции getword мы еще вернемся, сейчас же достаточно сказать, что она, обнаружив слово, выдает константу LETTER и копирует слово в свой первый параметр.

Величина NKEYS — это число служебных слов в keytab. Хотя мы могли бы сосчитать их «вручную», но легче и надежнее поручить это сделать машине, особенно если список может измениться. Можно было бы поступить и совсем по-другому: в конце списка начальных значений поставить нулевую ссылку, а цикл по keytab вести до обнаружения этого конца.

Однако так поступать не стоит, так как размер массива полностью определяется в момент трансляции. Число строк в массиве равно

размер keytab/*размер* key

В языке есть унарная операция sizeof, выполняющаяся во время трансляции; ею можно пользоваться для вычисления размера любого объекта.

Выражение sizeof (*объект*) дает целое значение, равное размеру указанного объекта. (Размер выдается в некоторых единицах, называемых «байтами»; это размер для величин типа char.) Объект может быть и фактической переменной, и массивом, и записью, и именем некоторого основного типа вроде int или double, и даже именем производного типа, скажем некоторой записи. В нашем случае число служебных слов равно размеру массива, деленному на размер одного элемента. Эти вычисления выполняются в операторе #define, устанавливающем значение NKEYS:

```
#define  NKEYS  (sizeof(keytab) / sizeof(struct key))
```

Вернемся к функции getword. Фактически мы написали более универсальную функцию, чем это нужно для нашей программы, но это не такие уж большие усложнения. Функция getword выдает следующее «слово» из входной информации, причем слово это либо строка букв или цифр, начинающаяся с буквы, либо один единственный символ. Тип обнаруженного объекта указывается возвращаемым функцией значением: LETTER, если обнаружено слово; EOF, если обнаружен конец файла, и, наконец, сам обнаруженный и отличный от букв символ:

```
getword(w, lim)      /* взять очередное слово */
char *w;
int lim;
{
     int c, t;

     if (type(c = *w++ = getch()) != LETTER) {
          *w = '\0';
          return(c);
     }
     while (--lim > 0) {
          t = type(c = *w++ = getch());
          if (t != LETTER && t != DIGIT) {
               ungetch(c);
               break;
          }
     }
     *(w-1) = '\0';
     return(LETTER);
}
```

Здесь используются подпрограммы getch и ungetch, приведенные в гл. 5: когда слово собрано, getword взяла на один символ больше. Чтобы вернуть этот символ обратно на вход, обращаются к ungetch.

Для определения типа каждого отдельного входного символа getword обращается к функции type. Ниже приводится версия этой функции, ориентированная *только на алфавит* ASCII:

```
type(c)    /* тип символа в алфавите ASCII */
int c;
{
     if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
          return(LETTER);
     else if (c >= '0' && c <= '9')
          return(DIGIT);
     else
          return(c);
}
```

Символические константы LETTER и DIGIT могут иметь любые значения, не совпадающие с символами, отличными от буквенно-цифровых и EOF; самое простое взять:

```
#define   LETTER    'a'
#define   DIGIT     '0'
```

Функция getword может работать быстрее, если обращение к функции type заменить на выборку из соответствующего массива type[]. В стандартной библиотеке Си есть макроопределения isalpha и isdigit, которые работают именно так.

Упражнение 7.1. Проведите соответствующие изменения в getword и определите, насколько изменилась скорость работы программы.

Упражнение 7.2. Напишите вариант программы type, не зависящий от множества допустимых символов.

Упражнение 7.3. Напишите вариант программы, который не считает вхождений слов, встречающихся в строках, взятых в кавычки.

7.4. ССЫЛКИ НА ЗАПИСИ

Для иллюстрации некоторых соображений о совместном использовании ссылок и массивов записей вернемся вновь к программе подсчета служебных слов. На этот раз вместо индексов массива будем использовать ссылки.

Внешнее описание для keytab изменять нет необходимости, но функции main и binary нужно модифицировать:

```
main()     /* подсчет служебных слов; вариант со ссылками */
{
     int  t;
     char word[MAXWORD];
     struct key *binary(), *p;

     while ((t = getword(word, MAXWORD)) != EOF)
          if (t == LETTER)
               if ((p=binary(word, keytab, NKEYS)) != NULL)
                    p->keycount++;
     for (p = keytab; p < keytab + NKEYS; p++)
          if (p->keycount > 0)
               printf("%4d %s\n", p->keycount, p->keyword);
}
struct key *binary(word, tab, n) /* поиск слова */
char *word;                      /* в tab[0]...tab[n-1] */
struct key tab[];
int n;
{
     int  cond;
     struct key *low = &tab[0];
     struct key *high = &tab[n-1];
     struct key *mid;

     while (low <= high) {
          mid = low + (high-low) / 2;
          if ((cond = strcmp(word, mid->keyword)) < 0)
               high = mid - 1;
```

```
        else if (cond > 0)
            low = mid + 1;
        else
            return(mid);
    }
    return(NULL);
}
```

Здесь стоит обратить внимание на несколько вещей. Первое, в описании binary нужно указывать, что она выдает ссылку на запись типа key, а не целое число; этот факт описывается и в main, и в binary. Если binary находит слово, то выдается ссылка на него; если же слово не обнаружено, то выдается NULL.

Второе, все обращения к элементам keytab делаются с помощью ссылок. Это приводит к одному важному изменению в binary: средний элемент нельзя теперь вычислять по простой формуле:

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

так как сложение двух ссылок не дает сколь-либо осмысленного ответа (даже если его и поделить на 2), и фактически это запрещенное действие. Такое выражение нужно заменить на

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

Оно присваивает mid ссылку на элемент, лежащий на полпути между low и high.

Стоит обратить внимание, как записываются начальные значения для low и high. Ссылки можно инициировать адресами предварительно определенных объектов, точно так же, как мы это уже делали. В main мы писали:

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Если p — ссылка на некоторую запись, то арифметические действия со значением p учитывают фактический размер записи; поэтому p++ увеличивает ссылку так, что она указывает на следующий элемент в массиве записей. Но не думайте, что размер записи есть сумма размеров ее элементов: так как для различных объектов может потребоваться выравнивание в памяти, в записи могут оказаться «дыры».

И последнее, может быть, несколько «постороннее», замечание о формате программы. Если функция выдает значение усложненного типа, как в

```
struct key *binary(word, tab, n)
```

то имя самой функции «смотрится» довольно плохо и для его обнаружения нужен текстовый редактор. Поэтому иногда такие описания имеют следующий вид:

```
struct key *
binary(word, tab, n
```

Однако это все-таки дело вкуса; выберите форму, которая вам нравится, и следуйте ей всегда.

Предположим, мы хотим составить программу для более общей, чем ранее, постановки задачи о подсчете вхождения любых слов во входной информации. Так как список слов мы заранее не знаем, мы его не можем отсортировать и поэтому не можем пользоваться поиском метода деления пополам. Мы даже не можем использовать линейный просмотр для определения, появлялось уже некоторое слово или нет: программа должна работать эффективно. (Более точно, ожидаемое время ее работы должно расти пропорционально квадрату числа слов во входной информации.) Как же нужно организовать данные, чтобы эффективно работать со списком произвольных слов?

Одно из решений заключается в том, чтобы постоянно держать список слов в упорядоченном состоянии, помещая каждое из слов по мере его появления в надлежащее место. Конечно, этого не следует делать, сдвигая слова в линейном массиве, так как мы опять же придем к большим потерям. Вместо этого мы воспользуемся структурой данных, называемой *двоичным деревом.*

В таком дереве различным словам соответствуют различные вершины, причем каждая вершина состоит из:

ссылки на «текст» самого слова;
счетчика числа вхождений;
ссылки на левую «сыновнюю» вершину;
ссылки на правую «сыновнюю» вершину.

Ни одна вершина не может иметь более двух сыновей, но может иметь и одного, и даже не иметь их вовсе.

Дерево поддерживается в таком состоянии, что левое поддерево содержит только слова, меньшие, чем слово в вершине дерева, а правое поддерево — только большие слова. Для того чтобы определить, находится ли вновь появившееся слово в дереве или нет, надо, начиная с основания, сравнивать его со словами, хранящимися в вершинах. Если есть совпадение, вопрос исчерпан. Если новое слово меньше слова в дереве, поиск продолжается в левом поддереве, иначе — в правом. Если поддерева в требуемом направлении нет, то нового слова в дереве нет, и мы сразу же получаем место, куда его нужно поместить, образуя новое поддерево. Этот процесс по существу рекурсивен, поскольку поиск в любой из вершин использует поиск в одном из сыновних поддеревьев. Поэтому для включения и печати слов наиболее естественно использовать рекурсивные подпрограммы.

Вернемся вновь к описанию вершины, это простая запись с четырьмя компонентами:

```
struct tnode {          /* основная вершина */
    char *word;         /* ссылка на текст */
    int  count;         /* число вхождений */
    struct tnode *left;         /* левое поддерево */
    struct tnode *right;        /* правое поддерево */
};
```

Может быть такое «рекурсивное» описание вершины и выглядит несколько рискованным, но фактически оно совершенно правильное. Запись не может включать саму себя, но ведь и

```
struct tnode *left;
```

описывает left как ссылку на вершину, а не саму вершину.

Сама программа удивительно небольшая, правда она использует как вспомогательные уже написанные нами программы. С помощью getword вводится новое входное слово, программа alloc обеспечивает место для запоминания этого слова.

Главная программа просто читает с помощью getword слова и размещает их в дереве с помощью tree:

```
#define   MAXWORD   20

main()    /* подсчет частот слов */
{
     struct tnode *root, *tree();
     char word[MAXWORD];
     int  t;

     root = NULL;
     while ((t = getword(word, MAXWORD)) != EOF)
          if (t == LETTER)
               root = tree(root, word);
     treeprint(root);
}
```

Сама программа tree довольно проста. Главная программа задает ей слово и верхний уровень (основание) дерева. На каждом этапе это слово сравнивается со словом, уже хранящимся в этой вершине, и отправляется к левому или правому поддереву (рекурсивным обращением к программе tree). В конце концов слово совпадет с каким-либо словом, находящимся в дереве (в этом случае счетчик будет увеличен), либо встретится нулевая ссылка, что говорит о том, что должна быть создана и включена в дерево новая вершина. Если создается новая вершина, то tree выдает ссылку на нее, которую ставят в нужное место «родительской» вершины:

```
struct tnode *tree(p, w) /* занести w в дерево p */
struct tnode *p;
char *w;
{
     struct tnode *talloc();
     char *strsave();
     int  cond;

     if (p == NULL) {     /* пришло новое слово */
          p = talloc();   /* новая вершина */
          p->word = strsave(w);
          p->count = 1;
          p->left = p->right = NULL;
     } else if ((cond = strcmp(w, p->word)) == 0)
          p->count++;     /* повторное слово */
     else if (cond < 0)   /* меньшее в левое поддерево */
          p->left = tree(p->left, w);
     else                 /* большее в правое поддерево */
          p->right = tree(p->right, w);
     return(p);
}
```

Память для новой вершины получают, обращаясь к подпрограмме talloc; это просто адаптация ранее приведенной функции alloc. Она выдает ссылку на свободное место для хранения вершины дерева. (В не-

который момент мы еще к ней вернемся.) С помощью strsave новое слово копируется куда-то, счетчик получает начальное значение и устанавливаются нулевые ссылки на двух сыновей. Эта часть программы работает только в случае достижения «кромки» дерева, когда нужно добавлять новую вершину. Контроль за значениями, выдаваемыми из strsave и talloc мы опускаем (чего никак нельзя делать в реальной программе).

Программа treeprint печатает дерево начиная с левых поддеревьев, т. е. в каждой вершине печатаются: левое поддерево (все слова меньше данного), само слово вершины и правое поддерево (все слова больше данного). Если вы ощущаете неуверенность в понимании рекурсии, нарисуйте какое-либо дерево и проследите за его печатью с помощью treeprint; это одна из наиболее «прозрачных» программ, которую мы смогли отыскать:

```
treeprint(p)    /* рекурсивная печать дерева p */
struct tnode *p;
{
     if (p != NULL) {
          treeprint(p->left);
          printf("%4d %s\n", p->count, p->word);
          treeprint(p->right);
     }
}
```

Практическое замечание: если дерево становится «несбалансированным» из-за того, что слова не приходят в случайном порядке, то время работы программы может расти очень быстро. В худшем случае, если слова приходят в некотором порядке, программа будет просто моделировать дорогостоящий линейный поиск. Существуют обобщения двоичных деревьев, а именно 2-3 деревья и AVL деревья, свободные от такого недостатка, но их мы здесь описывать не будем.

Прежде чем покончить с этим примером, стоит сделать некоторое отступление, связанное с программами выделения памяти. Ясно, что в программе должна быть лишь одна программа выделения памяти, даже если ее необходимо выделять для объектов разного вида. Однако, если обработка требований, скажем, для ссылок на символы и для ссылок на записи типа tnode идет в одной программе, возникают два вопроса. Первое, как удовлетворить типичное для большинства реальных машин требование об ограничениях на выравнивание в памяти при размещении объектов некоторых типов (например, целые числа часто должны начинаться с четных адресов). Второе, как нужно копировать описания, учитывая, что alloc по необходимости будет выдавать ссылки разного вида.

Чаще всего требование о выравнивании удовлетворяется легко за счет некоторого проигрыша в памяти: просто предполагается, что программа выделения памяти всегда выдает ссылку, удовлетворяющую *всем* ограничениям на выравнивание. Например, для PDP-11 достаточно всегда выдавать четную ссылку, так как начиная с четных адресов можно располагать любого вида объекты. Это будет обходиться нам потерей пространства для одного символа при запросах на объекты нечетной длины. Подобным же образом можно поступать и на других машинах. Таким образом, реализация функции alloc может оказаться непереносимой, но зато полезной! Программа alloc из гл. 6 не гарантирует какого-либо конкретного выравнивания, но в гл. 9 мы покажем, как правильно построить ее работу.

Вопрос о типе описания для alloc будет мучителен для любого языка, серьезно относящегося к контролю типов. В языке Си самое лучшее — описать alloc как функцию, возвращающую ссылку на символ, а затем уже с помощью операции приведения преобразовывать такую ссылку на нужный тип. Таким образом, если p было описано как

```
char *p;
```

то

```
(struct tnode *) p
```

в любом выражении преобразует ее в ссылку на запись типа tnode. Следовательно, talloc записывается как

```
struct tnode *talloc()
{
    char *alloc()

}   return((struct tnode *) alloc(sizeof(struct tnode)));
```

Для сегодняшних трансляторов этого более чем достаточно, однако и в будущем это останется надежным приемом.

Упражнение 7.4. Напишите программу, читающую программу на языке Си и печатающую в алфавитном порядке группы имен переменных, у которых совпадают первые 7 символов, а различие проявляется где-то дальше. Предусмотрите, чтобы число 7 было параметром.

Упражнение 7.5. Напишите основную программу перекрестных ссылок; она должна печатать список всех слов в документе, а для каждого слова — список номеров строк, где оно встречалось.

Упражнение 7.6. Напишите программу печати различных слов во входной информации, их надо разместить в порядке убывания частоты вхождения. Перед каждым словом печатается значение счетчика.

7.6. ПРОСМОТР ТАБЛИЦ

В этом разделе мы будем составлять ядро пакета для просмотра таблиц, что будет еще одной иллюстрацией работы с записями. Такие подпрограммы типичны для программ работы с таблицами имен (идентификаторов) в макропроцессорах или трансляторах. Возьмем, например, оператор #define из языка Си. Если попадается строка

```
#define  YES
```

то имя YES и заменяющий текст 1 сохраняются в некоторой таблице. Позже, когда имя YES встретится, скажем, в операторе

```
inword = YES;
```

оно должно быть заменено на 1.

Есть две основные программы для манипуляции с именами и замещающими их текстами. Первая, install(s, t), записывает в таблицу имя s и замещающий текст t. И s, и t—строки символов. Вторая, lookup(s), ищет s в таблице и, если имя было обнаружено, выдает ссылку на него; если же имя не было найдено, выдается ссылка NULL.

Мы используем алгоритм, основанный на функциях расстановки: пришедшее имя свертывается в небольшое положительное целое и это целое затем употребляется как индекс для массива ссылок. Элементы массива указывают на начало цепочки блоков, описывающих имена с одинаковыми

свертками. Если ссылка — NULL, то ни одно имя с такой сверткой не приходило.

Любой блок в цепочке — это запись, состоящая из ссылок на имя, на замещающий текст и на следующий блок в этой цепочке. Нулевая ссылка на следующий блок сигнализирует о конце цепочки:

```
struct nlist { /* основная строка таблицы */
     char *name;
     char *def;
     struct nlist *next; /* следующая строка в списке */
{
```

Массив ссылок описывается так:

```
#define   HASHSIZE  100
static struct nlist *hashtab[HASHSIZE]; /* таблица указателей */
```

Функция свертки, используемая в lookup и install, просто складывает значения символов в строке по модулю размера массива. (Конечно, это не самый лучший алгоритм, но его достоинство — крайняя простота.)

```
hash(s)    /* формирование hashval для строки  s */
char *s;
{
     int hashval;

     for (hashval = 0; *s != '\0'; )
          hashval += *s++;
     return(hashval % HASHSIZE);
}
```

Процесс свертки дает начальный индекс в массиве hashtab, если строка в таблице есть, она будет в цепочке из блоков, начинающейся именно здесь. Поиск проводится программой lookup. Если она обнаруживает, что строка уже есть, то выдается ссылка на нее, если нет — выдается NULL:

```
struct nlist *lookup(s)  /* поиск s в */
char *s;
{
     struct nlist *np;

     for (np = hashtab[hash(s)]; np != NULL; np = np->next)
          if (strcmp(s, np->name) == 0)
               return(np);     /* найдена */
     return(NULL);  /* не найдена */
}
```

Программа lookup используется и в install. С ее помощью определяется, не было ли ранее уже установлено имя, о котором идет речь. Если это так, новое определение должно занять место старого. В противном случае формируется полностью новая строка таблицы. Если install выдает NULL, это означает, что по каким-то причинам для новой строки нет места:

```
struct nlist *install(name, def)    /* занести (name, def) */
char *name, *def;                   /*  в  hashtab */
{
     struct nlist *np, *lookup();
     char *strsave(), *alloc();
     int  hashval;

     if ((np = lookup(name)) == NULL) { /* не найдена */
          np = (struct nlist *) alloc(sizeof(*np));
          if (np == NULL)
               return(NULL);
          if ((np->name = strsave(name)) == NULL)
               return(NULL);
          hashval = hash(np->name);
          np->next = hashtab[hashval];
          hashtab[hashval] = np;
     } else          /* уже есть */
          free(np->def); /* освободить старое определение */
     if ((np->def = strsave(def)) == NULL)
          return(NULL);
     return(np);
}
```

Функция strsave просто копирует строку, заданную как параметр, в безопасное место, получаемое при обращении к alloc. Такая программа приводилась в гл. 6. Так как обращения к alloc и free могут следовать в произвольном порядке, то из-за выравнивания эта простая версия alloc здесь не подходит (см. гл. 8 и 9).

Упражнение 7.7. Напишите программу, изымающую имя и его определение из таблицы, управляемой функциями lookup и install.

Упражнение 7.8. Реализуйте простую версию процессора для обработки #define в программах на языке Си. Используйте программы из данного раздела; кроме того, полезными будут и getch и ungetch.

7.7. ПОЛЯ

Если память дефицитна, то может оказаться необходимым упаковывать несколько объектов в одно машинное слово. Особенно часто так поступают, группируя одноразрядные признаки в задачах типа обработки таблиц имен для трансляторов. Форматы данных, существенно присущие машине, определяемые, например, интерфейсом работы со внешними устройствами, также часто требуют возможности выделения частей слова.

Представим себе, что мы имеем дело с фрагментом транслятора, манипулирующим таблицей имен. С каждым идентификатором из программы связана некоторая информация, например: служебное ли это слово или нет; внешнее и/или статическое это имя и т. п. Наиболее компактный способ закодировать эту информацию — представить ее одноразрядными признаками в одном символе или целом.

Обычно это делается путем определения множества «масок», соответствующих последовательным разрядам. Например, так:

```
#define   KEYWORD   01
#define   EXTERNAL  02
#define   STATIC    04
```

(Числа должны быть степенями двойки.) В этом случае обращение к разрядам идет с помощью операций сдвигов, маскирования и отрицания, какие описывались в гл. 3.

Часто появляются даже некоторые идиомы вроде

```
flags |= EXTERNAL | STATIC;
```

здесь «включаются» разряды из переменной flags, соответствующие EXTERNAL и STATIC. Выражение

```
flags &= ~(EXTERNAL | STATIC);
```

«выключает» их, а условие

```
if ((flags & (EXTERNAL | STATIC)) == 0)
```

истинно, если оба разряда равны нулю.

Хотя с такими идиомами легко освоиться, однако в качестве альтернативы в Си предлагается непосредственное, а не с помощью логических операций, определение полей и доступ к ним. Поле — это последовательность соседних разрядов внутри одного целого значения. Синтаксис определения полей и доступ к ним основан на соответствующих понятиях для записи. Например, #define для упомянутой таблицы имен можно заменить таким определением трех полей:

```
struct {
    unsigned  is_keyword : 1;
    unsigned  is_extern : 1;
    unsigned  is_static : 1;
} flags;
```

Здесь определяется переменная с именем flags, содержащая три однораздядных поля. Число, следующее за двоеточием, задает число разрядов в поле. Поля описываются как unsigned, дабы подчеркнуть, что фактически это величина без знака.

Отдельные поля обозначаются теперь как flags.is_keyword или flags.is_extern и т. д., точно так же, как и другие элементы записи. Поля «ведут себя» как небольшие целые числа без знака и подобно другим целым могут входить в арифметические выражения. Таким образом, приведенные выше примеры можно переписать более естественным образом:

```
flags.is_extern = flags.is_static = 1;
```

устанавливает разряды в единицу;

```
flags.is_extern = flags.is_static = 0;
```

устанавливает их в нуль, а

```
if (flags.is_extern == 0 && flags.is_static == 0) ...
```

проверяет их.

Поля не могут переходить через границы целого числа; если размер поля таков, что это должно случиться, то данное поле выравнивается до границы следующего целого числа. Поля не обязательно именовать; безымянные поля (только двоеточие и размер) используются просто как «заполнители». Для принудительного выравнивания к границе следующего целого числа используется специальный размер 0.

Существуют некоторые детали, относящиеся только к полям. Наиболее важная из них связана с тем, что на одних машинах поля размещаются в слове слева направо, а на других—справа налево; все зависит от особенностей аппаратуры. Это означает, что хотя поля и крайне полезны при работе с данными, структура которых определяется самой программой, но, если речь идет о работе с данными, структура которых задана «снаружи», нужно очень внимательно следить, с какого конца слова они начинают размещаться.

Следует помнить и о других ограничениях: поля не имеют знака; их можно хранить лишь в целых (или, что эквивалентно, в целых без знака) переменных; они не могут быть массивами и не имеют адресов, поэтому к ним нельзя применять операцию &.

7.8. СМЕСИ

Смесь — это некоторая переменная, могущая хранить (в разное время) объекты различного типа и размера, причем транслятор сохраняет информацию о размере и требуемом выравнивании. Смеси дают возможность работать в одной и той же области памяти с данными различного вида, не внося в программу какую-либо информацию, зависящую от машины.

В качестве примера возьмем вновь таблицу имен для транслятора. Предположим, что константы могут быть целого или плавающего типа или же ссылками на строки символов. Значение каждой отдельной константы должно храниться в некоторой переменной надлежащего типа, причем с точки зрения управления таблицей наиболее удобно было бы, если бы это значение занимало один и тот же объем памяти и хранилось в одном месте. Для этого и предназначены смеси — они позволяют вводить переменные, могущие сохранять значение любого из нескольких типов. Как и в случае с полями, синтаксис базируется на синтаксисе записей:

```
union u_tag {
    int  ival;
    float fval;
    char *pval;
} uval;
```

Переменная uval достаточно велика, чтобы сохранять самый большой из этих трех типов, причем вне зависимости от машины, для которой идет трансляция, т. е. программа от машины не зависит. Любой из этих трех типов можно присваивать* uval и затем использовать в выражениях до тех пор, пока такое использование имеет смысл: получаемый тип должен быть типом, записанным последний раз. Помнить о том, какой тип сейчас хранится в переменной типа смесь, вменяется в обязанность программисту. Он за это отвечает. Если было записано значение одного типа, а вызывается — другого, результат зависит от машины.

Синтаксически обращение к элементам смеси выглядит так:

имя смеси.элемент

или

ссылка на смесь — > *элемент*

---

*Здесь речь идет, конечно же, о значениях этих типов, однако мы сохраняем «колорит» оригинала. — *Примеч. пер.*

Точно так же, как и для записей. Если для сохранения информации о типе, хранящемся в данный момент в uval, используется переменная utype, то часто в программе можно встретить фрагменты такого вида:

```
if (utype == INT)
     printf("%d\n", uval.ival);
else if (utype == FLOAT)
     printf("%f\n", uval.fval);
else if (utype == STRING)
     printf("%s\n", uval.pval);
else
     printf("bad type %d in utype\n", utype);
```

Смеси могут входить в записи и массивы и наоборот. Нотация при обращении к элементам смеси в записи (или к записи в смеси) совпадает со случаем вложенных записей. Если есть, например, массив из записей, описанный как

```
struct {
     char *name;
     int  flags;
     int  utype;
     union {
          int  ival;
          float fval;
          char *pval;
     } uval;
} symtab[NSYM];
```

то к переменной ival обращаются так:

```
symtab[i].uval.ival
```

К первому же символу в строке pval обращаются через

```
*symtab[i].uval.pval
```

Практически смесь — это запись, все элементы которой имеют нулевое смещение, причем сама запись достаточно велика, чтобы сохранить самый «весомый» элемент, и выравнена так, чтобы можно было работать со всеми смешиваемыми типами. Как и в случае записей, при работе со смесями сейчас можно использовать только обращение к ее отдельным элементам или взятие адреса; смеси нельзя присваивать, передавать функциям и получать от функций. Ссылки же на смеси можно использовать точно так же, как и ссылки на записи.

В гл. 9 на примере выделения памяти мы покажем, как можно использовать ссылки для форсирования выравнивания переменной к границам памяти разного типа.

7.9. ОПРЕДЕЛЕНИЕ ТИПА

В языке Си можно создавать имена для новых типов данных. Для этого используется *определение типа*, например описание

```
typedef int LENGTH;
```

делает имя LENGTH синонимом int. «Тип» LENGTH можно использовать в описаниях, в приведениях и т. д., точно так же, как и тип int:

```
LENGTH    len, maxlen;
LENGTH    *lengths[];
```

Аналогично описание

```
typedef char *STRING;
```

делает имя STRING синонимом char*, т. е. ссылкой на символ, и его можно употреблять в описаниях вроде такого:

```
STRING p, lineptr[LINES], alloc();
```

Обратите внимание, что описываемый тип в операторе typedef появляется не следом за служебным словом typedef, а в позиции имени переменной. Синтаксически служебное слово typedef подобно классу памяти, как extern, static и т. д. Для того чтобы подчеркнуть, что речь идет о имени, мы выделяем его с помощью букв верхнего регистра.

В качестве более сложного примера можно привести определение типа для ветвей дерева, о котором мы уже говорили в этой главе:

```
typedef struct tnode {          /* основная вершина */
     char *word;      /* ссылка на текст */
     int  count;      /* число вхождений */
     struct tnode *left;        /* левое поддерево  */
     struct tnode *right;       /* правое поддерево  */
} TREENODE, *TREEPTR;
```

Такое описание порождает два новых служебных слова для типов: TREENODE (запись) и TREEPTR (ссылка на эту запись). Теперь подпрограмма talloc может быть описана и так:

```
TREEPTR talloc()
{
     char *alloc();

     return((TREEPTR) alloc(sizeof(TREENODE)));
}
```

Необходимо подчеркнуть, что описание typedef не создает никакого нового в любом смысле типа; она просто добавляет новое имя для некоторого существующего типа. Не вводится и новая семантика: описанные таким способом переменные имеют те же самые свойства, которыми они обладали бы в случае явного развернутого описания. Фактически описание typedef подобно #define, за исключением того, что, поскольку оно интерпретируется транслятором, здесь могут быть текстовые подстановки, выходящие за рамки возможностей макропроцессора языка Си. Например,

```
typedef int (*PFI)();
```

порождает тип PFI для «ссылок на функцию, выдающую целое значение». Этот тип можно использовать, скажем, в таком контексте:

```
PFI strcmp, numcmp, swap;
```

в программе сортировки из гл. 6.

Есть две главные причины для использования описания типа. Первая заключается в возможности параметризации программы, ориентированной на перенос. Если для данных, могущих быть зависимыми от машины, используется описание типа, то для переноса нужно переписать только эти типы. Наиболее распространенной ситуацией представляется использование имен, определенных с помощью typedef, для различных целых типов. В этом случае для каждой конкретной машины делается надлежащий выбор среди short, int и long.

Вторая причина заключается в том, что описание типа обеспечивает лучшую документируемость программ. Ведь TREEPTR легче понять, чем ссылку на какую-то сложную запись.

И наконец, всегда есть вероятность, что в будущем транслятор или какая-либо другая программа вроде *lint* будет использовать содержащуюся в определении типа информацию для проведения дополнительного контроля программ.

Глава 8. ВВОД И ВЫВОД

В языке Си не предусмотрены какие-либо возможности для ввода и вывода, именно поэтому мы и откладывали этот разговор до настоящей главы. Как бы то ни было, реальные программы взаимодействуют со своим окружением многими, гораздо более сложными, чем мы упоминали до сих пор, способами. В этой главе мы опишем «стандартную ВВ библиотеку», т. е. множество функций, обеспечивающих стандартную систему ввода-вывода, для программ на Си. Функции проектировались с учетом удобного программирования взаимодействия с ними и отражают только те операции, которые можно найти в большинстве современных операционных систем. Программы достаточно эффективны, и пользователи редко чувствуют необходимость улучшить их «для эффективности», какими бы критичными ни были их задачи. Наконец, эти программы должны быть «переносимыми» в том смысле, что они должны существовать в «совместимом виде» в любой системе, где есть язык Си. Поэтому любая программа, следующая при использовании стандартной библиотеки определенным правилам взаимодействия, может быть перенесена из одной системы в другую практически без изменений.

Мы не будем даже и пытаться описать здесь полностью библиотеку ввода-вывода; нас больше интересует показать суть составления программ, взаимодействующих с окружающей их операционной системой.

8.1. ОБРАЩЕНИЕ К СТАНДАРТНОЙ БИБЛИОТЕКЕ

Любой входной файл, обращающийся к функциям стандартной библиотеки, должен содержать где-то в начале такую строку:

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

Файл stdio.h содержит некоторые макроопределения и переменные, используемые в библиотеке ВВ. Угловые скобки < и >, употребленные вместо обычных двойных кавычек, заставляют транслятор искать файл в каталоге, содержащем информацию о стандартных заголовках. В системе UNIX это обычно /usr/include.

Кроме того, при загрузке программы может оказаться необходимым явно указывать библиотеку. Например, в системе UNIX для PDP-11 команда трансляции программы должна быть такой:

cc *входные файлы и т. д.* —lS

где —lS указывает на загрузку стандартной библиотеки.

8.2. СТАНДАРТНЫЙ ВВОД И ВЫВОД

Самый простой механизм ввода — чтение по одному символу из стандартного источника ввода (обычно это пользовательский терминал) с помощью функции getchar. Обращение getchar( ) при каждом вызове дает очередной, следующий, входной символ. В большинстве окружений, где используется Си, вместо терминала можно подставить любой файл; если программа *prog* использует getchar, то, написав команду

```
prog <infile
```

мы заставим *prog* читать информацию из файла infile, а не с терминала. Переключение ввода делается так, что сама *prog*, очевидно, не изменяется. В частности, строка "<infile" не включается даже в число командных параметров, доступных через argv. Переключение ввода неощутимо и в случае, когда входная информация приходит из другой программы через механизм, называемый «каналом». Команда

```
otherprog | prog
```

выполняет две программы: *otherprog* и *prog*, причем стандартный ввод для *prog* идет из стандартного вывода для *otherprog*.

Функция getchar возвращает значение EOF, если она встречает в текущем вводе признак конца файла. В стандартной библиотеке определяется, что символическая константа EOF равна —1. (В файле stdio.h есть некоторое # define.) Однако все проверки следует писать в терминах EOF, а не —1, дабы не быть зависимыми от специфического значения.

Обращение putchar(c) выдает символ c в стандартный выходной поток, который по умолчанию также считается терминалом. Выходная информация может быть направлена в любой файл; для этого используется >: Если *prog* использует putchar, то команда

```
prog >outfile
```

вызовет выдачу стандартного выходного потока в файл outfile, а не на терминал. В системе UNIX можно пользоваться и каналом

```
prog | anotherprog
```

Стандартный выводной поток из *prog* замыкается на стандартный вводный поток в *anotherprog*. И опять *prog* ничего «не знает» о переключении.

Выдающаяся с помощью printf информация также идет в стандартный выходной поток, и обращения к putchar и к printf можно произвольно чередовать.

Удивительно много программ занимаются чтением только одного вводного потока и порождают только один выводной поток. Работа таких программ может полностью обеспечиваться функциями getchar, putchar и printf, поэтому достаточно начать с них. В частности, это же справедливо и для случая подмены файлов или использования канала для замыкания вывода одной программы на ввод другой. Рассмотрим, например, программу *lower*, преобразующую вводимые символы в соответствующие символы нижнего регистра:

```
#include  <stdio.h>

main()      /* перевод на нижний регистр */
{
      int c;

      while ((c = getchar()) != EOF)
            putchar(isupper(c) ? tolower(c) : c);
}
```

«Функции» isupper и tolower фактически представляют собою макроопределения в stdio.h. Макроопределение isupper проверяет, относится ли ее параметр к буквам верхнего регистра; если это так, то выдается ненулевой ответ, иначе — нуль. Макроопределение tolower переводит букву верхнего регистра на нижний регистр. Как бы ни были реализованы эти функции на конкретной машине, внешнее их поведение остается тем же самым. Поэтому программа, пользующаяся ими, может не знать особенности множества символов.

Для преобразования нескольких файлов можно пользоваться программой типа обслуживающей программы *cat* из UNIX. Она соединяет файлы:

```
cat file1 file2 ... | lower >output
```

и нам уже нет необходимости изучать, как обращаться к файлам из программы. (Программу *cat* мы в данной главе еще опишем.)

Как и другие функции, функции getchar и putchar из стандартной библиотеки ввода-вывода могут быть и макроопределениями, что позволяет избегать расходов на обращение к функции при каждом обращении за символом. Как это делается, мы покажем в гл. 9.

8.3. ФОРМАТНЫЙ ВЫВОД — PRINTF

Две программы: printf для вывода и scanf для ввода (см. следующий раздел) — позволяют переводить числовые величины в символьное представление и обратно. Позволяют они и формировать или интерпретировать форматные строки. Неформально в предыдущих главах мы уже использовали printf. Сейчас мы дадим более точное и полное описание:

```
printf(control, arg1, arg2, ...)
```

Эта функция преобразует, располагает должным образом и печатает свои параметры* через стандартный выходной поток под управлением строки control. Управляющая строка содержит объекты двух типов: обычные символы, которые просто копируются в выходной поток, и спецификации преобразования, каждая их которых вызывает преобразование и печать очередного параметра из обращения к printf.

Каждая спецификация преобразования начинается с символа % и заканчивается некоторым символом, задающим преобразование. Между % и символом преобразования могут встречаться:

знак минус, указывающий, что преобразованный параметр должен быть выравнен влево в своем поле;

строка цифр, задающая минимальный размер поля. Преобразованное число будет напечатано в поле минимум указанного размера. Если в преобразованном параметре символов меньше, чем размещается в таком поле, то слева будут добавлены пробелы (или справа, если задано выравнивание влево). Так что размер поля будет выдержан. Добавляются

* Обратите внимание, что в книге термины «параметр» и «аргумент» автор употребляет как синонимы. — *Примеч. пер.*

обычно пробелы, но если размер поля задан с начальным нулем (этот нуль уже не означает, что размер восьмеричный), то добавляются нули.

точка, отделяющая размер поля от последующей строки цифр;

строка цифр (точность), задающая максимальное число символов, которое нужно напечатать из некоторой строки, или же число цифр, которые нужно печатать справа от десятичной точки в числах типов float или double;

маркер длины l, указывающий, что соответствующий элемент данных относится к типу long, а не просто int.

Символы преобразования и их смысл следующие:

d — параметр преобразуется в десятичное представление;

o — параметр преобразуется в восьмеричное представление без знака и без ведущего нуля;

x — параметр преобразуется в шестнадцатеричное представление без знака (и без ведущего 0x);

h — параметр преобразуется в десятичное представление без знака;

c — параметр рассматривается как один символ;

s — параметр суть строка; символы строки печатаются до тех пор, пока не встретится «нулевой» символ или же не будет напечатано число символов, заданное точностью;

e — параметр рассматривается как величина типа float или double и преобразуется в десятичное представление вида [—]m.nnnnnnE[±]xx, где длина строки из «n» задается точностью. По умолчанию точность равна шести;

f — параметр рассматривается как величина типа float или double и преобразуется в десятичное представление вида [—] mmm.nnnnn, где число «n» задается точностью. По умолчанию точность равна 6. Заметим, что при спецификации f точность не определяет число печатаемых значащих цифр;

g — используется как %e или %f, как сокращение, незначащие нули не печатаются.

Если символ после % не есть символ преобразования, то этот символ печатается. Так, сам % можно напечатать, задав %%. Большинство форматов преобразований очевидны и уже были проиллюстрированы в ранних главах. Исключение представляет лишь понятие точности по отношению к строкам. В приводимой таблице отражены эффекты различных спецификаций при печати строки «hello, world» (12 символов). Поле мы специально обрамляем двоеточием, чтобы были видны его размеры.

```
:%10s:      :hello, world:
:%-10s:     :hello, world:
:%20s:      :        hello, world:
:%-20s:     :hello, world        :
:%20.10s:   :          hello, wor:
:%-20.10s:  :hello, wor          :
:%.10s:     :hello, wor:
```

Предостережение: printf использует первый параметр, чтобы определить, сколько всего параметров и каковы их типы. Программа будет сбиваться, и вы получите абсурдные результаты, если не будет хватать параметров или у них будет неподобающий тип.

Упражнение 8.1. Напишите программу, печатающую произвольную входную информацию, но с некоторыми мерами предосторожности. Как минимум, нужно печатать в восьмеричном или шестнадцатеричном виде (это зависит от местных традиций) символы, не имеющие графического представления. Длинные строчки нужно «загибать».

Функция scanf представляет собою вводный аналог функции printf; в ней предусмотрены многие из упомянутых преобразований, но выполняемые в «обратном» направлении:

```
scanf(control, arg1, arg2, ...)
```

Функция читает символы из стандартного вводного потока, интерпретирует их в соответствии с форматами, заданными в control, и записывает результаты в соответствующие параметры. Управляющий параметр описывается ниже, а оставшиеся параметры, *каждый из которых должен быть некоторой ссылкой,* указывают, где нужно сохранять преобразованную соответствующим образом входную информацию.

Управляющая строка обычно содержит спецификации преобразования и используется для непосредственной интерпретации входной последовательности. В управляющую строку могут включаться:

пробелы, символы табуляции и переходы на новую строку («пустые» символы); эти символы игнорируются;

обычные символы (кроме %); считается, что они должны совпадать с очередными непустыми символами во входном потоке; спецификации преобразования, состоящие из символа %, возможно, символа запрещения присваивания *, возможно, числа, задающего максимальный размер поля, и самого символа преобразования.

Спецификация преобразования управляет преобразованием очередного входного поля. Результат обычно помещается в переменную, на которую указывает соответствующий параметр. Однако, если символ *, говорит, что присваивание запрещается, входное поле просто пропускается, присваивание не происходит. Входное поле определяется как строка непустых символов, ограниченная либо очередным пустым символом, либо размером поля, если он задан. Это предполагает, что scanf будет в поисках вводимых полей просматривать информацию до границы строки, так как конец строки есть пустой символ.

Символ преобразования указывает, как нужно интерпретировать входное поле; соответствующий параметр, как этого требует семантика Си с ее передачей значения, должен быть ссылкой. Допускаются такие символы преобразования:

d — на входе ожидается десятичное целое. Соответствующий параметр должен быть ссылкой на целое;

o — на входе ожидается восьмеричное целое (с начальным нулем или без него). Соответствующий параметр должен быть ссылкой на целое;

x — должно прийти шестнадцатеричное целое (с начальными 0x или без них). Соответствующий параметр должен быть ссылкой на целое;

h — на входе ожидается короткое целое, соответствующий параметр должен быть ссылкой на короткое целое;

c — ожидается появление одиночного символа, параметр должен быть ссылкой на символ. Очередной вводимый символ помещается в указанное место. Обычный пропуск до очередного пустого символа в этом случае не производится. Для чтения следующего не пустого символа используйте %1s;

s — ожидается появление строки символов. Соответствующий параметр должен быть ссылкой на символ, указывающей на достаточно большой массив символов, способный вместить всю строку и заключительный \0, который добавит сама функция;

f — ожидается появление числа с плавающей точкой. Соответствующий параметр должен быть ссылкой на переменную типа float. Символ преобразования e есть синоним для f. Входной формат для float предусматривает, возможно, знак; строку цифр, возможно, с десятичной точкой и, возможно, поле порядка, содержащее символ E или e, за которым идет целое число, может быть со знаком.

Перед символами преобразования d, o и x может стоять буква l, указывающая, что в списке параметров стоит ссылка на переменную типа long, а не просто на int. Если эта же буква стоит перед символами e и f, то в списке параметров речь идет о ссылках на объекты двойной точности, а не просто float.

Например, обращение

```
int  i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name);
```

с входной строкой

```
25   54.32E-1  Thompson
```

присвоит значение 25 переменной i, значение 5.432—x и строку "Thompson" с добавленным концевым\0—переменной name. Поля входной строки могут разделяться многими пробелами, символами табуляции и переходами на новую строку, что кому нравится. Обращение

```
int  i;
float x;
char name[50];
scanf("%2d %f %*d %2s", &i, &x, name);
```

с входной строкой

```
56789  0123  45a72
```

присвоит 56—i, 789.0—x, пропустит 0123 и разместит строку "45" в переменной name. Последующее обращение к любой программе ввода приведет к поиску информации в строке, начиная с буквы a. В этих двух примерах name есть ссылка, и поэтому перед ним *не* должен стоять символ &.

В качестве другого примера можно теперь переписать примитивный калькулятор из гл. 5 и для обработки входной информации использовать scanf:

```
#include  <stdio.h>

main()     /* простейший калькулятор */
{
     double sum, v;

     sum = 0;
     while (scanf("%lf", &v) != EOF)
          printf("\t%.2f\n", sum += v);
}
```

Программа scanf останавливается по исчерпанию управляющей строки либо при несоответствии входной информации управляющим спецификациям. Как результат она выдает число (значение) успешно опознанных и присвоенных входных элементов. В конце файла выдается EOF, это значение отлично от нуля, а нуль означает, что очередной входной символ не соответствует первой спецификации из управляющей строки. Следующее обращение к scanf возобновляет поиск сразу за последним уже считанным символом.

И последнее предупреждение: параметры scanf *должны* быть ссылками. Одна из наиболее распространенных ошибок — это, например, такое обращение:

```
scanf("%d", n);
```

вместо

```
scanf("%d", &n);
```

8.5. ФОРМАТНЫЕ ПРЕОБРАЗОВАНИЯ В ПАМЯТИ

Функции scanf и printf имеют двойников под названиями sscanf и sprintf. Эти программы выполняют те же преобразования, но работают не с файлами, а со строками. Общий вид обращений к ним таков:

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

Функция sprintf приводит аргументы из arg1, arg2 и т. д. к формату, заданному, как и ранее, в строке control, но результат помещает в параметр string, а не в стандартный выходной поток. Конечно, эта строка должна быть достаточно большой, чтобы вместить результат. Например, если name — символьный массив, а n — некоторое целое, то обращение

```
sprintf(name, "temp%d", n);
```

порождает в массиве name строку вида temp*nnn*, где *nnn* — значение n.

Функция sscanf выполняет обратное преобразование — в соответствии с заданным в control форматом она просматривает в string информацию и помещает результирующие величины в arg1, arg2 и т. д. Параметры должны быть ссылками. Обращение

```
sscanf(name, "temp%d", &n);
```

присваивает значение строки цифр, находящейся в name следом за temp, переменной n.

Упражнение 8.2. Перепишите программу калькулятора из гл. 5. Для ввода и преобразования чисел используйте scanf и/или sscanf.

8.6. ДОСТУП К ФАЙЛАМ

Все программы, с которыми мы все время имели дело, читали информацию из стандартного входного потока и записывали ее в стандартный выходной поток, причем мы предполагали, что эти возможности для любой из программ магически предопределены локальной операционной системой.

Следующим шагом в знакомстве со вводом и выводом будет написание программ для работы с файлами, которые уже *не* будут так жестко связаны с программами. Примером программы, где явно проявляется необходимость в таких действиях, может служить программа *cat*. Эта программа объединяет несколько поименованных файлов, причем имена этих файлов поступают из входного потока. *Cat* можно использовать для выдачи файлов на терминальное устройство почти как универсальный входной коллектор для программ, не имеющих возможности обращаться к файлам по именам. Команда

```
cat x.c y.c
```

например, печатает содержимое x.c и y.c на стандартном выходном устройстве.

Суть проблемы заключается в том, как распорядиться именами читаемых файлов, т. е. как связать внешние имена, которыми мыслит пользователь, с операторами, которые фактически выполняют чтение.

Правила просты: прежде чем читать или записывать информацию в файл, его нужно *открыть* с помощью стандартной библиотечной функции fopen. Эта функция берет внешнее представление имени (скажем, x.c или y.c.), что-то делает, как-то связывается с операционной системой (детали всего этого знать нет необходимости) и возвращает некоторое внутреннее имя; оно и используется впоследствии при чтении или записи в файл.

Внутреннее имя — это фактически ссылка (ее называют *ссылкой на файл)* на некоторую запись, где содержится информация о файле: скажем, местоположение буфера, позиция текущего символа в этом буфере, читается файл или записывается и т. п. Подробности пользователю можно не знать, так как в стандартных определениях для ввода-вывода, получаемых из stdio.h, есть определение записи под названием FILE. Однако ссылки на файлы описывать нужно. И это делается, например, так:

```
FILE *fopen(), *fp;
```

что означает: fp — ссылка на FILE, а fopen выдает ссылку на FILE. Заметим, что FILE есть имя типа, подобно int, а не тип записи, и реализуется это через оператор typedef. (Детали того, как все это работает в системе VNIX, приводятся в гл. 9).

Обращение к fopen в программе делается так:

```
fp = fopen(name, mode);
```

Первый параметр fopen — *имя* файла, это строка символов. Второй параметр — *mode* также строка символов, указывающая, как будет использоваться файл. Возможны такие виды использования: чтение ("r"), запись ("w") или дополнение ("a").

Если вы открываете файл для записи или дополнения, причем он еще не существует, то он создается (если это можно сделать). Открытие существующего файла для записи приводит к уничтожению его старого содержимого. Попытка прочитать несуществующий файл — это ошибка, есть и другие аналогичные ошибки. (Скажем, попытка прочитать файл, если он для вас запрещен.) При обнаружении ошибки fopen выдает нулевую ссылку со значением NULL (для удобства она также определена в stdio.h).

Теперь нужно сказать, как читать уже открытый файл или же записывать в него информацию. Есть несколько возможностей, из которых самые простые — функции getc и putc. Getc выдает из файла очередной символ; ей нужно задать лишь ссылку на требуемый файл. Таким образом, оператор

```
c = getc(fp)
```

помещает в c очередной символ из файла, указанного с помощью fp. Если же достигнут конец файла, то в c получим EOF.

Функция putc по смыслу противоположна getc, обращение

```
putc(c, fp)
```

заносит символ c в файл fp и возвращает c. Как и getchar и putchar, getc и putc могут быть не функциями, а макроопределениями.

С началом работы любой программы автоматически открываются три файла, и для них предусматриваются ссылки на файлы. Эти стандарт-

ный вход, стандартный выход и стандартный выход для ошибок, соответствующие ссылки на файлы называются stdin, stdout, stderr. Обычно все они связаны с терминальным устройством, однако stdin и stdout можно связать и с файлами, и с каналами (см. разд. 8.2).

Getchar и putchar можно определить в терминах getc, putc, stdin и stdout следующим образом:

```
#define getchar()   getc(stdin)
#define putchar(c)  putc(c, stdout)
```

Для форматного ввода или вывода в файл можно использовать функции fscanf и fprintf. Они идентичны scanf и printf, но только первый параметр — это ссылка на файл, задающая читаемый или записываемый файл, а управляющая строка идет как второй параметр.

Получив эти предварительные сведения, мы теперь можем приступить к написанию программы *cat*, объединяющей файлы. В ее основе лежит схема, удобная для многих программ: если в строке-команде есть параметры, то они обрабатываются по очереди. Если параметров нет, то обрабатывается информация из стандартного входного потока. При таком построении программу можно использовать и как автономную, и как фрагмент некоторого большого процесса:

```
#include  <stdio.h>

main(argc, argv)     /* cat: конкатенация файлов */
int argc;
char *argv[];
{
     FILE *fp, *fopen();

     if (argc == 1) /* аргументов нет; копирование стандартного входа */
          filecopy(stdin);
     else
          while (--argc > 0)
               if ((fp = fopen(*++argv, "r")) == NULL) {
                    printf("cat: can't open %s\n", *argv);
                    break;
               } else {
                    filecopy(fp);
                    fclose(fp);
               }
}

filecopy(fp)    /* копирование файла fp в стандартный выход */
FILE *fp;
{
     int c;

     while ((c = getc(fp)) != EOF)
          putc(c, stdout);
}
```

Ссылки на файлы stdin и stdout в библиотеке ввода-вывода предопределены как ссылки на стандартные входной и выходной потоки; их

можно использовать везде, где допускаются объекты типа FILE*. Однако это константы, а *не* переменные, и не пытайтесь им что-либо присваивать.

Функция fclose противоположна fopen; она разрывает связь между ссылкой на файл и внешним именем, связь, которая была установлена fopen. Теперь ссылка на файл свободна, и ее можно использовать для другого файла. Так как большинство операционных систем ограничивают число одновременно открытых файлов для программы, лучше освобождаться от всего, что уже не нужно. В *cat* мы так и поступаем. Кроме того, для выходных файлов именно fclose опустошает буфер, где putc накапливает информацию. (При нормальном окончании программы для каждого из открытых файлов идет автоматическое обращение к fclose.)

8.7. РЕАКЦИЯ НА ОШИБКИ—STDERR И EXIT

Реакция на ошибки в *cat* выглядит далеко не идеальной. Дело в том, что, если какой-либо из файлов по некоторым причинам окажется недоступным, диагностика будет печататься лишь в конце объединенного файла. Если выдача идет на терминал, то это вполне приемлемо, однако такое решение будет плохим, если выдача идет прямо в файл или через канал в другую программу.

Для таких ситуаций вводится второй выходной файл, его называют stderr, и он связывается с любой из программ аналогично файлам stdin и stdout. Что бы ни случилось, информация, записанная в stderr появляется на пользовательском терминале, даже если стандартный выходной поток от терминала отключен.

Переделаем программу *cat* так, чтобы она выдавала сообщение об ошибке в стандартный файл ошибок:

```
#include  <stdio.h>

main(argc, argv)     /* конкатенация файлов  */
int argc;
char *argv[];
{
     FILE *fp, *fopen();

     if (argc == 1) /* аргументов нет; копирование стандартного входа */
          filecopy(stdin);
     else
          while (--argc > 0)
               if ((fp = fopen(*++argv, "r")) == NULL) {
                    fprintf(stderr,
                         "cat: can't open %s\n", *argv);
                    exit(1);
               } else {
                    filecopy(fp);
                    fclose(fp);
               }
     exit(0);
}
```

Об ошибках программа сигнализирует двумя способами. Диагностическое сообщение, выдаваемое fprintf, идет в stderr, т. е. попадает на пользовательский терминал, а не пропадает где-то в канале или в каком-либо выходном файле.

Кроме этого, в программе используется и стандартная библиотечная функция exit, заканчивающая при обращении к ней выполнение програм-

мы. Аргумент при exit доступен любому процессу, обратившемуся к данному, поэтому результат работы (успех или неуспех) программы может быть проверен в программе, использовавшей данную как подпроцесс. По соглашению, если возвращается значение 0, все прошло хорошо; отличное же от нуля значение сигнализирует о ненормальной ситуации.

Функция exit обращается для каждого открытого выходного файла к fclose, опустошая тем самым любые выходные буфера, а затем идет обращение к программе с именем _exit. Эта функция немедленно заканчивает программу, не трогая каких-либо буферов. Если нужно, к _exit можно обратиться и непосредственно.

8.8. ВВОД И ВЫВОД СТРОК

В стандартной библиотеке есть программа fgets, очень похожая на функцию getline, которую мы уже в этой книге использовали. Обращение

```
fgets(line, MAXLINE, fp)
```

читает из файла fp в символьный массив line очередную входную строчку (включая символ новой строки). Может быть считано самое большое MAXLINE — 1 символов. Получившаяся строка оканчивается символом \0. Обычно из fgets выдается line, а в конце файла выдается NULL. (Наша getline выдает длину строчки, а в конце файла — нуль.)

При выводе функция fputs записывает в файл целую строку (не обязательно даже содержащую символ новой строки):

```
fputs(line, fp)
```

Чтобы продемонстрировать, что в функциях вроде fgets и fputs нет ничего таинственного, мы их здесь приводим точно в таком виде, в каком они находятся в библиотеке ввода-вывода:

```
#include  <stdio.h>

char *fgets(s, n, iop)   /* взять из iop не больше n символов */
char *s;
int n;
register FILE *iop;
{
     register int c;
     register char *cs;

     cs = s;
     while (--n > 0 && (c = getc(iop)) != EOF)
          if ((*cs++ = c) == '\n')
               break;
     *cs = '\0';
     return((c == EOF && cs == s) ? NULL : s);
}
fputs(s, iop)   /* выдать строку s в файл iop */
register char *s;
register FILE *iop;
{
     register int c;

     while (c = *s++)
          putc(c, iop);
}
```

Упражнение 8.3. Напишите программу для сравнения двух файлов и печати первой строчки и местоположение первого символа, где обнаружено расхождение.

Упражнение 8.4. Модифицируйте программу опознания образа из гл. 6. Она должна теперь работать либо с группой поименованных файлов, либо, если такие файлы, как параметры, не указаны, — со стандартным входным файлом.

Упражнение 8.5. Напишите программу печати группы файлов. Каждый файл надо начинать с новой страницы, печатать заголовок файла и вести «собственную» нумерацию страниц.

8.9. НЕКОТОРЫЕ ПОЛЕЗНЫЕ ФУНКЦИИ

В стандартной библиотеке предусмотрено много функций, однако лишь немногие из них оказались очень полезными. Мы уже упоминали функции работы со строками: strlen, strcpy, strcat и strcmp. Теперь дополним этот список.

Проверки классов символов и преобразования

Существует несколько макроопределений для проверки символов и их преобразования:

isalpha (c) ненулевое значение, если с буква, иначе — 0;
isupper (c) ненулевое значение, если с на верхнем регистре, иначе — 0;
islower (c) ненулевое значение, если с на нижнем регистре, иначе — 0;
isdigit (c) ненулевое значение, если с цифра, иначе — 0;
isspace (c) ненулевое значение, если с пробел, символ табуляции или новой строки, иначе — 0;
toupper (c) перевод с на верхний регистр;
tolower (c) перевод с на нижний регистр.

Ungetc

В стандартной библиотеке есть несколько упрощенный вариант функции ungetch, о которой говорилось в гл. 4; она называется ungetc. Обращение

```
ungetc(c, fp)
```

прячет символ с обратно в файл fp. Вернуть в файл можно только один символ. Функцию можно использовать с любыми функциями ввода или макроопределениями вроде scanf, getc или getchar.

Обращение к системе

Функция system(s) выполняет команду, содержащуюся в символьной строке s, а затем продолжает выполнение текущей программы. Содержимое s строго взаимосвязано с локальной операционной системой. Тривиальный пример: в системе UNIX обращение

```
system("date");
```

вызовет выполнение программы date, а она напечатает дату и текущее время.

Функция calloc похожа на alloc, использовавшуюся в предыдущих главах. Обращение

```
calloc(n, sizeof(object))
```

выдает ссылку на память, достаточную для размещения n объектов заданного размера. Если требование выполнить нельзя, выдается NULL. Память инициируется «нулями».

Ссылка соответствующим образом выравнена, но ее необходимо приводить к нужному типу. Например:

```
char *calloc();
int *ip;

ip=(int *) calloc(n, sizeof(int));
```

cfree (p) освобождает память, на которую указывает p, причем p первоначально было получено обращением к calloc. Ограничений на порядок освобождения нет, однако попытка освободить память, не полученную путем обращения к calloc, считается ужасной ошибкой.

В следующей главе мы покажем, как сделать программу типа calloc, позволяющую выделять блоки памяти, которые можно освобождать в любом порядке.

Глава 9. ВЗАИМОДЕЙСТВИЕ С СИСТЕМОЙ UNIX

Материал данной главы относится к вопросам взаимодействия программ на языке Си с операционной системой UNIX. Поскольку большинство пользователей языка работают в системе UNIX, то для многих читателей такое знакомство окажется полезным. Если даже вы работаете с Си на другой машине, то от знакомства с приведенным материалом вы только выиграете.

Глава освещает три основные темы: ввод-вывод, систему файлов и выделение памяти. Первые две предполагают хорошее знакомство с внутренними характеристиками системы UNIX.

В гл.8 мы уже касались вопросов взаимодействия с системой общих для целого ряда операционных систем. Программы стандартной библиотеки в любой конкретной системе пишутся с учетом возможностей ввода-вывода данной системы. В последующих разделах мы опишем основную систему входов для работы ввода-вывода в операционной системе UNIX и покажем, как с помощью этих входов можно реализовать стандартную библиотеку.

9.1. ДЕСКРИПТОРЫ ФАЙЛОВ

В операционной системе UNIX все работы по вводу и выводу трактуются как чтение и запись в файлы, поскольку все внешние устройства, даже пользовательский терминал, — это файлы в системе файлов. Это означает, что взаимодействие между программой и любым внешним устройством идет по единым, самым общим правилам.

В наиболее общем случае, прежде чем читать файл или записывать в него, необходимо информировать систему о том, что вы намерены делать. Этот процесс называется «открытием» файла. Если вы хотите записывать что-то в файл, то может оказаться, что его нужно создать. Система проверяет, имеете ли вы право это делать (существует ли файл? имеете ли вы разрешение работать с ним?) и, если все благополучно, выдает небольшое положительное целое число, называемое *дескриптором файла*. В любой работе по вводу-выводу этот дескриптор используется для идентификации файла вместо имени. (Можно провести грубую аналогию с READ(5,...) или WRITE(6,...) в Фортране.) Вся информация об открытом файле сохраняется в системе, пользователь ссылается на файл только с помощью дескриптора.

Так как ввод и вывод через пользовательский терминал — наиболее распространенный случай, то для такого варианта предусмотрены специальные соглашения. Когда интерпретатор команд («shell») запускает программу, он открывает три файла с дескрипторами файлов 0, 1 и 2 для стандартного ввода, стандартного вывода и стандартного вывода ошибок. Обычно все эти файлы связаны с терминалом, и поэтому, если прог-

рамма читает через дескриптор файла 0, а пишет через 1 и 2, она может работать, не заботясь об открытии этих файлов.

Пользователь программы может *перестроить* работу с файлами с помощью < и >:

```
prog <infile >outfile
```

В этом случае интерпретатор переводит неявное присваивание дескрипторам файлов 0 и 1 с терминала на поименованные файлы. Обычно дескриптор файла 2 остается прикрепленным к терминалу, так что сообщения об ошибках идут туда. Аналогичные же действия проводятся, если вход или выход связывается с каналом. Необходимо заметить, что во всех этих случаях все присваивания файлов выполняются самим интерпретатором, а не программой. Если программа использует файл 0 для ввода, а 1 и 2 для вывода, то она не знает, откуда и куда идет информация.

9.2. НИЖНИЙ УРОВЕНЬ ВВОДА-ВЫВОДА — READ и WRITE

На самом нижнем уровне ввода-вывода в UNIX не предусматривается ни буферизации, ни какого-либо другого обслуживания; фактически это прямые входы в операционную систему. Весь ввод и вывод идут через две функции: read и write. И у той и у другой первый параметр — дескриптор файла. Второй аргумент — это буфер в вашей программе, куда поступают либо откуда уходят данные. Третий параметр — число передаваемых байтов. Обращение идет так:

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

После обращения выдается счетчик байтов; он определяет, сколько фактически байтов было передано. При чтении число переданных байтов может быть меньше числа запрашиваемых. Если возвращаемое значение равно нулю, это предполагает конец файла, значение —1 указывает на некоторую ошибку. При записи возвращаемое значение есть число фактически записанных байтов; если оно не совпадает с числом, указанным в обращении, то это сигнализирует об ошибке.

Число считываемых или записываемых байтов довольно произвольно. Наиболее распространенные значения: 1, если речь идет об одном символе за один раз (небуферизованный обмен), и 512, что соответствует размеру физического блока для многих внешних устройств. Этот размер наиболее эффективен, однако даже обращение «символ за раз» не так уж неэффективно.

Собрав все эти факты вместе, мы можем теперь написать простую программу копирования «входа на выход», эквивалентную программе копирования файлов, приведенной в гл. 2. В системе UNIX эта программа копирует все, что угодно, и куда угодно, поскольку входы и выходы можно присоединить к любому файлу или устройству:

```
#define   BUFSIZE   512   /* наилучший размер для PDP-11 UNIX */

main()    /* копирование входа на выход */
{
     char buf[BUFSIZE];
     int  n;

     while ((n = read(0, buf, BUFSIZE)) > 0)
          write(1, buf, n);
}
```

Если размер файла не кратен BUFSIZE, то некоторые обращения к функции read будут считывать несколько меньшее число байтов, которые нужно записать, следующее же за этим обращение к read будет выдавать нуль.

Полезно посмотреть, как read и write можно использовать для конструирования программ более высоких уровней, вроде getchar, putchar и т. п. Вот, например, вариант getchar без буферизованного ввода:

```
#define   CMASK     0377 /*для отбрасывания знака у символов > 0 */

getchar() /* небуферизованный ввод одного символа */
{
     char c;

     return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

Переменную c *необходимо* описать как char, так как read работает со ссылкой на символ. Выдаваемый символ должен быть «маскирован» числом 0377, что гарантирует его положительность, иначе размножение знака может сделать его отрицательным. (Константа 0377 соответствует машине PDP-11, на других машинах она может быть другой.)

Второй вариант getchar вводит информации большими «кусками», а выдает по одному символу за обращение.

```
#define   CMASK     0377 /* для отбрасывания знака у символов > 0 */
#define   BUFSIZE   512

getchar() /* вариант с буфером */
{
     static char    buf[BUFSIZE];
     static char    *bufp = buf;
     static int     n = 0;

     if (n == 0) {  /* буфер пуст */
          n = read(0, buf, BUFSIZE);
          bufp = buf;
     }
     return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

9.3. OPEN, GREAT, CLOSE, UNLINK

Файлы, отличные от стандартных файлов ввода, вывода и ошибок, мы должны, чтобы их можно было читать или писать, явно открывать. Для этого в системе существуют две возможности: open и creat.

Функция open очень похожа на fopen, о которой говорилось в гл. 8. Однако вместо ссылки на файл она выдает дескриптор файла, который есть просто значение типа int:

```
int fd;

fd = open(name, rwmode);
```

Как и раньше, параметр name — это символьная строка, соответствующая внешнему имени файла. Параметр, определяющий вид доступа, rwmode, однако, другой: rwmode равен 0 для чтения, 1 — для записи и 2 — для чтения и записи. Функция open выдает —1, если произошла какая-то ошибка, иначе же выдается дескриптор нужного файла.

Попытка открыть несуществующий файл является ошибкой. Для порождения нового или перезаписи старого файла есть вход creat. Обращение

```
fd = creat(name, pmode);
```

возвращает дескриптор файла, если есть возможность создать файл с таким именем, иначе выдается —1. Если такой файл уже существует, то функция делает его длину нулевой (т.е. вся информация из него «выбрасывается»). Создание уже существующего файла не считается ошибкой.

Если речь идет о совсем новом файле, то при создании с ним связывается вид защиты, задаваемый параметром pmode. В файловой системе UNIX предусмотрено девять разрядов для информации о защите файла, управляющих чтением, записью и разрешающих исполнение для собственника файла, для группы собственников и для всех прочих. Таким образом, для задания статуса доступа больше всего подходит восьмеричное число из трех цифр. Например, 0755 разрешает чтение, запись и исполнение для собственника и чтение и исполнение для группы и для всех прочих.

В качестве примера приведем упрощенную версию обслуживающей программы *cp* из UNIX, копирующей один файл в другой. (Основное упрощение заключается в том, что наша версия копирует только один файл, и второй аргумент не может быть каталогом.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW для владельца, R для групп и пр. */

main(argc, argv)     /* cp: копировать f1 в f2 */
int argc;
char *argv[];
{
     int  f1, f2, n;
     char buf[BUFSIZE];

     if (argc != 3)
          error("Usage: cp from to", NULL);
     if ((f1 = open(argv[1], 0)) == -1)
          error("cp: can't open %s", argv[1]);
     if ((f2 = creat(argv[2], PMODE)) == -1)
          error("cp: can't create %s", argv[2]);

     while ((n = read(f1, buf, BUFSIZE)) > 0)
          if (write(f2, buf, n) != n)
               error("cp: write error", NULL);
     exit(0);
}

error(s1, s2)   /* печать сообщения об ошибке */
char *s1, *s2;
{
     printf(s1, s2);
     printf("\n");
     exit(1);
}
```

Существует ограничение на число файлов, которые может одновременно открыть одна программа (обычно 15—25). Поэтому любая программа, которая «намерена» работать со многими файлами, должна быть готова к повторному использованию дескрипторов файлов. Программа close разрывает связь между дескриптором файла и открытым файлом, освобождая тем самым дескриптор для работы с каким-либо другим файлом. Окончание программы с помощью оператора exit или возврата из главной программы закрывает все открытые файлы.

Функция unlink(filename) исключает файл с указанным именем из файловой системы.

Упражнение 9.1. Перепишите программу cat из гл. 8, используя read, write, open и close вместо их эквивалентов из стандартной библиотеки. Проведите какие-либо эксперименты, чтобы определить относительные скорости этих двух версий.

9.4. СЛУЧАЙНЫЙ ДОСТУП. SEEK и LSEEK

Обычно файлы ввода-вывода — это последовательные файлы: любое чтение или запись происходит в файле справа от предыдущего положения. Однако, если необходимо, файл можно читать и писать в произвольном порядке. В системе обращение к функции lseek дает возможность двигаться по файлу без фактического чтения или записи. Обращение

```
lseek(fd, offset, origin);
```

заставляет измениться текущее положение файла с дескриптором fd. Новое положение задается смещением offset относительно места, указанного параметром origin. Последующие запись и чтение будут начинаться с этого нового места. Параметр offset типа long, а fd и origin — целые. Origin может принимать значения 0, 1, и 2, означающие, что смещение отсчитывается соответственно от начала, от текущей позиции или от конца файла. Для добавления информации в файл нужно, прежде чем писать ее, найти конец файла:

```
lseek(fd, 0L, 2);
```

Чтобы вернуться к началу («перемотка»):

```
lseek(fd, 0L, 0);
```

Обратите внимание на параметр 0L; его можно записать и как (long)0.

С помощью функции lseek ценою замедления доступа файлы можно в большей или меньшей степени трактовать как большие массивы. Например, следующая простая функция читает произвольное число байтов из произвольного места файла:

```
get(fd, pos, buf, n) /* чтение n байтов, начиная с pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0);  /* переход к pos */
    return(read(fd, buf, n));
}
```

В версии, предшествующей 7-й версии UNIX, основной вход в систему ввода-вывода имел название seek. Эта функция была идентична lseek, но параметр offset был типа int, а не long. Поэтому, так как целые числа на PDP-11 занимают 16 разрядов, то offset для seek было ограничено числом 65535. В связи с этим значения параметра origin, равные 3, 4, 5, означали, что заданное смещение нужно умножать на 512 (число байтов в физическом блоке) и только после этого интерпретировать их соответственно как 0, 1, 2. Таким образом, для выделения нужного места в большом файле требовалось два поиска: первый выделял блок, а второй, с origin, равным 1, «подводил» к нужному байту в этом блоке.

Упражнение 9.2. Ясно, что seek можно выразить в терминах lssek и наоборот. Напишите каждую из функций в терминах другой.

9.5. ПРИМЕР. РЕАЛИЗАЦИЯ FOREN и GETS

Чтобы продемонстрировать, насколько эти функции согласованы одна с другой, покажем, как в стандартной библиотеке реализованы программы fopen и getc.

Напомним, что файлы в стандартной библиотеке описываются с помощью ссылок на файл, а не дескрипторов файла. Ссылка на файл — это ссылка на некоторую запись, содержащую какую-то информацию о файле: ссылку на буфер, так что файл можно читать большими порциями; счетчик оставшихся в буфере символов; ссылку на текущее местоположение символа в буфере; некоторые признаки, описывающие вид работы (чтение/запись и т. д.), и, наконец, дескриптор файла.

Описание записи, характеризующей файл, находится в файле stdio.h. Этот файл должен быть включен (с помощью #include) в любой исходный файл, в котором используются программы из стандартной библиотеки. Он также включается и в функции библиотеки. В приводимом ниже фрагменте из stdio.h имена, предназначенные только для использования в библиотечных функциях, начинаются с подчеркивания, так что вероятность их совпадения с именами из пользовательских программ невелика:

```
#define   _BUFSIZE  512
#define   _NFILE    20    /*    число обрабатываемых файлов */

typedef struct _iobuf {
     char *_ptr;     /* место очередного символа */
     int  _cnt;      /* число оставшихся символов */
     char *_base;    /* местоположение буфера */
     int  _flag;     /* вид доступа к файлу */
     int  _fd;       /* дескриптор файла */
} FILE;
extern FILE _iob[_NFILE];

#define   stdin     (&_iob[0])
#define   stdout    (&_iob[1])
#define   stderr    (&_iob[2])

#define   _READ     01    /* файл открыт для чтения */
#define   _WRITE    02    /* файл открыт для записи */
#define   _UNBUF    04    /* файл без буфера */
#define   _BIGBUF   010   /* есть большой буфер */
#define   _EOF 020  /* в этом файле был EOF */
```

```
#define   _ERR 040   /* в этом файле встречались ошибки */
#define   NULL 0
#define   EOF  (-1)

#define   getc(p)    (--(p)->_cnt >= 0 \
                ? *(p)->_ptr++ & 0377 : _fillbuf(p))
#define   getchar() getc(stdin)

#define   putc(x,p) (--(p)->_cnt >= 0 \
                ? *(p)->_ptr++ = (x) : _flushbuf((x),p))
#define   putchar(x)      putc(x,stdout)
```

Макроопределение getc обычно только уменьшает счетчик, увеличивает ссылку и выдает символ. (Длинное #define продолжается с помощью символа \). Если счетчик становится отрицательным, то getc обращается для заполнения буфера к функции _fillbuf, восстанавливает начальное состояние записи и выдает символ. Хотя функция и может содержать непереносимые конструкции, правила взаимодействия с нею могут оставаться переносимыми. Например, getc маскирует символ константой 0377, это блокирует имеющееся на PDP-11 размножение знака и обеспечивает положительность всех символов.

Хотя и без обсуждения деталей реализации, но мы включили сюда и определение putc, чтобы показать, что она работает так же, как и get, обращаясь при полном буфере к функции _flushbuf.

Теперь можно написать функцию fopen. Большая часть программы связана с «получением» открываемого файла, позиционированием его в нужное состояние (установка положения) и установкой признаков, указывающих надлежащее состояние. Fopen не запрашивает никакой памяти для буфера, это делается при первом чтении файла программой —fillbuf:

```
#include  <stdio.h>
#define   PMODE      0644 /* R/W для владельца ; R  для других */

FILE *fopen(name, mode)   /* открыть файл, выдать ptr */
register char *name, *mode;
{
     register int fd;
     register FILE *fp;

     if (*mode != 'r' && *mode != 'w' && *mode != 'a') {
          fprintf(stderr, "illegal mode %s opening %s\n",
               mode, name);
          exit(1);
     }
     for (fp = _iob; fp < _iob + _NFILE; fp++)
          if ((fp->_flag & (_READ | _WRITE)) == 0)
               break;     /* найдено свободное место */
     if (fp >= _iob + _NFILE) /* места нет */
          return(NULL);

     if (*mode == 'w')    /* доступный файл  */
          fd = creat(name, PMODE);
     else if (*mode == 'a') {
          if ((fd = open(name, 1)) == -1)
               fd = creat(name, PMODE);
          lseek(fd, 0L, 2);
     } else
          fd = open(name, 0);
```

```
        if (fd == -1)   /* недоступное имя */
            return(NULL);

        fp->_fd = fd;
        fp->_cnt = 0;
        fp->_base = NULL;
        fp->_flag &= ~(_READ | _WRITE);
        fp->_flag |= (*mode == 'r') ? _READ : _WRITE;
        return(fp);
    }
```

Функция _fillbuf несколько более сложная. Основная сложность заключается в том, что программа стремится обеспечить доступ к файлу даже в том случае, когда не хватает места для буферов ввода-вывода. Если место для нового буфера можно получить от calloc, все хорошо, если же нет, то _fillbuf будет работать с небуферизованным вводом, используя посимвольный ввод и сохраняя символ в некотором собственном массиве:

```
#include  <stdio.h>

_fillbuf(fp)   /* выделение и заполнение входного буфера */
register FILE *fp;
{
    static char smallbuf[_NFILE]; /* для работы без буфера */
    char *calloc();

    if ((fp->_flag&_READ)==0 || (fp->_flag&(_EOF|_ERR))!=0)
        return(EOF);
    while (fp->_base == NULL)   /* поиск места для буфера */
        if (fp->_flag & _UNBUF)   /* без буфера */
            fp->_base = &smallbuf[fp->_fd];
        else if ((fp->_base=calloc(_BUFSIZE, 1)) == NULL)
            fp->_flag |= _UNBUF; /* большой буфер получить нельзя */
        else
            fp->_flag |= _BIGBUF; /* есть большой */
    fp->_ptr = fp->_base;
    fp->_cnt = read(fp->_fd, fp->_ptr,
                fp->_flag & _UNBUF ? 1 : _BUFSIZE);
    if (--fp->_cnt < 0) {
        if (fp->_cnt == -1)
            fp->_flag |= _EOF;
        else
            fp->_flag |= _ERR;
        fp->_cnt = 0;
        return(EOF);
    }
    return(*fp->_ptr++ & 0377);   /* символ положительный */
}
```

При первом обращении к getc для конкретного файла обнаруживается, что счетчик нулевой и происходит обращение к _fillbuf. Если _fillbuf находит, что файл не открыт для чтения, сразу же выдается EOF. В других случаях делается попытка захватить большой буфер; если она неудачна, то устанавливается буфер на один символ и делается соответствующая отметка в переменной _flag. Установив буфер, _fillbuf просто обращается к read и заполняет его, затем устанавливается счетчик и ссылка, и выдается символ из начала буфера. Последующие обращения к _fillbuf находят буфер уже выделенным.

Остается пока неясным, с чего все начинается, каковы начальные значения? Для файлов stdin, stdout и stderr массив _iob должен быть описан и инициирован так:

```
FILE _iob[_NFILE] ={
     { NULL, 0, NULL, _READ, 0 },  /* stdin */
     { NULL, 0, NULL, _WRITE, 1 }, /* stdout */
     { NULL, 0, NULL, _WRITE | _UNBUF, 2 }   /* stderr */
};
```

Инициация в описании переменной _flag показывает, что stdin находится в состоянии чтения, stdout — записи, а stderr — небуферизованной записи.

Упражнение 9.3. Перепишите fopen и _fillbuf. Организуйте работу с полями вместо работы с отдельными разрядами.

Упражнение 9.4. Спроектируйте и напишите программы _flushbuf и fclose.

Упражнение 9.5. В стандартной библиотеке есть функция с таким обращением к ней:

```
fseek(fp, offset, origin)
```

она идентична lseek, но fp — ссылка на файл, а не дескриптор файла. Напишите fseek. Обеспечьте ее скоординированную работу по буферизации с другими программами библиотеки.

9.6. ПРИМЕР. РАСПЕЧАТКА КАТАЛОГОВ.

Иногда приходится обращаться к файловой системе не за содержимым файлов, а за информацией *о* самих файлах. Примером служит команда *ls* («list directory») из UNIX: она печатает имена файлов в некотором каталоге и, возможно, другую информацию, такую, как размер, статус доступа и т. д.

Поскольку любой каталог — это просто файл (по крайней мере в UNIX это так), то в командах вроде *ls* нет ничего специфического. Однако формат информации определяется не пользователем, а системой, и поэтому *ls* должна знать, как и что в системе представляется.

Мы проиллюстрируем некоторые из этих представлений на примере программы под названием *fsize*. Это некоторая специальная разновидность *ls*, печатающая размеры всех файлов, поименованных в списке ее параметров. Если один из файлов представляет собой каталог, то fsize рекурсивно применяет саму себя к такому каталогу. Если параметры вообще не заданы, она обрабатывает текущий каталог.

Начнем с простого обзора структуры файловой системы. Любой каталог — это файл, содержащий имена файлов и некоторые указания о их местоположении. Местоположение — это фактически индекс в другой таблице — таблице характеристик. Характеристика файла — это место, где хранится вся информация о файле за исключением его имени. Любая строка каталога состоит только из двух элементов: номера характеристики и имени файла. Точные спецификации получаются путем включения в программу файла sys/dir.h, который содержит описание

```
#define   DIRSIZ    14   /* максимальная длина имени файла */

struct direct  /* структура строки каталога */
{
     ino_t d_ino;    /* номер inode */
     char d_name[DIRSIZ];      /* имя файла  */
};
```

«Тип» ino_t вводится с помощью описания typedef, и описывает индекс в таблице характеристик. В UNIX для PDP-11 он определен как unsigned, но этой особенностью в программе нигде не пользуются, и в других системах это определение может быть другим. Таким образом, лучше здесь пользоваться описанием со словом typedef. Полное перечисление «системных» типов находится в файле sys/types.h.

Функция stat берет имя файла и выдает всю информацию из характеристики этого файла. Если выдается —1, то где-то была допущена ошибка. Так, обращение

```
struct stat stbuf;
char *name;

stat(name, &stbuf);
```

заполняет запись stbuf информацией из характеристики указанного файла. Структура выдаваемого stat значения описана в sys/stat.h и выглядит примерно так:

```
struct stat     /* запись, выдаваемая  stat */
{
        dev_t       st_dev;     /* устройство */
        ino_t       st_ino;     /* номер inode */
        short       st_mode;    /* вид */
        short       st_nlink;   /* число ссылок на файл */
        short       st_uid;     /* идентификатор собст. */
        short       st_gid;     /* групповой идентификатор собст. */
        dev_t       st_rdev;    /* для особых файлов */
        off_t       st_size;    /* размер файла  в символах */
        time_t      st_atime;   /* время последнего доступа  */
        time_t      st_mtime;   /* время последней работы */
        time_t      st_ctime;   /* время создания  */
};
```

Смысл большинства значений ясен из примечаний. Элемент st_mode содержит признаки, описывающие файл; описания, определяющие признаки, также являются и частью файла sys/stat.h, что иногда бывает удобно.

```
#define S_IFMT 0160000          /* тип файла */
#define   S_IFDIR   0040000     /* каталог */
#define   S_IFCHR   0020000     /* особый символ */
#define   S_IFBLK   0060000     /* особый блок */
#define   S_IFREG   0100000     /* регулярный */
#define S_ISUID     04000   /* при выполнении задать идентификатор пользователя */
#define S_ISGID     02000   /* при выполнении задать идентификатор групп */
#define S_ISVTX     01000   /* после использования сохранить текст */
#define S_IREAD     0400    /* чтение разрешено */
#define S_IWRITE    0200    /* запись разрешена */
#define S_IEXEC     0100    /* выполнение разрешено */
```

Теперь мы можем написать саму программу *fsize*. Если полученный из stat вид указывает, что файл не есть каталог, то мы имеем уже его размер и можем его сразу же напечатать. Если же он каталог, то мы должны выбирать из него и обрабатывать по одному файлы; в свою очередь здесь опять может оказаться подкаталог, так что процесс рекурсивен.

Главная программа, как обычно, в основном имеет дело с параметрами из команды. Она помещает каждый параметр для fsize в большой буфер:

```
#include <stdio.h>
#include <sys/types.h>    /* определение типов */
#include <sys/dir.h>      /* структура строки каталога */
#include <sys/stat.h>     /* запись, выдаваемая stat */
#define   BUFSIZE   256

main(argc, argv)     /* печать размеров файлов */
char *argv[];

    char buf[BUFSIZE];

    if (argc == 1) {     /* подразумевается текущий каталог */
        strcpy(buf, ".");
        fsize(buf);
    } else
        while (--argc > 0) {
            strcpy(buf, *++argv);
            fsize(buf);
        }
}
```

Функция fsize печатает размер файла. Однако, если файл есть некоторый каталог, то для обработки в нем всех файлов fsize вначале обращается к directory. Обратите внимание на использование признаков с именами S_IFMT и S_IFDIR из stat.h:

```
fsize(name)     /* печать размера файла */
char *name;
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't find %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        directory(name);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

Функция directory самая сложная. Однако основная ее работа связана с порождением полного путевого имени обрабатываемого в данный момент файла:

```
directory(name)      /* для всех файлов в name */
char *name;
{
    struct direct dirbuf;
    char *nbp, *nep;
    int i, fd;

    nbp = name + strlen(name);
    *nbp++ = '/';   /* добавить "слеш" к имени каталога */
    if (nbp+DIRSIZ+2 >= name+BUFSIZE)   /* имя слишком большое */
        return;
```

```
        if ((fd = open(name, 0)) == -1)
                return;
        while (read(fd, (char *)&dirbuf, sizeof(dirbuf))>0) {
                if (dirbuf.d_ino == 0)    /* место не используется */
                        continue;
                if (strcmp(dirbuf.d_name, ".") == 0
                  || strcmp(dirbuf.d_name, "..") == 0)
                        continue; /* пропустить себя и родителей */
                for (i=0, nep=nbp; i < DIRSIZ; i++)
                        *nep++ = dirbuf.d_name[i];
                *nep++ = '\0';
                fsize(name);
        }
        close(fd);
        *--nbp = '\0'; /* восстановить имя */
}
```

Если в данный момент некоторая строка каталога не используется (из-за того, что файл уже убран), то строка характеристики нулевая и эта позиция пропускается. В каждом каталоге есть строки для него самого (с именем «.») и для «родителя» — («..»); ясно, что они также должны пропускаться, иначе программа обработает все каталоги целиком.

Хотя программа *fsize* и несколько специфична, но в ней можно обнаружить пару хороших идей. Первая — многие из программ не являются «системными программами»: они просто используют ту информацию, которую формирует или поддерживает операционная система. Вторая — важно для таких программ то, что представление информации задается в стандартных файлах-«заголовках» вроде stat.h и программы просто включают эти файлы, а не «встраивают в себя» фактические описания.

9.7. ПРИМЕР. РАСПРЕДЕЛИТЕЛЬ ПАМЯТИ

В гл. 6 мы привели некоторую наивную версию программы alloc. В тот вариант, который мы составим теперь, ограничения уже вноситься не будут: обращения к alloc и к free могут следовать в любом порядке; для получения по мере необходимости дополнительной памяти alloc вызывает операционную систему. Кроме того, что программы полезны сами по себе, они еще иллюстрируют некоторые полезные соображения, связанные с написанием машинно-зависимых программ в относительно машинно-независимой форме, и демонстрируют как записи, смеси и определяемые типы используются в «реальной жизни».

Программа alloc будет по мере необходимости требовать память от операционной системы, а не выделять ее из «транслированного» массива фиксированного размера. Посколку разные процессы в программе могут требовать память и асинхронно, то выделяемая память не обязательно будет «непрерывной». Таким образом, свободная память хранится как список свободных блоков. Каждый блок содержит его размер, ссылку на следующий блок и самою память. Блоки хранятся в порядке возрастания адресов памяти, и последний блок, с самым большим адресом, содержит ссылку на первый. Фактически у нас не просто список, а кольцевой список.

При запросе список свободных адресов просматривается до тех пор, пока не будет найден достаточно большой блок. Если блок точно требуемого размера, то он выделяется из списка и передается пользователю. Если блок слишком большой, то он расщепляется и часть требуемого размера отдается пользователю, а остаток поступает вновь в список свободных. Если достаточно большого блока нет, то от операционной системы получается еще блок, он включается в список свободных и поиск возобновляется.

Освобождение также связано с поиском в списке свободных: чтобы найти место для освобождаемого блока. Если освобождаемый блок примыкает с той или другой стороны к уже свободному блоку, то они сливаются в один бóльший блок; тем самым не возникает слишком большая фрагментарность. Соседство определяется легко, так как список свободных поддерживается в порядке расположения в памяти.

Одна из проблем (о ней мы уже упоминали в гл. 5) заключается в том, что нужно позаботиться, чтобы память, поставляемая программой alloc, была бы надлежащим образом выравнена для объектов, которые в ней будут храниться. Хотя машины бывают и разными, но в каждой есть наиболее «критичный» тип. Если такой критичный тип допускает некоторую адресацию, то допускают ее и другие. Например, в IBM 360/370, в Honeywell 6000 и во многих других машинах любой объект можно хранить в том месте, которое подходит для двойной точности. В машинах PDP-11 для этого достаточно сослаться на тип int.

Любой свободный блок содержит ссылку на следующий блок в списке, информацию о размере блока и затем само свободное пространство; управляющая информация в начале блока называется «заголовком». Для упрощения выравнивания все блоки кратны размеру заголовка и заголовки выравнены надлежащим образом. Этого можно добиться с помощью смеси, где есть заголовок необходимой структуры и наиболее критичный к выравниванию тип:

```
typedef int ALIGN;   /* форсирование выравнивания на PDP-11 */

union header {  /* заголовок свободного блока */
     struct {
          union header *ptr;   /*очередной свободный блок*/
          unsigned size; /* размер этого свободного блока */
     } s;
     ALIGN      x;    /* форсирование выравнивания */
};   typedef union header HEADER;
```

В программе alloc требуемый размер (в символах) округляется до нужного (в единицах заголовка — квантах) размера. Фактически выделенный блок будет состоять из более чем одного такого кванта, и это значение записывается в элемент size из заголовка. Ссылка, которую выдает alloc, указывает на свободное пространство, а не на сам заголовок:

```
static HEADER base;  /* пустой список для начала процесса */
static HEADER *allocp = NULL;  /*  последний выделенный блок  */

char *alloc(nbytes)  /* универсальный распределитель памяти */
unsigned nbytes;
{
     HEADER *morecore();
     register HEADER *p, *q;
     register int nunits;

     nunits = 1+(nbytes+sizeof(HEADER)-1)/sizeof(HEADER);
     if ((q = allocp) == NULL) {     /* списка свободных нет */
          base.s.ptr = allocp = q = &base;
          base.s.size = 0;
     }
     for (p=q->s.ptr; ; q=p, p=p->s.ptr) {
          if (p->s.size >= nunits) {     /* достаточно большой */
               if (p->s.size == nunits)  /* точно */
                    q->s.ptr = p->s.ptr;
```

```
            else {      /* выделение остатка */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            allocp = q;
            return((char *)(p+1));
        }
        if (p == allocp)   /* замыкание списка свободных */
            if ((p = morecore(nunits)) == NULL)
                return(NULL);   /* не осталось ни одного */
    }
}
```

Переменная base используется для начала процесса. Если allocp равно NULL, как это бывает при первом обращении в alloc, то создается вырожденный список свободных блоков: он содержит один блок нулевого размера и ссылку на самого себя. Затем в любом случае просматривается список свободных. Поиск свободного блока подходящего размера начинается с точки (allocp), где был найден последний блок; такая стратегия позволяет сохранять список однородным. Если найден слишком большой блок, пользователю выдается «хвостовая» часть. При таком методе в первоначальном заголовке требуется изменить лишь его размер. В любом случае ссылка, выдаваемая пользователю, указывает на фактически свободную область, смещенную относительно заголовка на один квант. Обратите внимание, что p, прежде чем его выдавать пользователю, преобразуется в ссылку на символ.

Функция morecore получает память от операционной системы. Детали этого процесса варьируются от системы к системе. В UNIX системное обращение sbrk(n) выдает ссылку на дополнительные n байтов памяти. (Ссылка удовлетворяет всем требованиям выравнивания.) Так как запрос у системы памяти — операция сравнительно дорогая, мы не хотели бы поступать подобным образом при каждом обращении к alloc, поэтому morecore округляет число затребованных квантов до некоторой бо́льшей величины, и полученный большой блок будет затем делиться по мере необходимости. Величина превышения — это параметр системы. Его, если это нужно, можно и изменить.

```
#define   NALLOC    128   /* # число требуемых квантов */

static HEADER *morecore(nu)    /* запрос памяти у системы */
unsigned nu;
{
    char *sbrk();
    register char *cp;
    register HEADER *up;
    register int rnu;

    rnu = NALLOC * ((nu+NALLOC-1) / NALLOC);
    cp = sbrk(rnu * sizeof(HEADER));
    if ((int)cp == -1)   /* памяти нет*/
        return(NULL);
    up = (HEADER *)cp;
    up->s.size = rnu;
    free((char *)(up+1));
    return(allocp);
}
```

Обращение к sbrk, если места нет, будет выдавать —1, хотя и кажется, что лучше было бы получать NULL. Чтобы не заботиться о сравнении, -1 нужно преобразовать к типу int. Здесь часто используется преобразование, и поэтому функция относительно «невосприимчива» к деталям представления ссылок на различных машинах.

И наконец, функция free. Это простой просмотр списка свободных блоков, начинающегося с allocp, в поисках, куда бы включить освободившейся блок. Либо между двумя какими-то блоками, либо в один из концов. Во всяком случае, если освобождающийся блок примыкает к любому из соседей, то смежные блоки объединяются. Трудности заключаются только в том, чтобы размеры были верные, а ссылки указывали туда, куда надо:

```
free(ap)   /* занести блок ap в список свободных */
char *ap;
{
     register HEADER *p, *q;

     p = (HEADER *)ap - 1;      /* ссылка на заголовок */
     for (q=allocp; !(p > q && p < q->s.ptr); q=q->s.ptr)
          if (q >= q->s.ptr && (p > q || p < q->s.ptr))
               break;      /* в один или другой конец */

     if (p+p->s.size == q->s.ptr) {  /* добавить к верхнему */
          p->s.size += q->s.ptr->s.size;
          p->s.ptr = q->s.ptr->s.ptr;
     } else
          p->s.ptr = q->s.ptr;
     if (q+q->s.size == p) {   /* добавить к нижнему */
          q->s.size += p->s.size;
          q->s.ptr = p->s.ptr;
     } else
          q->s.ptr = p;
     allocp = q;
}
```

Хотя процесс выделения памяти существенно машинно-зависим, но приведенная выше программа служит иллюстрацией того, что этой зависимостью от машины можно управлять и сосредоточивать ее в очень небольшой части программы. Употребление описания типа и смесей позволяет решить проблему выравнивания (при условии, что sbrk выдает соответствующую ссылку). Явно выполняемые преобразования сглаживают затруднения с преобразованиями ссылок и позволяют работать и при плохо спроектированном взаимодействии с системой. Даже если детали нашей программы и относятся только к распределению памяти, общий подход с успехом применим и к решению других проблем.

Упражнение 9.6. Стандартная библиотечная функция calloc(n, size) выдает ссылку на n объектов указанного размера size, причем память инициирована нулевыми значениями. Напишите эту программу, используя alloc в качестве модели или же обращаясь к ней.

Упражнение 9.7. Функция alloc воспринимает требуемый размер, не контролируя его правдоподобность. Free исходит из того, что блок, который ее просят освободить, содержит поле нужного размера. Усовершенствуйте эти программы, обратив больше внимания на контроль ошибок.

Упражнение 9.8. Напишите программу bfree(p, n), освобождающую произвольный блок p из n символов и работающую со списком свободных блоков, поддерживаемым программами alloc и free. Используя bfree, пользователь может в любой момент добавлять в список свободных блоков любые статические и внешние массивы.

Приложение

СПРАВОЧНОЕ РУКОВОДСТВО ПО ЯЗЫКУ СИ

1. ВВЕДЕНИЕ

В этом руководстве мы описываем язык Си для машин DEC PDP-11, Honeywell 6000, системы IBM/370 и Interdata 8/32. Если встречаются различия, предпочтение отдается PDP-11, хотя мы стараемся выделять детали, зависящие от реализации. За немногими исключениями различия объясняются специфическими особенностями аппаратуры, сами же трансляторы в общем достаточно совместимы между собой.

2. СОГЛАШЕНИЯ О ЛЕКСИЧЕСКИХ ПОНЯТИЯХ

Существует шесть классов лексем: идентификаторы, служебные слова, константы, строки, операции и другие разделители. Пробелы, символы табуляции, переходы на новую строчку и примечания (все вместе будем называть «пустыми символами»), как это будет описано ниже, игнорируются, но могут использоваться для отделения одной лексемы от другой.

Если входной текст уже разбит до некоторого символа на лексемы, следующей лексемой будет максимально длинная последовательность символов, могущих вообще быть лексемой.

2.1. Примечания

Пара символов /* начинает примечание, а заканчивается оно парой */. Примечания не могут вкладываться одно в другое.

2.2. Идентификаторы (имена)

Любой идентификатор — это последовательность букв и цифр с первым символом обязательной буквой. Подчеркивание "_" считается буквой. Строчные и прописные буквы—различные буквы. В идентификаторе имеют значение лишь первые восемь символов, хотя их может быть и больше. На внешние идентификаторы, используемые во всякого рода ассемблерах и загрузчиках, накладываются еще большие ограничения:

DEC PDP-11	7 символов на двух регистрах;
Honeywell 6000	6 символов на одном регистре;
IBM 360/370	7 символов на одном регистре;

2.3. Служебные слова

Перечисленные ниже идентификаторы фиксируются как служебные слова и в другом смысле использоваться не могут:

```
int         extern      else
char        register    for
float       typedef     do
double      static      while
struct      goto        switch
union       return      case
long        sizeof      default
short       break       entry
unsigned    continue
auto        if
```

Слово entry в каком-либо трансляторе сейчас не реализовано, и мы его резервируем для дальнейшего использования. В некоторых реализациях кроме перечисленных слов зарезервированы слова fortran и asm*.

2.4. Константы

Существуют константы нескольких видов, они перечисляются ниже. В разд. 2.6 приведены характеристики констант, зависящие от машины.

2.4.1. Целые константы

Любая целая константа, представляющая собою последовательность цифр, считается восьмеричной, если она начинается с 0 (цифры нуль), иначе это десятичная константа. Цифры 8 и 9 как восьмеричные представляют собою соответственно восьмеричные значения 10 и 11. Если последовательность начинается с пары 0x или 0X (цифра нуль), то константа считается шестнадцатеричной. Шестнадцатеричные цифры включают буквы от a или A до f или F, представляющие значения от 10 до 15. Любая десятичная константа, значение которой выходит за пределы самого большого в машине целого со знаком, считается длинной; аналогично, если восьмеричная или шестнадцатеричная константа превышает максимальное для машины целое без знака, она считается длинной.

2.4.2. Явная длинная константа

Если десятичная, восьмеричная или шестнадцатеричная константа заканчивается буквой l или L, то это длинная константа. Как это можно увидеть ниже, в некоторых машинах целые и длинные целые значения могут оказаться идентичными.

2.4.3. Символьные константы

Символьная константа — это некоторый символ, заключенный в одиночные кавычки, например 'x'. Значение символьной константы — числовое значение символа в некотором зависящем от машины множестве символов.

Символы, не имеющие графического представления, символ ' (одиночная кавычка) и \ (обратная дробная черта) можно представлять комбинацией символов (образов) в соответствии с приводимой ниже таблицей:

---

* Мы оставляем это слово без перевода, так как смысл его не совсем ясен и в дальнейшем мы его не встречали. Скорее всего это assembler — ассемблер. — *Примеч. пер.*

новая строчка	\n
горизонтальная табуляция	\t
шаг назад	\b
возврат каретки	\r
авторегистр	\f
обратная дробная черта	\\
одиночная кавычка	\'
последовательность двоичных разрядов	*ddd*

Комбинация *ddd* состоит из символа \, за которым следует 1, 2 или 3 восьмеричные цифры, задающие значение желаемого символа. Специальный случай — \0 (без последующих цифр) представляет символ NUL. Если за обратной дробной чертой идет символ, не входящий в число указанных, то обратная дробная черта просто игнорируется.

2.4.4. Константы с плавающей точкой

Константа с плавающей точкой содержит целую часть, десятичную точку, дробную часть, символы e или E и целый показатель (экспоненту), возможно, со знаком. И целая, и дробная части — последовательности цифр. Любая из этих частей может отсутствовать, но не обе сразу. Аналогично и десятичная точка или e с порядком могут быть опущены (но не то и другое). Любая константа с плавающей точкой считается значением с двойной точностью.

2.5. Строки

Строка—последовательность символов, заключенная в двойные кавычки, т. е. "...". Строки считаются значениями типа «массив символов», относящимися к статической памяти (см. ниже разд. 4) и инициированными заданными символами. Любые строки, даже если они идентичны по написанию, считаются различными объектами. Транслятор помещает в конец каждой строки нулевой байт \0, так что программа, просматривающая строку, может найти ее конец. Символу двойной кавычки в строке должен предшествовать символ \, и, кроме того, можно использовать любые другие указанные «образы» символов. Символ \, за которым идет сразу символ новой строчки, игнорируется.

2.6. Зависимость от машины

В приведенной таблице суммируются особенности представления значений, меняющиеся от машины к машине. Хотя это и влияет на перенос программ, но на практике при этом возникает меньше проблем, чем можно было ожидать априори:

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
`char`	8 разрядов	9 разрядов	8 разрядов	8 разрядов
`int`	16	36	32	32
`short`	16	36	16	16
`long`	32	36	32	32
`float`	32	36	32	32
`double`	64	72	64	64
диапазон	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

На всех четырех типах машин порядок чисел с плавающей точкой занимает 8 разрядов.

3. НОТАЦИЯ ДЛЯ СИНТАКСИСА

В этой книге мы используем для описания синтаксиса такую нотацию, при которой синтаксические понятия выделяются курсивом, а терминальные слова и символы — другим шрифтом. Возможные альтернативы синтаксических понятий перечисляются в отдельных строчках текста. Возможное вхождение понятия или терминального символа отмечается индексом «opt». Например, запись

$$\{\ \textit{выражение}_{opt}\ \}$$

означает, что в скобках, возможно, стоит выражение. Полностью синтаксис языка приводится в разд. 18.

4. ЧТО ТАКОЕ ИМЯ?

Наш язык основан на интерпретации двух атрибутов идентификаторов: его *класса памяти* и его *типа*. Класс памяти определяет местонахождение (ячейку) и время существования памяти, сопоставленной с некоторым идентификатором, а тип определяет смысл значений, хранящихся в этой памяти.

Есть четыре описываемых класса памяти: автоматическая, статическая, внешняя и память на регистрах. Автоматические переменные локальны по отношению к каждому обращению к некоторому блоку (разд. 9.2), при выходе из этого блока они уничтожаются. Статические переменные также локальны по отношению к блоку, но их значения сохраняются даже при выходе из блока и могут использоваться при повторном входе в этот блок. Внешние переменные существуют и сохраняют свои значения на протяжении выполнения всей программы; они могут использоваться для связи между функциями и даже между отдельно оттранслированными функциями. Регистровые переменные хранятся в быстрых регистрах машины, если это возможно; подобно автоматическим переменным они локальны в блоке и уничтожаются при выходе из блока.

В языке Си предусмотрено несколько основных типов объектов. Объекты, описанные как символы (char), достаточно велики, чтобы хранить любой из элементов зависящего от реализации множества символов. Если некоторый оригинальный символ из этого множества символов хранится в символьной переменной, то ее значение эквивалентно целому числу, представляющему данный символ. В символьных переменных могут храниться и другие величины, но эта возможность связана с особенностями машины и реализации.

Можно пользоваться целыми числами трех размеров: они описываются как short int, int или long int. Для длинных целых предусматривается память не меньше, чем для коротких, но при реализации и длинные, и короткие целые могут быть сделаны эквивалентными обычным целым. «Обычные» целые соответствуют «естественному» размеру, предусмотренному в машине, на язык которой идет трансляция, а другие размеры используются лишь для особых нужд.

Целые без знака, описываемые как unsigned, подчиняются правилам арифметики по mod 2^n, где n — число разрядов в представлении. (На машине PDP-11 работа с длинными значениями без знака не предусматривается.)

В некоторых реализациях значения с плавающей точкой единичного размера (float) могут совпадать со значениями с плавающей точкой двойного размера.

Поскольку объекты вышеуказанных типов бывает полезно интерпретировать как числа, мы будем в этом случае говорить об *арифметических* типах. Типы char и int вместе будем называть *целыми* типами, а типы float и double — *плавающими* типами.

Кроме основных арифметических типов, существует концептуально бесконечное множество производных типов, строящихся из основных. Это:

массивы объектов, относящихся к большинству из типов;

функции, возвращающие объект заданного типа;

ссылки на объекты заданного типа;

записи, состоящие из последовательностей объектов различных типов;

смеси, способные содержать один из нескольких объектов различных типов.

Как правило, эти методы построения составных объектов могут применяться рекурсивно.

5. ОБЪЕКТЫ И АДРЕСА

Любой объект—это некоторая область памяти, с которой можно работать; а *адрес*—некоторое выражение, ссылающееся на такой объект. Очевидным примером адреса будет идентификатор. Существуют операции, порождающие адреса: например, если E выражение ссылочного типа, то * E — адресное выражение, соответствующее объекту, на который ссылалось E. Понятие «адрес» появляется из оператора присваивания E1=E2, где левый операнд E1 должен «адресовать» изменяемую переменную. Описывая операции, мы всегда будем указывать, применимы ли они к адресным операндам и порождают ли они сами адреса.

6. ПРЕОБРАЗОВАНИЯ

Некоторые операции в зависимости от их операндов могут вызывать преобразование значений операндов из одного типа в другой. Ниже мы объясняем, к чему приводят такие пребразования, а в разд. 6.6 приводим общие правила преобразований, применимые для большинства обычных операций. Если нужно, они будут уточняться при рассмотрении каждой операции в отдельности.

6.1. Символы и целые числа

Везде, где можно использовать целое число, можно использовать и короткие целые, и символы. В любом случае соответствующее значение преобразуется к целому. Преобразование коротких целых к длинным всегда включает «размножение» знака, ведь целые — это величины со знаком. Будет происходить размножение знака для символов или нет, зависит от машины, однако гарантируется, что элементы стандартного множества символов не отрицательны. Из машин, о которых идет речь в этой книге, знак размножается лишь в PDP-11. Здесь значения символьных переменных лежат в диапазоне от — 128 до 127, но все символы алфавита ASCII положительные. У символьной константы, заданной как восьмеричный образ, может произойти размножение знака, например '\377' имеет значение — 1.

При преобразовании длинного целого в короткое или в символ его левая часть «отсекается»: лишние разряды просто уничтожаются.

6.2. Значения с плавающей точкой и двойная точность

В языке Си считается, что все арифметические операции выполняются с двойной точностью. Поэтому всякий раз, когда в выражении появляется значение с плавающей точкой, его «удлиняют» до удвоенной точности, добавляя к дробной части «нули». При преобразовании значения с двойной точностью к одинарной, например при присваивании, прежде чем отбросить лишние для обычной точности разряды, происходит округление.

6.3. Плавающие типы и целые

Преобразование значений с плавающей точкой в целом зависит до некоторой степени и от машины, в частности в различных машинах по-разному происходит преобразование отрицательных чисел. Результат преобразования будет неопределен, если значение не «помещается» в отведенную ему память.

Преобразование целых значений в значения с плавающей точкой не вызывает осложнений, но, если для результата не обеспечено соответствующее число разрядов, будет происходить некоторая потеря точности.

6.4. Ссылки и целые

Целое или длинное целое значение можно добавлять к ссылке или вычитать из нее. В этом случае первый операнд будет преобразован в соответствии с правилами, приведенными при обсуждении операции сложения.

Если есть две ссылки к объектам одного типа, можно вычитать одну из другой: результат будет преобразован в целое значение по правилам, указанным при обсуждении операции вычитания.

6.5. Целые без знака

Если комбинируется целое без знака и простое целое, то целое преобразуется в целое без знака и результат будет целым без знака. Его значение есть наименьшее целое без знака, совпадающее с целым со знаком по (mod $2^{\text{размер слова}}$). Если мы имеем дело с представлением в дополнительном коде, то это преобразование лишь концептуальное, на самом деле с поразрядным представлением ничего не происходит.

Если целое без знака преобразуется в длинное целое, то значение результата численно то же самое, что и целое без знака, т. е. преобразование размера приводит к добавлению «слева» нулей.

6.6. Арифметические преобразования

Для подавляющего большинства операций преобразования операндов и тип результата определяются одними и теми же правилами. Эти правила можно назвать «обычными арифметическими преобразованиями».

Первое. Любой операнд типа char или short преобразуется в целое, а любой операнд плавающего типа преобразуется к двойной точности.

Второе. Если один из операндов двойной точности, другой преобразуется к двойной точности, и получается результат того же типа.

Иначе. Если один из операндов длинный, второй преобразуется в длинное, и результат будет этого же типа.

Иначе. Если один из операндов — целое без знака, другой преобразуется в целое значение без знака, и результат будет того же типа.

Иначе. Оба операнда должны быть типа int, и он же будет типом результата.

7. ВЫРАЖЕНИЯ

Приоритеты операций, входящих в выражения, соответствуют порядку основных подразделов данного раздела. Причем первый приоритет наивысший. Таким образом, например, выражения, фигурирующие в качестве операндов операции + (разд. 7.4), есть выражения, определенные в разделах 7.1—7.3. Внутри одного подраздела операции имеют одинаковый приоритет. В каждом разделе о таких операциях указывается, в каком порядке они выполняются: слева направо или справа налево. Приоритеты и порядок выполнения всех операций выражений приводятся в грамматике разд. 18.

Все, что не охватывается этими правилами и связано с порядком вычисления, остается неопределенным. В частности, транслятор «оставляет за собою право» вычислять подвыражения в порядке, который он сочтет наиболее эффективным, несмотря даже на то, что выражение включает побочный эффект. Порядок, при котором побочный эффект будет иметь место, остается неопределенным. Выражения, включающие коммутативные и ассоциативные операции (*, +, &, |, ^), могут произвольно переупорядочиваться, даже если будут добавляться скобки. Фиксировать определенный порядок вычислений можно явным сохранением промежуточного результата.

Обработка переполнения и контроль деления при вычислении выражения зависят от машины. Во всех существующих реализациях целое переполнение игнорируется. Реакция на деление на нуль и все инциденты для плавающей точки варьируются от машины к машине и обычно связаны с соответствующими библиотечными функциями.

7.1. Первичные выражения

Первичные выражения включают операции . , —>, индексирование и обращение к функции; эти операции выполняются слева направо.

Первичное выражение:
идентификатор
константа
строка
(*выражение*)
первичное выражение [*выражение*]
первичное выражение (*список выражений*$_{opt}$)
первичный адрес. идентификатор
первичное выражение —> *идентификатор*
список выражений:
выражение
список выражений , *выражение*

Любой идентификатор суть первичное выражение, при условии, что он был описан соответствующим образом, об этом будет сказано ниже. Его тип задается в его описании. Если тип идентификатора «массив ...», то тип значения выражения, состоящего из одного такого идентификатора, есть «ссылка на ...». Более того, идентификатор массива не есть адресное выражение. Аналогично идентификатор, описанный как «функция, дающая ...» при использовании в любой позиции, кроме имени функции, при обращении к ней преобразуется к типу «ссылка на функцию, дающую ...».

Любая константа суть первичное выражение. Его тип может быть int, long или double. Это зависит от вида константы. Символьные константы относятся к типу int, константы с плавающей точкой считаются двойными.

Любая строка также считается первичным выражением. Исходный ее тип «массив символов», но по приведенному выше правилу он переходит в тип «ссылка на символ», и в результате получается ссылка на первый символ в строке. (При некоторых инициациях существуют исключения из этого принципа; см. разд. 8.6.)

Выражения в скобках представляют собою первичные выражения. Их тип и значение идентичны типу и значению выражения внутри скобок. Наличие скобок не влияет на «адресность» выражения.

Первичное выражение, за которым в квадратных скобках следует выражение, считается первичным выражением. Интуитивно ясно, что это индексирование. Обычно первичное выражение относится к типу «ссылка на ...», индексирующее выражение — к типу int, а результат — к типу «...». По определению выражение E1 [E2] идентично выражению *((E1) + (E2)). Все, что нужно для понимания такой записи, содержится в этом разделе, причем об идентификаторе речь идет в разд. 7.1, а об операциях * и + соответственно в разд. 7.2. и 7.4; в разд. 14.3 эти сведения суммируются.

Обращение к функции представляет собою первичное выражение. За ним в скобках следует список (через запятую) выражений, являющихся фактическими аргументами этой функции. Первичное выражение должно относиться к типу «функция, дающая ...», а результат обращения будет типа «...». Как об этом будет сказано ниже, еще не упоминавшийся идентификатор (за ним сразу же следует левая, открывающая скобка) считается контекстуально описанным как имя функции, дающей целое значение; таким образом, в этом, наиболее распространенном случае функцию, дающую целую величину, можно не описывать.

Любой аргумент типа float преобразуется к двойной точности перед обращением, а типа char или short преобразуется к типу int, имена массивов, как обычно, преобразуются в ссылки. Автоматически больше никаких преобразований не происходит: в частности, транслятор не сравнивает типы фактических аргументов с типами формальных аргументов. Если же преобразование необходимо, то используется операция приведения (см. разд. 7.2 и 8.7).

В процессе обращения к функции создается копия каждого фактического параметра, т. е. передача аргументов в языке Си происходит строго по значению. Любая функция может менять значения своих формальных параметров, но эти изменения не оказывают никакого влияния на значения фактических параметров. С другой стороны, можно передать ссылку и сознавать , что функция может изменять значение объекта, на который указывает эта ссылка. Порядок вычисления аргументов в языке не определен, и заметим, что различные трансляторы в этом случае поступают по-разному.

К любой функции разрешены рекурсивные обращения.

Первичное выражение, за которым идет точка, а затем следует идентификатор есть выражение. Первое выражение должно быть адресным выражением, именующим некоторую запись или смесь, а идентификатор должен именовать элемент этой записи или смеси. В результате мы имеем адресное значение, указывающее на поименованный элемент записи или смеси.

Первичное выражение, за которым следует «стрелка», построенная из символов — и >, и некоторый идентификатор представляют собой выражение. Первое выражение должно быть ссылкой на запись или смесь, а идентификатор должен именовать элемент этой записи или смеси. В резуль-

тате получаем адресное значение, указывающее на поименованный элемент записи или смеси, на которую указывало ссылочное выражение.

Таким образом, выражение E1 — > MOS означает то же самое, что и (*E1). MOS. Записи и смеси рассматриваются в разд. 8.5. Приведенные здесь правила для использования записей и смесей не фиксируются строго, дабы можно было отступать от механизмов, связанных с типами. См. разд. 14.1.

7.2. Унарные операции

Выражения с унарными операциями выполняются справа налево.

Унарное выражение:
* *выражение*
& *адрес*
— *выражение*
! *выражение*
~ *выражение*
++ *адрес*
-- *адрес*
адрес ++
адрес --
(*имя типа*) *выражение*
sizeof *выражение*
sizeof (*имя типа*)

Унарная операция * означает *косвенность*: выражение должно быть ссылкой и в результате получается адрес, сопоставленный с объектом, на который указывала ссылка. Если тип выражения есть «ссылка на ...», то тип результата есть «...».

Результатом унарной операции — является ссылка на объект, сопоставленный с адресом. Если тип адреса есть «...», то тип результата — «ссылка на ...».

Результатом унарной операции & будет значение операнда с обратным знаком (отрицание). При этом выполняются обычные арифметические преобразования. Отрицание величины без знака образуется путем вычитания ее из 2^n, где n — число разрядов в значении типа int. Унарная операция + отсутствует.

Результатом операции логического отрицания ! будет 1, если операнд имел значение 0, и 0, если операнд был отличен от нуля. Результат относится к типу int. Эту операцию можно применять к значению любого арифметического типа или к ссылке.

Операция ~ означает обращение (двоичного представления) операнда. При этом выполняются обычные арифметические преобразования. Тип операнда должен быть целым.

Операция ++ (использованная как префикс) увеличивает значение объекта, сопоставленного с адресным операндом. Результатом является новое значение операнда, а не его адрес. Выражение ++x эквивалентно x+=1. Вопросы преобразования рассматриваются при обсуждении операции сложения (разд. 7.4) и операции присваивания (разд. 7.14).

Префиксная операция -- аналогично префиксной ++ уменьшает значение, заданное адресным операндом.

Если к некоторому адресу применить постфиксную операцию ++, то результатом будет значение, с которым сопоставлен этот адрес. После запоминания этого результата значение объекта увеличивается так же, как и при префиксной операции ++. Тип результата тот же, что и тип адресного выражения.

При применении к некоторому адресу постфиксной операции -- в результате получаем значение, с которым сопоставлен этот адрес. После

запоминания этого результата объект уменьшается так же, как и при префиксной операции — —. Тип результата тот же, что и тип адресного выражения.

Если перед выражением стоит в скобках имя типа, эта конструкция приводит к преобразованию выражения к поименованному типу. Такая конструкция называется *приведением.* Имена типов рассматриваются в разд. 8.7.

Операция sizeof выдает размер операнда в байтах. (Понятие *байт* в языке не определено, и о нем речь заходит только в связи со смыслом операции sizeof. Однако во всех существующих реализациях байтом называется память, объем которой позволяет сохранять один символ.) Если операция применяется к массиву, то результат — общее число байтов в массиве. Размер определяется на основе описания объектов, фигурирующих в выражении. Значение такой операции семантически представляет целую константу и может использоваться везде, где требуется указывать константу. В основном они используются для работы с подпрограммами типа распределения памяти или с подпрограммами ввода-вывода.

Операцию sizeof можно применять и к имени типа; оно указывается в скобках. В этом случае выдается в байтах размер любого объекта указанного типа.

Конструкция sizeof(*тип*) рассматривается как нечто целое, так что выражение sizeof (*тип*) — 2 означает (sizeof (*тип*)) — 2.

7.3. Мультипликативные операции

Мультипликативные операции *, / и % выполняются слева направо. При этом выполняются обычные арифметические преобразования.

Мультипликативное выражение:
выражение * *выражение*
выражение / *выражение*
выражение % *выражение*

Бинарная операция * означает умножение. Это ассоциативная операция, и выражения с несколькими умножениями на одном уровне транслятор может «перестраивать».

Деление обозначается бинарной операцией /. Если делятся положительные целые числа, то дробная часть отбрасывается так, что получается ближайший к нулю результат; если же один из операндов отрицателен, то форма округления зависит от машины. На всех упоминавшихся в книге машинах остаток имеет тот же знак, что и у делимого. И всегда (если b не равно нулю) справедливо равенство (a/b)*b+a%b=a.

Бинарная операция % означает взятие остатка от деления первого выражения на второе. Выполняются обычные арифметические преобразования. Операнды не должны быть типа float.

7.4. Аддитивные операции

Аддитивные операции + и — выполняются слева направо, при этом выполняются обычные арифметические операции. С каждой из операций связаны и некоторые другие возможности.

Аддитивное выражение:
выражение + *выражение*
выражение — *выражение*

Результат операции + есть сумма операндов. К ссылке на объект в массиве можно добавлять любое значение целого типа. Это значение в таких случаях преобразуется в адресное смещение. Для этого оно умножается на

длину объекта, на который указывает ссылка. В результате получается ссылка того же типа, что и исходная, но указывающая на другой объект в том же массиве, смещенный соответствующим образом относительно первоначального. Таким образом, если P — ссылка на некоторый объект в массиве, то выражение P+1 есть ссылка на следующий объект в этом массиве.

Никакие другие типы комбинировать со ссылками нельзя.

Операция + ассоциативна, и выражение с несколькими операциями сложения одного уровня может быть переупорядочено транслятором.

Результат операции — представляет собою разность операндов; при этом выполняются обычные арифметические преобразования. Кроме того, из ссылки можно вычитать значение любого целого типа, в этом случае проводятся преобразования, аналогичные преобразованиям для случая сложения.

Если вычитаются две ссылки на объекты одного типа, результат преобразуется (путем деления на длину объекта) в значение типа int, представляющее число объектов, разделяющих указанные объекты. В общем случае, если только речь не идет о ссылках на объекты одного массива. такое преобразование может давать неожиданные результаты. Это происходит из-за того, что ссылки на объекты одного типа не обязательно разнятся на величину, кратную длине объекта.

7.5. Операции сдвига

Операции сдвигов << и >> выполняются слева направо. И в том и в другом случае проводятся обычные арифметические преобразования операндов, каждый из которых должен быть целого типа. Затем правый операнд преобразуется к типу int, тип результата — это тип левого операнда. Результат операции не определен, если правый операнд отрицательный, равен или превышает размер объекта в разрядах.

Сдвиговое выражение:
выражение << *выражение*
выражение >> *выражение*

Значение выражения E1<<E2 есть E1 (рассматриваемое как набор разрядов), сдвинутое влево на E2 разрядов; освобождающиеся разряды заполняются «нулями». Значение E1>>E2 есть E1, сдвинутое на E2 разрядов вправо. Если E1 типа unsigned, то гарантируется, что сдвиг вправо—логический, т. е. идет заполнение нулями. В других же случаях (например, на PDP-11) может выполняться и арифметический сдвиг (т. е. свободные разряды заполняются копией знакового разряда).

7.6. Операции отношения

Операции отношения выполняются слева направо, но этот факт мало что дает, ибо выражение a<b<c все равно интерпретируется не так, как принято в математике.

Выражение отношения:
выражение < *выражение*
выражение > *выражение*
выражение <= *выражение*
выражение >= *выражение*

Операции < (меньше), > (больше), <= (меньше или равно) и >= (больше или равно) дают результат 0, если указанное отношение несправедливо, и 1, если справедливо. Тип результата int. Обычные арифметические преобразования выполняются. Можно сравнивать две ссылки,

результат зависит от взаимного расположения в памяти объектов, указанных ссылками.

7.7. Операции сравнения на равенство

Выражение равенства:
выражение == *выражение*
выражение != *выражение*

Операции == (равно) и != (не равно) полностью аналогичны операциям отношения, за исключением того, что их приоритет ниже. (Таким образом, a<b==c<d дает результат 1, если выражения a<b и c<d имеют одно и то же значение.)

Ссылку можно сравнивать с целым значением, но результат зависит от машины, если только целое не есть константа 0. Гарантируется, что если ссылке присвоено значение 0, то она не указывает на какой-либо объект и будет равна нулю; при обычных преобразованиях такая ссылка считается нулем.

7.8. Поразрядная операция И

Выражение И:
выражение & *выражение*

Операция & ассоциативна, и выражение, куда она входит, может быть переупорядочено. Обычные арифметические преобразования выполняются. Результат — поразрядная функция И от операндов. Операция применима только к операндам целого типа.

7.9. Поразрядная (исключающая) операция ИЛИ

Выражение исключающего ИЛИ:
выражение ^ *выражение*

Операция ^ ассоциативна, и поэтому выражение, включающее такую операцию, может быть переупорядочено. Выполняются обычные арифметические преобразования. Результат — поразрядная исключающая функция ИЛИ от операндов. Операция применима только к целым операндам.

7.10. Поразрядная (включающая) операция ИЛИ

Выражение ИЛИ:
выражение | *выражение*

Операция | ассоциативна, и выражение, включающее |, может быть переупорядочено. Выполняются обычные арифметические операции преобразования; результат — поразрядная функция ИЛИ операндов. Операция применяется только к целым операндам.

7.11. Логическая операция И

Логическое выражение И:
выражение && *выражение*

Операции && выполняются слева направо. Такая операция дает результат 1, если оба ее операнда отличны от нуля, в других случаях результат — нуль. В отличие от & при операции && гарантируется вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд равен 0.

Операнды не обязательно одного типа, но каждый должен быть одного из фундаментальных типов или быть ссылкой. Результат всегда типа int.

7.12. Логическая операция ИЛИ

Логическое выражение ИЛИ:
выражение выражение

Операции || выполняются слева направо. Операция дает 1, если один из ее операндов отличен от нуля, в противном случае результат — нуль. В отличие от операции | при операции || гарантируется вычисление слева направо; более того, второй операнд не вычисляется, если значение первого операнда отлично от нуля.

Операнды не обязательно должны быть одного типа, но это должен быть один из фундаментальных типов или ссылка. Результат всегда типа int.

7.13. Операция условия

Условное выражение:
выражение ? выражение : выражение

Условные выражения выполняются справа налево. Выполнение одного условного выражения происходит следующим образом: сначала вычисляется первое выражение; если оно отлично от нуля, то результат — значение второго выражения, иначе — значение третьего выражения. Если это можно, то для приведения второго и третьего выражений к общему типу выполняются обычные арифметические преобразования; иначе, если оба выражения — ссылки одного типа, результат будет иметь этот общий тип; иначе, одно выражение должно быть некоторой ссылкой, а другое — константой 0 и результат имеет тип этой ссылки. Вычисляется только второе или третье выражение.

7.14. Операции присваивания

Существует несколько операций присваивания, все они выполняются справа налево. Для всех них требуется, чтобы левым операндом был адрес, и тип присваивающего выражения есть тип его левого операнда. Результат операции — значение, хранящееся в левом операнде после того как произошло присваивание. Обе части составной операции присваивания являются отдельными лексемами.

Присваивающее выражение:
адрес = *выражение*
адрес += *выражение*
адрес −= *выражение*
адрес *= *выражение*
адрес /= *выражение*
адрес %= *выражение*
адрес >>= *выражение*
адрес <<= *выражение*
адрес &= *выражение*
адрес ^= *выражение*
адрес |= *выражение*

При простом присваивании = значение выражения заменяет собой значение объекта, с которым сопоставлен адрес. Если оба операнда арифметического типа, то правый операнд, прежде чем выполнится присваивание, преобразуется к типу левого операнда.

Выполнение выражения вида E1 *op*= E2 можно пояснить, сказав, что оно эквивалентно выражению E1 = E2*op* (E2), однако E1 вычисляется лишь один раз. В операциях += и −= левый операнд может быть ссылкой, в этом случае правый операнд (целый) преобразуется, как объяснялось в разд. 7.4. Все правые операнды и все левые операнды, не являющиеся ссылками, должны быть арифметического типа.

Сейчас трансляторы позволяют присваивать ссылку объекту целого типа, целое — ссылке и любую ссылку — ссылке другого типа. Присваивание — это просто копирование, без преобразования. При таком программировании перенос программ может оказаться невозможным, так как могут возникать ссылки, приводящие при их использовании к адресным аварийным ситуациям. Однако гарантируется, что присваивание ссылке константы 0 будет порождать нулевую ссылку, отличную от ссылки на какой-либо другой объект.

7.15. Операция запятая

Выражение с запятой:
выражение , выражение

Пара выражений, разделенных запятой, вычисляется слева направо, и значение левого выражения уничтожается. Тип и значение результата есть тип и значение правого операнда. Эти операции выполняются слева направо. В контекстах, где запятая имеет специальное значение, скажем в любом списке фактических аргументов функции (разд. 7.1.) или в списках инициации (разд. 8.6), операция запятая, такая, как описано в этом разделе, может появляться только в скобках, например обращение содержит три аргумента, причем второй имеет значение 5.

```
f(a, (t=3, t+2), c)
```

8. ОПИСАНИЯ

Описания используются для определения интерпретации каждого из идентификаторов; не обязательно, что они приводят к выделению памяти, сопоставляемой с идентификатором. Описания имеют вид

Описание:
спецификация описания список описателей$_{opt}$;

Описатели в списке описателей содержат описываемые идентификаторы. Спецификации описаний состоят из последовательности спецификаций типа и класса памяти.

Спецификации описания:
спецификация типа спецификации описания$_{opt}$
спецификация памяти спецификации описания$_{opt}$

Этот список должен быть самосогласованным, в смысле, описываемом ниже.

8.1. Спецификация класса памяти

Спецификации класса памяти могут быть такими:

спецификация памяти:
`auto`
`static`
`extern`
`register`
`typedef`

Спецификация typedef **не резервирует памяти и называется «спецификацией памяти» только с точки зрения синтаксического удобства; о нем речь идет в разд. 8.8. Смысл различных классов памяти обсуждается в разд. 4.**

Описания auto, static **и** register **выступают и как определения, указывающие, что в этом случае нужно зарезервировать память соответствующего объема. В случае описания** extern **для указанных идентификаторов вне функции, где они описаны, должно существовать некоторое внешнее определение (разд. 10).**

Описание register лучше всего понимать как описание auto, подсказывающее в то же время транслятору, что описанные переменные используются достаточно интенсивно. Эффективны только несколько первых таких описаний. Кроме того, на регистрах могут храниться только переменные вполне определенных видов, на PDP-11 это переменные типа int, char или ссылка. Для переменных, хранящихся в регистрах, существует и другое ограничение: к ним не применима адресующая операция &. Если использовать описания регистров должным образом, то можно надеяться получить более короткие и более быстрые программы, однако в будущем улучшения в блоках формирования команд могут сделать их бесполезными.

В описании можно задать самое большое одну спецификацию памяти. Если спецификация памяти опущена, подразумевается, что в функции она auto, а вне — extern. Исключение: сами функции никогда не бывают автоматическими.

8.2. Спецификация типа

Спецификация типа имеет вид:

спецификация типа:
char
short
int
long
unsigned
float
double
спецификация записи или смеси
имя определенного типа

Слова long, short и unsigned можно мыслить как прилагательные, и поэтому допускаются такие комбинации:

```
short int
long int
unsigned int
long float
```

Последняя означает то же самое, что и double. Во всех остальных случаях в описании можно задать самое большое одну спецификацию типа. Если спецификация типа пропущена, подразумевается, что она int.

Спецификации для записей и смесей рассматриваются в разд. 8.5, случай же с именем определенного типа — в разд. 8.8.

8.3. Описатели

Список описателей, фигурирующих в описании,— это последовательность через запятую описателей, каждый из которых может иметь инициатор.

Список описателей:
описатель с инициатором
описатель с инициатором , список описателей
описатель с инициатором:
описатель инициатор$_{opt}$

Инициаторы разбираются в разд 8.6. Спецификаторы в описании указывают тип и класс памяти объектов, упоминающихся в описателях. Описатели имеют такой синтаксис:

описатель:
идентификатор
(описатель)
** описатель*
описатель( )
описатель [*константное выражение* $_{opt}$]

Порядок выполнения такой же, как в выражении.

8.4. Смысл описателей

Каждый описатель предполагает, что если конструкция такого вида, как этот описатель, появляется в выражении, то она соответствует объекту указанных типа и класса памяти. Любой описатель содержит точно один идентификатор — именно этот идентификатор и описывается.

Если в качестве описателя появляется просто идентификатор, то он имеет тип, заданный спецификацией в заголовке описания.

Описатель в круглых скобках идентичен простому идентификатору, однако скобки могут изменять смысл сложных описателей (см. приведенные ниже примеры).

Теперь представим себе описание T D1, где T — спецификация типа (например, int или что еще), а D1 — описатель. Предположим, это описание определяет, что идентификатор имеет тип «...T», где «...» пусто, если D1 просто идентификатор. Таким образом, тип x в «int x» есть просто int. Если D1 имеет вид *D, то тип упомянутого идентификатора « ... ссылка на T».

Если D1 имеет вид D( ), то упомянутый идентификатор имеет тип « ... функция, дающая T».

Если D1 имеет вид D [*константное выражение*] или D [ ], то упомянутый идентификатор имеет тип «... массив T». В первом случае константное выражение есть выражение, значение которого определяется во время трансляции, причем его тип — int. (Константное выражение определяется в разд. 15.) Если рядом стоят несколько спецификаций «массив», то порождается многомерный массив; константное выражение, определяющее границы массива, можно опускать только для первого элемента этой последовательности. Такой пропуск имеет смысл (полезен), если это внешний массив и фактическое его определение, резервирующее память, приводится где-либо еще. Первое константное выражение можно опускать и в том случае, если за описателем следует инициация. В этом случае размер определяется числом заданных начальных значений.

Массивы можно строить на основе любого из основных типов, ссылок, записей или смесей, других массивов (порождая многомерные массивы).

Однако не все возможности, допускаемые приведенным выше синтаксисом, на самом деле можно использовать. Существуют такие ограничения: функции не могут выдавать массивов, записей, смесей или функций, хотя они могут выдавать ссылки на такие объекты; не существует массивов функций, хотя могут быть массивы ссылок на функции. Аналогично записи или смеси не могут содержать функции, но могут содержать ссылки на функции.

Например, описание

```
int i, *ip, f(), *fip(), (*pfi)();
```

описывает целое i, ссылку ip на целое, функцию f, дающую целое, функцию fip, дающую ссылку на целое, и ссылку pfi на функцию, дающую целое. Особенно полезно сравнить два последних случая. Толкование *fip( ) таково: *(fip( )), так что описание предполагает, а в некоторых конструкциях в выражении и требует, обращение к функции fip, а затем кос-

венное обращение через результат (ссылку) для получения целого. В описателе (*pfi) ( ) дополнительные скобки необходимы, так как это тоже выражение для того, чтобы указать обращение через ссылку на функцию за функцией, к которой затем идет обращение; оно (обращение) дает в результате целое значение.

Еще один пример:

```
float fa[17], *afp[17];
```

Здесь описывается массив чисел типа float и массив ссылок на такие числа. Наконец, в описании

```
static int x3d[3][5][7];
```

вводится статический трехмерный массив целых чисел размером 3×5×7. Более детально: x3d — массив из трех элементов, каждый элемент — массив из пяти массивов, а в каждом таком массиве уже по семь целых чисел. В выражении может появиться и иметь смысл любая из таких конструкций: x3d, x3d[i], x3d[i] [j], x3d[i] [j] [k]. Первые три из них имеют тип «массив», а последняя — int.

8.5. Описание записей и смесей

Запись есть объект, состоящий из последовательности поименованных элементов. Каждый элемент может иметь свой тип (любой). Смесь есть объект, могущий в любое время содержать один (любой) из нескольких элементов. Спецификация для записи и смеси имеет одну и ту же структуру.

Спецификация записи или смеси:

запись или смесь { *список описаний записи* }
идентификатор записи или смеси { *список описаний записи* }
идентификатор записи или смеси

запись или смесь:
struct
union

Список описаний записи есть последовательность описаний элементов записи или смеси.

Список описаний записи:
описание записи
описание записи список описаний записи

описатель записи:
спецификация типа список описателей записи;

список описателей записи:
описатель записи
описатель записи , список описателей записи

В обычном случае описатель записи — это лишь описатель элементов записи или смеси. Элемент записи может состоять и из заданного числа разрядов. Такой элемент называется так же *полем;* его размер отделяется от имени поля двоеточием.

Описатель записи:
описатель
описатель : константное выражение
: константное выражение

Описываемые внутри записи объекты получают некоторые адреса, причем эти адреса увеличиваются по мере продвижения слева направо. Каждый элемент, не относящийся к классу полей, начинается с адреса, присущего его типу, поэтому внутри записи могут встречаться безымянные пустоты. Элементы-поля пакуются в машинные слова, соответствующие целому типу, но не переходят границы слов. Если поле не помещается в оставшейся части

слова, то оно помещается в следующее слово. Никакое поле не может быть больше слова. Поля размещаются справа налево в PDP-11 и слева направо—в других машинах.

Описатель записи без какого-либо описателя: только двоеточие и размер определяет безымянное поле. Оно может оказаться полезным при подгонке к задаваемому снаружи смещению. Как особый случай безымянное поле размера 0 указывает на выравнивание следующего поля к границе слова. «Следующее поле»—это именно поле, а не просто обычный элемент записи, так как в этом случае выравнивание произойдет автоматически.

В языке не вводится каких-либо ограничений на тип величин, описывающихся как поля, но при реализации не требуется заботиться о типах, отличных от целого. Более того, даже поля типа int можно рассматривать как величины без знака. В PDP-11 поля не имеют знака и могут быть только целыми. Во всех реализациях нет массивов из полей, и к полям нельзя применить операцию адресации &, поэтому ссылок на поля нет.

Смесь можно мыслить как запись, все элементы которой начинаются со смещением 0, и ее размера хватает для хранения любого из ее элементов. Смесь в любой момент может хранить самое большое один из своих элементов.

Спецификация записи или смеси второго типа, т. е.

struct *идентификатор список описаний записи*
union *идентификатор список описаний записи*

описывает идентификатор как тип записи или смеси именно той структуры, которая приведена в списке описаний. После этого в последующих описаниях можно пользоваться спецификацией третьего типа:

struct *идентификатор*
union *идентификатор*

Типы записи позволяют определять записи, ссылающиеся сами на себя; они позволяют однажды выписать громоздкую часть описания и использовать ее несколько раз. Описать запись или смесь, содержащую вхождение самих себя нельзя, но запись или смесь могут содержать ссылку на самих себя.

Имена элементов и типов записей могут совпадать с обычными переменными. Однако имена элементов должны отличаться от имен типов записей.

Две записи могут иметь общую начальную последовательность элементов, т. е. один и тот же элемент может появляться в двух различных записях, если он и там, и там имеет один и тот же тип и если все предыдущие элементы были такими же. (Фактически транслятор только проверяет, что некоторое имя в двух различных записях имеет одинаковый тип и одинаковое смещение от начала; если предыдущие элементы были другими, то такая конструкция не переносится с машины на машину.)

Простой пример описания записи:

```
struct tnode {
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

Запись содержит массив из 20 символов, целое и две ссылки на подобные же записи. При наличии такого описания описание

говорит, что s — запись упомянутого вида, а sp — ссылка на запись такого вида. После таких описаний выражение

```
sp->count
```

ссылается на элемент count в записи, на которую указывает ссылка sp, а выражение

```
s.left
```

относится к ссылке на левое поддерево в записи s;

```
s.right->tword[0]
```

относится к первому символу элемента tword в правом поддереве в s.

8.6. Инициация

Любой описатель может задать начальное значение описываемого идентификатора. Инициатор начинается с символа = и состоит из выражения или списка значений, заключенного в скобки (фигурные).

Инициатор:
= *выражение*
= { *список инициаторов* }
= { *список инициаторов*, }

список инициаторов:
выражение
список инициаторов, *список инициаторов*
{ *список инициаторов* }

Все выражения инициаторов для статических или внешних переменных должны быть константными выражениями (рассмотрены в разд. 15) или выражениями, сводящимися к адресам предварительно описанных переменных, при этом допускается смещение от них на величину, опять же заданную константным выражением. Автоматические и регистровые переменные могут инициироваться произвольными выражениями, включающими константы и предварительно описанные переменные и функции.

Если статическая или внешняя переменные не инициируются, то гарантируется, что они получат значение 0; неинициированные же автоматические переменные или регистровые будут содержать непредсказуемый «мусор».

Если инициатор относится к *скаляру* (т. е. ссылке или любому объекту арифметического типа), то он представляет собой единственное выражение (возможно, в скобках). Начальное значение этого объекта определяется этим выражением, при этом выполняются те же преобразования, что и при присваивании.

Если инициатор относится к составному объекту (записи или массиву), то он состоит из списка выражений, отделенных друг от друга запятыми, заключенного в скобки; эти выражения—инициаторы для элементов, записанные в порядке перечисления элементов или увеличения индексов. Если составной объект содержит составные подобъекты, то это правило распространяется рекурсивно и на них. Если в списке инициаторов меньше, чем элементов в объекте, то оставшиеся принимают нулевые значения. Автоматические составные объекты и смеси инициировать не разрешается.

Скобки можно толковать следующим образом. Если инициатор начинается с левой скобки, то последующий список инициаторов, разделенных запятыми, инициирует элементы составного объекта; если инициаторов больше, чем объектов,— это ошибка. Если же инициатор не начинается со скобки, то из списка берется столько элементов, сколько элементов в объекте; оставшиеся элементы списка используются для инициации следующих элементов объекта, в состав которого данный входит как часть.

И наконец, для инициации элементов символьного массива можно использовать строку. Это некоторая сокращенная форма инициации, где последовательность символов строки инициирует элементы массива.

Пример. Описание

```
int x[] = { 1, 3, 5 };
```

описывает и инициирует x как одномерный массив из трех элементов, так как размер его не задан, но есть три инициатора:

```
float y[4][3] = {
      { 1, 3, 5 },
      { 2, 4, 6 },
      { 3, 5, 7 },
};
```

В такой инициации скобки расставлены полностью: 1, 3, 5 инициируют первую строку массива y [0], т. е. y [0] [0], y [0] [1] и y [0] [2]. Аналогично две следующие строки инициируют y [1] и y [2]. Инициация заканчивается ранее, чем требуется, поэтому y [3] инициируется нулями. Точно того же эффекта можно достичь с помощью описания

```
float y[4][3] = {
      1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Со скобки начинается инициатор для y, а не для y[0], поэтому из списка используются три элемента. Аналогично следующие три элемента из списка берутся для y[1], а затем для y[2]. Описание же

```
float y[4][3] = {
      { 1 }, { 2 }, { 3 }, { 4 }
};
```

инициирует первый столбец y (считая его двумерным массивом), а оставшиеся заполняются нулями.

Наконец, описание

```
char msg[] = "Syntax error on line %s\n";
```

задает символьный массив, элементы которого инициированы символами строки.

8.7. Имена типов

В программе в двух местах желательно употребление «названий» для типов данных: для задания явного преобразования типа и в качестве аргумента для «функции» sizeof. Это можно сделать с помощью «имени типа», такое имя вводится с помощью описания некоторого фиктивного безымянного объекта нужного типа, причем имя объекта вообще отсутствует.

имя типа:
    спецификация типа абстрактный описатель
абстрактный описатель:
    пусто
    (абстрактный описатель)
    ** абстрактный описатель ( )*
    абстрактный описатель константное выражение $_{opt}$]

Чтобы избежать двусмысленности в конструкции:

(абстрактный описатель)

абстрактный описатель не должен быть пустым. При таком ограничении возможно единственным образом локализовать те места в абстрактном описателе, где должен был бы появляться идентификатор, будь эта конструкция описателем в некотором описании. Поименованный тип — это тип, имеющий этот гипотетический идентификатор. Например, описания:

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

именуют соответственно типы: «целое», «ссылка на целое», «массив из трех ссылок на целое», «ссылка на массив из трех целых», «функция, дающая ссылку на целое», «ссылка на функцию, дающую целое».

8.8. Описание типа

Описание, в котором в качестве класса памяти фигурирует слово typedef, не определяет какую-либо память, а вместо этого определяет идентификатор, (его позже можно использовать в качестве служебного слова), как если бы это было название основного или производного типа.

имя описанного типа:

идентификатор

Внутри области действия описания со словом typedef каждый идентификатор, входящий в такое описание, становится синтаксическим эквивалентом слова для типа, связанного с этим идентификатором в соответствии с порядком, описанным в разд. 8.4. Например, после описаний

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im;} complex;
```

конструкции

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

будут уже вполне допустимыми; тип переменной distance — int, тип metricp — ссылка на int, z — определенная выше запись, а zp — ссылка на такую запись.

Описание typedef не вводит новые типы, а вводит лишь синонимы для типов, которые можно было бы задать и другим способом. Таким образом, в приведенном выше примере тип переменной distance считается точно таким же, как и у любого другого объекта типа int.

9. ОПЕРАТОРЫ

Все операторы, за исключением специально оговоренных случаев, выполняются один за другим.

9.1. Оператор-выражение

Чаще всего операторами бывают выражения; в этом случае оператор имеет такой вид:

выражение;

Обычно операторы-выражения являются присваиваниями или обращениями к функциям.

9.2. Составной оператор или блок

Для тех случаев, когда вместо одного оператора желательно использовать несколько, предусмотрен составной оператор (иногда его называют «блок»).

Составной оператор:
{ *список описаний*$_{opt}$ *список операторов*$_{opt}$ }
список описаний:
описание
описание список описаний
список операторов:
оператор
оператор список операторов

Если какие-либо из идентификаторов в списке описаний уже были до этого описаны, то внешние описания запоминаются на время работы данного блока, а после окончания они вновь вступают в силу.

Инициация автоматических или регистровых переменных выполняется при каждом входе в блок через его начало. Сейчас можно передать управление внутрь блока (но на это не стоит ориентироваться); при этом инициация выполняться не будет. Статические переменные инициируются только один раз, в начале выполнения программы. Внешние описания внутри блока не резервируют никакой памяти, поэтому инициация их не допускается.

9.3. Условный оператор

Существуют два вида условных операторов:

if (*выражение*) *оператор*
if (*выражение*) *оператор* else *оператор*

В любом случае вычисляется выражение, и если оно отлично от нуля, то выполняется первый подоператор. Во втором случае второй подоператор выполняется, если значение выражения нуль. Как обычно, двусмысленное использование else разрешается путем его сопоставления с ближайшим if без парного else.

9.4 Оператор while

Оператор имеет такой вид:

while (*выражение*) *оператор*

Подоператор повторно выполняется до тех пор, пока значение выражения остается отличным от нуля. Проверка происходит перед каждым выполнением оператора.

9.5. Оператор do

Оператор имеет вид

do *оператор* while (*выражение*)

Подоператор повторно выполняется до тех пор, пока выражение не станет нулем. Проверка происходит после каждого выполнения оператора.

9.6. Оператор for

Оператор имеет вид

for (*выражение-1*$_{opt}$; *выражение-2*$_{opt}$; *выражение-3*$_{opt}$) *оператор*

Этот оператор эквивалентен такому фрагменту программы:

выражение-1
while (*выражение-2*) {
оператор
выражение-3;

Таким образом, первое выражение задает исходные установки для цикла, второе выполняет проверку, а третье выражение часто задает приращение и выполняется после каждой итерации.

Любое или даже все выражения можно опускать. Пропуск *выражения-2* неявно заменяет конструкцию while на while(1); если же пропущены другие выражения, то они просто опускаются из поясняющей схемы.

9.7. Оператор переключателя

Оператор переключателя вызывает передачу управления на один из нескольких операторов, в зависимости от значения выражения. Он имеет вид

switch (*выражение*) *оператор*

При вычислении выражения выполняются обычные арифметические преобразования, но результат должен быть типа int. Оператор обычно бывает составным. В нем любой из операторов можно пометить одним или несколькими префиксами, имеющими вид

case *константное выражение*:

Константное выражение должно быть целым. Никакие две константы варианта в одном переключателе не могут быть одинаковыми. Константное выражение точно определяется в разд. 15.

Может также встречаться, но только единожды, префикс вида

default :

При выполнении оператора переключателя вычисляется выражение, и его значение сравнивается с каждой из констант вариантов. Если одна из констант вариантов равна значению выражения, то управление передается на оператор, идущий за этой константой. Если ни одна из констант с выражением не совпала, но есть префикс default, то управление передается на оператор с таким префиксом. Если префикса default нет и совпадения не было, то в переключателе не выполняется ни один оператор.

Если операторы, выполняемые по выбору, не приводят к каким-либо передачам управления, то программа продолжает «идти по префиксам» беспрепятственно.

Обычно оператор, с которым имеет дело переключатель, бывает составным. В начале этого оператора могут быть описания, но инициация автоматических или регистровых переменных не выполняется.

9.8. Оператор разрыва

Оператор вида break; приводит к окончанию выполнения ближайшего внешнего оператора while, do, for или swith; управление передается оператору, следующему за заканчиваемым.

9.9. Оператор продолжения

Оператор continue; приводит к передаче управления в организующую цикл часть ближайшего внешнего оператора while, do или for, т. е. в конец цикла. Более того, можно сказать, что в каждом из операторов:

```
while (...) {        do {              for (...) {
  ...                  ...               ...
contin: ;            contin: ;         contin: ;
}                    } while (...);    }
```

оператор continue эквивалентен оператору goto contin. Следом за contin: идет пустой оператор, о котором речь идет в разд. 9.13.

9.10. Оператор возврата

Выход из функции к обратившейся программе происходит с помощью оператора return одного из таких двух видов:

```
return ;
return выражение;
```

В первом случае возвращаемое значение не определено. Во втором случае обратившейся программе выдается значение выражения. Если нужно, то, как и при присваивании, производится преобразование выражения к типу функции, где встречается этот оператор. Выход из функции «через конец» эквивалентен возврату без выдаваемого значения.

9.11. Оператор перехода

Безусловную передачу управления можно выполнить с помощью оператора

```
goto идентификатор;
```

Идентификатор должен быть некоторой меткой (см. разд. 9.12), помещенной в текущей функции.

9.12. Помеченный оператор

Перед любым оператором можно поставить метку, имеющую вид

идентификатор:

Такая конструкция описывает идентификатор в качестве метки. Метка используется только для указания точки перехода в операторе goto. Областью действия метки является текущая функция, исключая любые подблоки, в которых этот идентификатор переописывается (см. разд. 11).

9.13. Пустой оператор

Пустой оператор имеет вид

;

Его полезно использовать, если нужно поставить метку перед концом составного оператора или же для задания пустого тела цикла в таких операторах, как while.

10. ВНЕШНИЕ ОПРЕДЕЛЕНИЯ

Любая программа на нашем языке содержит последовательность внешних определений. Внешнее определение описывает идентификатор как относящийся к классу памяти extern (подразумеваемому) или static и имеющий заданный тип. Спецификация типа (см. разд. 8.2) может так же отсутствовать, в этом случае подразумевается, что тип — int. Область действия внешних определений распространяется до конца файла, в котором они описаны, точно так же, как описания действуют до конца блока. Синтаксис внешних определений аналогичен синтаксису всех описаний, однако только на этом уровне можно задавать тело функции.

10.1. Определение внешних функций

Определение функции имеет такой вид:

определение функции:
спецификации описания$_{opt}$ *описатель функции тело функции*

Из спецификаций описания допускаются лишь спецификации памяти extern и static (различие между ними объясняется в разд. 11.2). Описатель функции подобен описателю для «функция, дающая ...», но в нем определяется еще и список формальных параметров.

описатель функции:
описатель (*список параметров*$_{opt}$)
список параметров:
идентификатор
идентификатор, список параметров

Тело функции имеет вид

тело функции:
список описаний составной оператор

В списке описаний можно описывать только идентификаторы из списка параметров. Если тип идентификатора не задан, то он считается int. Из классов памяти можно задавать только регистры; если сделать так, то при обращении к функции соответствующий параметр будет, если возможно, скопирован в регистр.

Простой пример полного определения функции:

```
int max(a, b, c)
int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Здесь int—спецификация типа; max(a, b, c)—описатель функции; int, a, b, c—список описаний для формальных параметров; {...}—блок, задающий само действие.

Все фактические параметры типа float преобразуются к типу double, так что в описателях формальных параметров float можно просто воспринимать как double. Аналогично тому, как упоминание массива в любом контексте (в частности, и в фактических параметрах) рассматривается как ссылка на первый его элемент, описание формального параметра вида «массив...» можно читать как «ссылка на ...». И наконец, так как записи, смеси и функции передавать как фактические параметры нельзя, то описывать формальные параметры как записи, смеси или функции бесполезно, хотя ссылки на такие объекты, конечно, допускаются.

10.2. Определение внешних данных

Определение внешних данных имеет вид

определение данных:
описание

Класс памяти таких данных может быть extern (он обычно подразумевается) или static, но не может быть auto или register.

11. ПРАВИЛА ДЛЯ ОБЛАСТЕЙ ДЕЙСТВИЯ

В языке Си не обязательно транслировать сразу всю программу: исходный текст программы может храниться в нескольких файлах, а заранее оттранслированные подпрограммы могут загружаться из библиотек. Взаи-

модействие функций может осуществляться как путем явных взаимных обращений, так и манипуляциями со внешними данными.

Поэтому можно говорить о двух видах областей действия. Первый вид можно назвать *лексической областью* действия идентификатора: фактически это фрагмент программы, где можно пользоваться идентификатором, не рискуя получить диагностическое сообщение «неописанный идентификатор». Второй вид области действия связан со внешними идентификаторами, в этом случае речь идет о правилах, определяющих, ссылаемся ли мы на один и тот же объект при употреблении некоторого идентификатора.

11.1 Лексические области действия

Лексическая область действия идентификатора, описанного во внешнем определении, простирается от этого определения до конца входного файла, где оно появилось. Лексическая область действия идентификаторов формальных параметров распространяется на всю функцию, с которой они связаны. Лексическая область действия идентификатора, описанного в заголовке блока, распространяется до конца этого блока. Лексическая область метки действия — это вся функция, в которой она появилась.

Поскольку все использования некоторого внешнего идентификатора относятся к одному и тому же объекту (см. разд. 11.2), то транслятор проверяет на совместимость все описания этого внешнего идентификатора. Фактически его областью действия становится весь файл, где он появляется.

Однако во всех случаях, если идентификатор явно описывается в заголовке блока, включая и блок, представляющий функцию, то действие всех внешних по отношению к этому блоку описаний идентификатора «приостанавливается» до конца блока.

Напомним еще раз (разд. 8.5.), что идентификаторы, сопоставленные с обычными переменными, и идентификаторы для элементов записей и смесей или типов записей представляют собой два различных класса: конфликты между ними невозможны. На элементы и признаки распространяются те же правила локализации, что и для других идентификаторов. Имена, описываемые с помощью слова typedef, относятся к тому же классу, что и обычные идентификаторы. Их можно переописывать во внутренних блоках, но во внутренних описаниях должен фигурировать явный тип:

```
typedef float distance;
...
{
        auto int distance;
        ...
```

Во втором описании должен указываться тип int или же оно будет восприниматься как описание без описателей и с типом distance.

11.2. Область действия внешних имен

Если в функции используется описанный как внешний идентификатор, то где-либо в файлах или библиотеках, составляющих полную программу, должно быть внешнее определение этого идентификатора. Все функции данной программы, употребляющие некоторый внешний идентификатор, имеют дело с одним и тем же объектом, поэтому надо быть внимательным, чтобы тип и размер, указанные в определении, совпадали с соответству-

ющими атрибутами, указанными в каждой из функций, работающих с этой информацией.

Появление зарезервированного слова extern во внешнем определении указывает, что память для описываемых в данный момент идентификаторов должна быть выделена при обработке другого файла. Таким образом, в программе, состоящей из нескольких файлов, определение внешних данных без спецификации extern должно появляться в одном и только одном файле. Во всех других файлах, где нужно дать внешнее определение того же идентификатора, в определение следует включать extern. Инициация идентификатора может выполняться только в описании, выделяющем память.

Идентификаторы, описанные как static на верхнем уровне во внешних описаниях и других файлах невидимы. Функции можно описывать как static.

12. КОМАНДЫ УПРАВЛЕНИЯ ТРАНСЛЯЦИЕЙ

Транслятор для языка Си имеет препроцессор, выполняющий подстановки для макровызовов, проводит условную трансляцию и вставляет указанные файлы. Для такого препроцессора предназначены строки текста, начинающиеся с символа #. Синтаксис этих строк не связан с синтаксисом самого языка. Они могут появляться в любом месте и оказывают действие на весь текст до конца файла входной программы. (Обычные правила областей действия на них не распространяются.)

12.1. Замена лексем

Команда (строка) имеет вид

#define *идентификатор подстановка*

Обратите внимание: в конце нет никакой точки с запятой. Такая команда заставляет препроцессор заменять последующие вхождения идентификатора на указанную строку лексем. Если команда имеет вид

#define *идентификатор* (*идентификатор,..., идентификатор*) *подстановка*

причем между первым идентификатором и (нет никакого пробела, то это определение макроподстановки с аргументами. После такого описания вхождение первого идентификатора с последующей скобкой (, лексемами, разделенными запятыми, и скобкой) заменяется на строку-подстановку из определения. Причем каждое вхождение идентификатора, упомянутого среди формальных параметров определения, заменяется на соответствующую лексему-строку из обращения. Фактические аргументы обращения суть лексемы-строки, отделенные одна от другой запятой, однако запятая, встретившаяся в кавычках или «защищенная» скобками, уже не считается разделителем для аргументов. Число формальных и фактических параметров должно быть одинаковым. На тексты внутри строк или символьные константы механизмы подстановок не распространяются.

Какого бы вида подстановка ни произошла, результирующая строка вновь просматривается и в ней ищутся определенные идентификаторы. При описании и той и другой подстановки могут встречаться «длинные» определения, продолжающиеся в следующей строке, для этого в конце строки, у которой есть продолжение, ставится символ \.

Определения такого типа очень хорошо подходят для введения констант — параметров реализации, например, в таких комбинациях:

```
#define TABSIZE 100

int table[TABSIZE];
```

Команда вида

#undef *идентификатор*

приводит к тому, что препроцессор начинает считать указанный идентификатор неопределенным, т. е. не подлежащим замене.

12.2. Включение файла

Команда вида

#include *"имя файла"*

приводит к тому, что вместо этой команды подставляется содержимое всего файла с указанным именем. Поименованный файл сначала разыскивается в каталоге начального входного файла. Если его там нет, то он ищется в других «стандартных» местах. При другом варианте команды

#include <*имя файла*>

поиск идет только в стандартных местах, а не в каталоге входного файла.

Команды включения могут оказаться «вставленными» одна в другую.

12.3. Условная трансляция

Команда транслятора, имеющая вид

#if *константное выражение*

проверяет, будет ли отличным от нуля выражение, составленное из констант (см. разд. 15). Команда вида

#ifdef *идентификатор*

проверяет, определен ли в данный момент в трансляторе указанный идентификатор, т. е. входил ли он в команду вида #define. Команда вида

#ifndef *идентификатор*

определяет, неопределенный ли в данный момент указанный идентификатор.

За любой из этих трех команд может следовать произвольное число строк текста, возможно, содержащих команду вида

#else

и заканчивающихся командой

#endif

Если проверяемое условие справедливо, то строки между #else и #endif игнорируются. Если проверяемое условие не выполняется, то игнорируются все строки между проверкой и командой #else, а если ее нет, то командой #endif.

Такие команды могут вкладываться одна в другую.

12.4. Управление строками

Для «удобства» других препроцессоров, которые могут формировать программу на языке Си, команда вида

#line *константа идентификатор*

заставляет транслятор при диагностике «поверить», что номер следующей строки входного текста такой, который указывает константа, а текущий файл именуется упомянутым идентификатором. Если идентификатор отсутствует, то ранее запомненное имя файла не изменяется.

13. НЕЯВНЫЕ ОПИСАНИЯ

Тип и класс памяти идентификаторов в описаниях задавать не всегда обязательно. Класс памяти очевиден из контекста во внешних определениях, в описаниях формальных параметров и у элементов записей. В лю-

бых описаниях внутри функции, если указан класс памяти, но нет типа, предполагается, что идентификатор типа int. Если есть тип, но нет класса памяти, предполагается, что он auto. Исключение из последнего правила делается только для функции, так как функции с такой спецификацией бессмысленны. (Трансляторы не способны транслировать рабочую программу в стек.) Если тип идентификатора — «функция, дающая ...», то она неявно описывается как extern.

В любых выражениях еще не описанный идентификатор, за которым следует (, контекстуально описывается как «функция, дающая int»).

14. ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ О ТИПАХ

В данном разделе приводится информация об операциях с объектами некоторых типов.

14.1. Записи и смеси

С записями и смесями можно проделывать только две вещи: именовать один из их элементов (с помощью операции .) и брать их адрес (с помощью унарной операции &). Другие операции, такие, как присваивание им или их или же передача их в качестве параметров будут фиксироваться как ошибочные. В будущем предполагается, что эти операции, но не обязательно другие, станут допустимыми.

В разд. 7.1 говорилось, что в прямом или косвенном указании на элемент (через . или —>) правое имя должно относиться к элементу поименованной (или определенной ссылкой) записи, определяемой левым именем. Дабы позволить выходить за рамки правил работы с типами, это ограничение трансляторами не особенно соблюдается. Фактически перед. (точкой) допускается любое адресное значение, и считается, что оно относится к записи, где есть элемент с «правым» именем. Аналогично требуется только, чтобы выражение перед —> было ссылкой или целым. Если это ссылка, то считается, что она указывает на запись, где есть элемент с «правым» именем. Если это целое значение, то оно воспринимается как абсолютный адрес в памяти машины, где хранится соответствующая запись.

Такие конструкции переносить с машины на машину нельзя.

14.2. Функции

С функциями можно совершать только два действия: обращаться к ним и брать их адрес. Если имя функции появляется не в позиции вызова функции, то формируется ссылка на функцию. Таким образом, передача одной функции другой может быть сделана так:

```
int f();
...
g(f);
```

Однако определение *g* должно быть похоже на такое:

```
g(funcp)
int (*funcp)();
{
    ...
    (*funcp)();
    ...
}
```

Обратите внимание, что в вызывающей функции f должна быть явно описана, так как в обращении g(f) следом за ней идет (.

14.3. Массивы, ссылки и индексирование

При каждом появлении идентификатора массива в выражении он преобразуется в ссылку на первый элемент этого массива. Из-за этого преобразования массивы не являются адресными значениями. По определению операция индексирования [ ] интерпретируется так, что E1[E2] идентично *((E1)+(E2)). С учетом правил преобразования для операции +, если E1 — массив, а E2 — целое число, то E1[E2] ссылается на E2-й элемент из E1. Поэтому, несмотря на кажущуюся асимметричность, индексирование — операция коммутативная.

Такое же правило действует и для многомерных массивов. Если E — n-мерный массив размером $i \times j \times \ldots \times k$, то при появлении E в выражении происходит преобразование к ссылке на $(n-1)$-мерный массив размером $j \times \ldots \times k$. Если операция * явно или неявно в результате индексирования применяется к такой ссылке, то результат указывает на $(n-1)$-мерный массив, и преобразуется в ссылку.

Рассмотрим, например:

```
int x[3][5];
```

Здесь x — массив целых размером 3×5. Если x появляется в выражении, то оно преобразуется в ссылку на первый из трех массивов, состоящих из 5 целых элементов. В выражении x[i], эквивалентном *(x+i), сначала, как описывалось, x преобразуется в ссылку, затем i преобразуется к типу x, сюда включается умножение i на размер объекта, на который указывает ссылка, в данном случае это размер объекта из пяти целых чисел. Результаты складываются, и косвенность приводит к массиву (из пяти целых чисел). И все снова преобразуется в ссылку на первое из этих чисел. Если есть еще другой индекс, то этот аргумент снова перерабатывают и в результате получают целое число.

Отсюда можно сделать вывод, что в Си массивы хранятся по строкам (быстрее всего изменяется последний индекс); кроме того, первый индекс описания служит лишь для определения размера памяти, необходимой для хранения массива, и не играет никакой роли в вычислениях, связанных с процессом индексации.

14.4. Явное преобразование ссылок

Некоторые преобразования, связанные с ссылками, допустимы, однако они зависят от реализации. Эти преобразования полностью задаются явными операциями преобразования типов (см. разд. 7.2 и 8.7).

Всякую ссылку можно преобразовать в значение любого из целых типов, достаточно большого, чтобы хранить такую ссылку. Будет ли это тип int или long, зависит от машины. Само отображение также зависит от машины; для тех, кто знаком со структурой адресации в машине, в этом нет ничего удивительного. Ниже приводятся подробности, связанные с некоторыми конкретными машинами.

Любой объект целого типа можно явно преобразовать в ссылку. Преобразование идет так, что целое, полученное из ссылки, всегда можно превратить в ту же ссылку, однако обратное преобразование зависит от машины.

Ссылку одного типа можно превратить в ссылку другого типа. Получившаяся ссылка может привести к «неприятностям» при ее использо-

вании, если исходная ссылка не указывала на объект, расположенный в памяти с соответствующим смещением. Гарантируется, что ссылка на объект данного размера может быть преобразована в ссылку на объект меньшего размера и обратно без изменений.

Например, программа распределения памяти могла бы воспринимать размер размещаемого объекта в байтах и возвращать ссылку на char; ее можно было бы использовать таким образом:

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

Программа alloc должна обеспечивать (не зависящим от машины способом) выдачу в качестве результата значения, которое можно преобразовать в ссылку на значение типа double, только в этом случае функцию можно переносить с машины на машину.

На машине PDP-11 ссылкам соответствуют 16-разрядные целые числа; они адресуют объект с точностью до байта. При размещении объектов типа char выравнивание проводить нет необходимости, но во всех других случаях нужно начинать объект с четного байта.

На машине Honeywell 6000 ссылкам соответствуют 36-разрядные целые числа: адрес слова расположен в левых 18 разрядах, а два разряда, необходимые для выделения в слове символов, примыкают к нему справа. Таким образом, ссылки на char измеряются в единицах, равных 2^{16} байтам, а все другие ссылки — в единицах, равных 2^{18} словам. Объекты типа double или составные значения должны размещаться с четных слов, т. е. для них адрес равен 0 mod 2^{19}.

Машины IBM 370 и Interdata 8/32 подобны друг другу. И там, и там адресация идет в байтах; элементарный объект должен быть выравнен до границы, кратной его длине, так что ссылка на значение типа short должна быть равна 0 mod 2, на int или float — 0 mod 4, на double — 0 mod 8. Составные объекты выравниваются до точной границы, требуемой любой из их компонент.

15. КОНСТАНТНЫЕ ВЫРАЖЕНИЯ

В некоторых местах в языке речь идет о выражениях, которые должны «сводиться» к константе: после слова case, в границах массивов и при инициации. В первых двух случаях выражение должно включать только целые константы, символьные константы и «функцию» sizeof. Для объединения их в выражения могут использоваться бинарные операции:

```
+ - * / % & | ^ < < > > = = ! = < > <=> =
```

или унарные операции

```
- ~
```

или «тройная» операция

```
? :
```

Для группирования можно использовать скобки, но нельзя обращаться к функциям.

При инициации допустима бóльшая свобода: кроме выражений из констант (о них речь шла выше) можно использовать унарную операцию &, применяемую ко внешним или статическим объектам и ко внешним или статическим массивам, индексированным константными выражениями. Унарная операция & может появляться и неявно, при вхождении массивов без индексов или функций. Основное правило — инициирую-

щее значение должно сводиться либо к константе, либо к адресу предварительно описанного внешнего или статического объекта плюс/минус постоянное значение (смещение).

16. ПЕРЕНОСИМОСТЬ

Некоторые части языка по своей природе зависят от машины. Приводимый ниже список возможных «трудных» мест, конечно, неполон, но основные проблемы здесь нашли свое отражение.

Чисто машинные (аппаратные) особенности типа размера слова, выполнения операций с плавающей запятой или целого деления на практике порождают не так уж много проблем. Другие особенности в разных реализациях отражаются по-разному. Некоторые из них, скажем размножение знака (преобразование «отрицательного» символа в отрицательное целое) и порядок расположения байтов в слове, представляют собой такие вещи, на которые должно обращать особое внимание. Большинство же других особенностей приводят лишь к весьма незначительным следствиям.

Число регистровых переменных, действительно помещающихся в регистры, изменяется от машины к машине, как и множество доступных для них типов. Как бы то ни было, трансляторы работают соответствующим для каждой машины образом, и излишние или неподобающие описания register игнорируются.

Затруднения возникают только там, где практикуется «сомнительное» программирование. Поэтому в высшей степени безответственно писать программы, зависящие от каких-либо случайных особенностей машины.

В языке не определен порядок вычисления аргументов функции. На PDP-11 они вычисляются справа налево, а на других машинах — слева направо. Не определяется также и порядок, при котором имеет место побочный эффект.

Так как символьные константы фактически есть объекты типа int, то допускаются многосимвольные константы. Однако реализация их зависит от машины, так как она определяется порядком записи символов в слово, а он на разных машинах разный.

На PDP-11 поля в слове или символы в целом заполняются справа налево, а на других машинах — слева направо. В изолированных программах, где нет «трюкачества» с типами, например преобразования ссылки на целое в ссылку на символ и последующего обращения в память, эти особенности машин незаметны, но их следует учитывать, если речь идет о согласованных смещениях в памяти, заданных извне.

Язык, воспринимаемый различными трансляторами, различается в незначительных деталях. Из них стоит лишь упомянуть, что транслятор на PDP-11 не инициирует записи, содержащие разрядные поля, и в некоторых контекстах не воспринимает некоторые операции присваивания, там, где используется значение от присваивания.

17. АНАХРОНИЗМЫ

Так как Си непрерывно развивается, то в старых программах можно обнаружить устаревшие конструкции. Хотя большинство версий транслятора сейчас поддерживает такие анахронизмы, в конце концов они исчезнут; правда, останутся проблемы, связанные с переносом.

Вместо конструкции *op*= для операции присваивания в ранних версиях языка использовалась конструкция =*op*. Она приводила к двусмысленности в таких, например, местах, как

```
x=-1
```

так как = и — стоят рядом, то речь идет об уменьшении x, но можно легко представить, что переменной x присваивается —1.

Был изменен и синтаксис инициации: раньше не было знака равенства перед начальным значением, поэтому вместо

```
int  x    = 1;
```

писали

```
int  x    1;
```

изменение было сделано из-за того, что инициация

```
int  f    (1+2)
```

выглядела почти как описание функции, что «сбивало» транслятор.

18. СИНТАКСИС ЯЗЫКА СИ

Приведенный ниже синтаксис языка предназначается для облегчения понимания языка и не представляет собой точное его определение.

18.1. ВЫРАЖЕНИЯ

Основные выражения таковы:

выражение:
- *первичное*
- * *выражение*
- & *выражение*
- — *выражение*
- ! *выражение*
- ~ *выражение*
- ++ *адрес*
- — — *адрес*
- *адрес* ++
- *адрес* — —
- sizeof *выражение*
- (*имя типа*) *выражение*
- *выражение биноп выражение*
- *выражение* ? *выражение* : *выражение*
- *адрес присоп выражение*
- *выражение* , *выражение*

первичное:
- *идентификатор*
- *константа*
- *строка*
- (*выражение*)
- *первичное* (*список выражений*$_{opt}$)
- *первичное* | *выражение* |
- *адрес. идентификатор*
- *первичное* —> *идентификатор*

адрес:
- *идентификатор*
- *первичное* | *выражение* |
- *адрес* . *идентификатор*
- *первичное* —> *идентификатор*
- * *выражение*
- (*адрес*)

Операции первичных выражений

( ) [ ] . —>

имеют высший приоритет и выполняются слева направо. Унарные операции

* & — ! ~ + + — — sizeof (*имя типа*)

имеют приоритет, меньший, чем у первичных операций, но больший, чем у любой из бинарных операций. Выполняются они справа налево. Бинарные операции и операция условия выполняются слева направо, а приоритет их соответствует порядку перечисления в нижеприведенном определении:

бинор:

```
*     /     %
+     -
>>    <<
<     >     <=     >=
==    !=
&
^
|
&&
||
?:
```

Все операции присваивания имеют один приоритет и выполняются справа налево:

присоп:

$= + = - = * = / = \% = >> = << = \& = \hat{} = | =$

Операция «запятая» имеет самый низкий приоритет и выполняется слева направо.

18.2. Описания

Описание:
спецификации описания список описателей с инициатором

спецификации описания
спецификация типа спецификации описания$_{opt}$
спецификация памяти спецификации описания$_{opt}$

спецификация памяти:
auto
static
extern
register
typedef

спецификация типа:
char
short
int
long
unsigned
float
double
спецификация записи или смеси
имя определенного типа

список описателей с инициатором:
описатель с инициатором
описатель с инициатором , список описателей с инициатором

описатель с инициатором:
описатель инициатор$_{opt}$

описатель:
идентификатор
(*описатель*)
* *описатель*
описатель ( )
описатель [*константное выражение*$_{opt}$]

спецификация записи или смеси:
struct { *список описаний записи* }
struct *идентификатор* { *список описаний записи* }
struct *идентификатор*
union { *список описаний записи* }
union *идентификатор* { *список описаний записи* }
union *идентификатор*

список описаний записи:
описание записи
описание записи список описаний записи

описание записи:
спецификация типа список описателей записи:

список описателей записи:
описатель записи
описатель записи , список описателей записи

описатель записи:
описатель
описатель : константное выражение
: константное выражение

инициатор:
= *выражение*
= { *список инициаторов* }
= { *список инициаторов* , }

список инициаторов:
выражение
список инициаторов , список инициаторов
{ *список инициаторов* }

имя типа:
спецификация типа абстрактный описатель

абстрактный описатель:
пусто
(*абстрактный описатель*)
* *абстрактный описатель*
абстрактный описатель ( )
абстрактный описатель [*константное выражение*$_{opt}$]

имя описанного типа:
идентификатор

18.3. Операторы

составной оператор:
{ *список описаний*$_{opt}$ *список операторов*$_{opt}$ }

список описаний:
описание
описание список описаний

список операторов:
оператор
оператор список операторов

оператор:
составной оператор
выражение ;
if (*выражение*) *оператор*
if (*выражение*) *оператор* else *оператор*
while (*выражение*) *оператор*
do *оператор* while (*выражение*)
for (*выражение-1*$_{opt}$; *выражение-2*$_{opt}$; *выражение-3*$_{opt}$) *оператор*
switch (*выражение*) *оператор*
case *константное выражение* : *оператор*
default : *оператор*
break ;
continue ;
return ;
return *выражение* ;
goto *идентификатор* ;
идентификатор : *оператор*
;

18.4. Внешние определения

программа:
внешнее определение
внешнее определение программа

внешнее определение:
определение функции
определение данных

определение функции:
спецификация типа$_{opt}$ *описатель функции тело функции*

описатель функции:
описатель (*список параметров*$_{opt}$)

список параметров:
идентификатор
идентификатор , *список параметров*

тело функции:
список описаний типа сама функция

сама функция:
{ *список описаний*$_{opt}$ *список операторов* }

определение данных:
extern$_{opt}$ *спецификация типа*$_{opt}$ *список описателей с инициатором*$_{opt}$
static$_{opt}$ *спецификация типа*$_{opt}$ *список описателей с инициатором*$_{opt}$

18.5. Препроцессор

#define *идентификатор подстановка*
#define *идентификатор* (*идентификатор* , ... , *идентификатор*) *подстановка*
#undef *идентификатор*
#include *«имя файла»*
#include <*имя файла*>
#if *константное выражение*
#ifdef *идентификатор*
#ifndef *идентификатор*
#else
#endif
#line *константа идентификатор*

ЗАДАЧИ ПО ЯЗЫКУ СИ

ПРЕДИСЛОВИЕ

Если судить по объему справочного руководства, Си не очень «большой» язык. Это объясняется немногими ограничивающими правилами, налагаемыми на язык. Пользователи Си быстро сумеют оценить его изящность, проистекающую из ясных принципов построения языка. Такая изящность может быть излишне таинственной для начинающего программиста. Отсутствие ограничений означает, что программы на Си могут писаться, и пишутся, с использованием развитых форм выражений, что может показаться новичку ошибочным. Способность конструкций языка Си образовывать единое целое позволяет быстро находить ясные и компактные средства для решения программистских задач.

Процесс изучения языка программирования можно представить как последовательное прохождение трех этапов (каждый этап, несомненно, может повторяться многократно). Первый этап состоит в освоении синтаксиса языка, по крайней мере настолько, что транслятор перестает «ругаться», обнаружив в вашей программе бессмысленные конструкции. Второй этап заключается в осмыслении правильно построенных транслятором конструкций языка. На третьем этапе вырабатывается определенный стиль программирования, соответствующий духу языка, т. е. умение писать ясные, краткие и правильные программы.

Головоломки в этой книге предназначены для того, чтобы помочь читателю на втором этапе обучения языку. Они бросают вызов искусству владения читателем основными правилами языка, заводят его в «редко посещаемые закоулки» языка, подводят к граничным условиям и знакомят с немногими откровенными ловушками. (Конечно, Си, как всякий реальный язык программирования, имеет свою долю неясностей, которая познается на опыте.)

Примеры этой книги не следует рассматривать, как образцы хорошего программирования. Но то, что делает программу плохой, может головоломку сделать интересной, это скажем:

двусмысленность выражений, требующая для их интерпретации обращения к описанию языка;

сложность структуры: структуру данных и структуру программы нелегко удержать в голове;

неочевидность конструкций, возникающая при использовании их нестандартным способом.

Язык Си до сих пор остается развивающимся языком, поэтому некоторые конструкции, приведенные в книге, могут быть у вас не реализованы, и, наоборот, некоторые реализованные конструкции могут здесь не исследоваться. Это зависит от выбранного вами «локального» транслятора. Но поскольку, к счастью, развитие языка происходит

единообразно, маловероятно, чтобы используемый вами транслятор какие-то конструкции реализовал не так, как здесь описано.

Как пользоваться этой книгой?

Ее следует рассматривать как дополнительное пособие к книге Б. Кернигана и Д. Ритчи «Язык программирования Си». Задачник делится на главы, и каждая глава посвящена какой-нибудь одной теме. В главе содержатся программы, отражающие различные аспекты этой темы. В программы вкраплены операторы печати, и основная цель приводимых головоломок состоит в том, чтобы выяснить, что напечатает каждая программа. Все программы независимы друг от друга, хотя встречающиеся в тексте позднее подразумевают, что вам понятны свойства языка, иллюстрируемые предшествующими программами.

Результаты, выдаваемые на печать каждой программой, приводятся вслед за текстом программы. Каждая программа выполнялась в виде, полностью совпадающем с приведенным в тексте, на машинах фирмы DEC PDP-11/70 и VAX-11/780 под управлением операционной системы UNIX. Для тех немногих случаев, когда результаты, полученные на этих двух машинах различны, приводятся оба результата.

Большая часть книги посвящена подробному разбору решения задачи, т. е. пояснению процесса выполнения программы. Многие решения задач сопровождаются предостережениями и советами, касающимися программирования на Си.

Порядок работы с задачником может быть таким:

познакомиться в книге Б. Кернигана и Д. Ритчи с соответствующей темой;

для каждой программы из главы задачника, посвященной этой теме:

мысленно выполнить программу,

сравнить ваши результаты с приводимыми на следующей странице задачника,

прочесть разбор решения задачи.

ЗАДАЧИ

Глава 1. Операции

Программы на языке Си строятся из операторов, операторы — из выражений и выражения — из операндов и операций. Си имеет необычайно богатое множество операций; за подтверждением вы можете обратиться к сводной таблице операций в приложении 2. Следствием разнообразия операций является то, что правила, определяющие порядок применения операций к операндам, играют основную роль в понимании выражений языка. Порядок выполнения операций и их приоритеты приведены в таблице приложения 1. Воспользуйтесь этой таблицей для решения задач данной главы.

Операции 1: Основные арифметические операции

Что напечатает следующая программа?

```
main()
{
      int x;

      x = - 3 + 4 * 5 - 6; printf("%d\n",x);          (Операции 1.1)
      x = 3 + 4 % 5 - 6; printf("%d\n",x);            (Операции 1.2)
      x = - 3 * 4 % - 6 / 5; printf("%d\n",x);        (Операции 1.3)
      x = ( 7 + 6 ) % 5 / 2; printf("%d\n",x);        (Операции 1.4)
}
```

Результаты:

```
11      (Операции 1.1)
1       (Операции 1.2)
0       (Операции 1.3)
1       (Операции 1.4)
```

Пояснения начинаются на с. 221.

Операции 2: Операции присваивания

Что напечатает следующая программа?

```
#define PRINTX printf("%d\n",x)

main()
{
      int x=2, y, z;

      x *= 3 + 2; PRINTX;              (Операции 2.1)
      x *= y = z = 4; PRINTX;          (Операции 2.2)
      x = y == z; PRINTX;              (Операции 2.3)
      x == ( y = z ); PRINTX;          (Операции 2.4)
}
```

Результаты:

```
10      (Операции 2.1)
40      (Операции 2.2)
1       (Операции 2.3)
1       (Операции 2.4)
```

Пояснения начинаются на с. 223.

Операции 3: Логические операции и операции увеличения

Что напечатает следующая программа?

```
#define PRINT(int) printf("%d\n",int)

main()
{
      int x, y, z;

      x = 2; y = 1; z = 0;
      x = x && y || z; PRINT(x);          (Операции 3.1)
      PRINT( x || ! y && z );             (Операции 3.2)
```

```
    x = y = 1;
    z = x ++ - 1; PRINT(x); PRINT(z);          (Операции 3.3)
    z += - x ++ + ++ y; PRINT(x); PRINT(z);    (Операции 3.4)
    z = x / ++ x; PRINT(z);                    (Операции 3.5)
}
```

Результаты:

```
1       (Операции 3.1)
1       (Операции 3.2)
2       (Операции 3.3)
0
3       (Операции 3.4)
0       (Операции 3.5)
?
```

Пояснения начинаются на с. 224.

Операции 4: Поразрядные операции

Что напечатает следующая программа?

```
#define PRINT(int) printf("int = %d\n",int)

main()
{
    int x, y, z;

    x = 03; y = 02; z = 01;
    PRINT( x | y & z );          (Операции 4.1)
    PRINT( x | y & ~ z );        (Операции 4.2)
    PRINT( x ^ y & ~ z );        (Операции 4.3)
    PRINT( x & y && z );         (Операции 4.4)

    x = 1; y = -1;
    PRINT( ! x | x );            (Операции 4.5)
    PRINT( ~ x | x );            (Операции 4.6)
    PRINT( x ^ x );              (Операции 4.7)
    x <<= 3; PRINT(x);           (Операции 4.8)
    y <<= 3; PRINT(y);           (Операции 4.9)
    y >>= 3; PRINT(y);           (Операции 4.10)
}
```

Результаты:

```
x | y & z  = 3          (Операции 4.1)
x | y & ~ z  = 3        (Операции 4.2)
x ^ y & ~ z  = 1        (Операции 4.3)
x & y && z  = 1         (Операции 4.4)
! x | x  = 1            (Операции 4.5)
~ x | x  = -1           (Операции 4.6)
x ^ x  = 0              (Операции 4.7)
x = 8                   (Операции 4.8)
y = -8                  (Операции 4.9)
y = ?                   (Операции 4.10)
```

Пояснения начинаются на с. 227.

Операции 5: Отношения и условия

Что напечатает следующая программа?

```
#define PRINT(int) printf("int = %d\n",int)

main()
{
     int x=1, y=1, z=1;

     x += y += z;
     PRINT( x < y ? y : x );                    (Операции 5.1)

     PRINT( x < y ? x ++ : y ++ );
     PRINT(x); PRINT(y);                        (Операции 5.2)

     PRINT( z += x < y ? x ++ : y ++ );
     PRINT(y); PRINT(z);                        (Операции 5.3)


     x=3; y=z=4;
     PRINT( (z >= y >= x) ? 1 : 0);             (Операции 5.4)
     PRINT( z >= y && y >= x );                 (Операции 5.5)
}
```

Результаты:

```
x < y ? y : x  = 3                  (Операции 5.1)
x < y ? x ++ : y ++  = 2            (Операции 5.2)
x = 3
y = 3
z += x < y ? x ++ : y ++  = 4       (Операции 5.3)
y = 4
z = 4
(z >= y >= x) ? 1 : 0 = 0           (Операции 5.4)
z >= y && y >= x  = 1               (Операции 5.5)
```

Пояснения начинаются на с. 230.

Операции 6: Выполнение операций и их приоритеты

Что напечатает следующая программа?

```
#define PRINT3 (x, y, z) printf ("x = %d\ty = %d\tz = %d\n", x, y, z);
main()
{
        int x, y, z;

        x = y = z = 1;
        ++x || ++y && ++z; PRINT3(x,y,z);      (Операции 6.1)

        x = y = z = 1;
        ++x && ++y || ++z; PRINT3(x,y,z);      (Операции 6.2)

        x = y = z = 1;
        ++x && ++y && ++z; PRINT3(x,y,z);      (Операции 6.3)

        x = y = z = -1;
        ++x && ++y || ++z; PRINT3(x,y,z);      (Операции 6.4)

        x = y = z = -1;
        ++x || ++y && ++z; PRINT3(x,y,z);      (Операции 6.5)

        x = y = z = -1;
        ++x && ++y && ++z; PRINT3(x,y,z);      (Операции 6.6)
}
```

Результаты:

```
x=2   y=1   z=1      (Операции 6.1)
x=2   y=2   z=1      (Операции 6.2)
x=2   y=2   z=2      (Операции 6.3)
x=0   y=-1  z=0      (Операции 6.4)
x=0   y=0   z=-1     (Операции 6.5)
x=0   y=-1  z=-1     (Операции 6.6)
```

Пояснения начинаются на с. 231.

Глава 2. Основные типы

Язык Си имеет относительно немного основных типов. В выражении свободно могут перемешиваться операнды разных типов. Тип результата выражения будет определяться некоторыми простыми правилами преобразования основных типов (иерархией простых типов). Эта иерархия приведена на схеме приложения 4.

Для решения некоторых задач данного раздела необходимо знать целые значения, соответствующие определенным символам. В таблице

приложения 3 приведены такие значения для символов в кодировке ASCII. В нескольких задачах ответы получаются различными для машин VAX и PDP-11, поэтому приведены оба ответа.

Основные типы 1: Символ, строка и целый тип

Что напечатает следующая программа?

```
#include <stdio.h>

#define PRINT(format,x) printf("x = %format\n",x)

int  integer = 5;
char character = '5';
char *string = "5";

main()
{
     PRINT(d,string); PRINT(d,character); PRINT(d,integer);
     PRINT(s,string); PRINT(c,character); PRINT(c,integer=53);
     PRINT(d,( '5'>5 ));                          (Основные типы 1.1)

     {
          int sx = -8;
          unsigned ux = -8;

          PRINT(o,sx); PRINT(o,ux);
          PRINT(o, sx>>3 ); PRINT(o, ux>>3 );
          PRINT(d, sx>>3 ); PRINT(d, ux>>3 );     (Основные типы 1.2)
     }
}
```

Результаты:

```
string = an address                          (Основные типы 1.1)
character = 53
integer = 5
string = 5
character = 5
integer=53 = 5
( '5'>5 ) = 1
sx = 177770                                  (Основные типы 1.2-PDP 11)
ux = 177770
 sx>>3  = 177777 or 017777
 ux>>3  = 17777
 sx>>3  = -1 or 8191
 ux>>3  = 8191

sx = 37777777770                             (Основные типы 1.2-VAX)
ux = 37777777770
```

```
sx>>3   = 37777777777 or 03777777777
ux>>3   = 3777777777
sx>>3   = -1 or 536870911
ux>>3   = 536870911
```

Пояснения начинаются на с. 233.

Основные типы 2: Приведение целых чисел и чисел с плавающей точкой

Что напечатает следующая программа?

```
#include <stdio.h>

#define PR(x)  printf("x = %.8g\t",(double)x)
#define NL putchar('\n')
#define PRINT4(x1,x2,x3,x4)  PR(x1); PR(x2); PR(x3); PR(x4); NL

main()
{
     double d;
     float f;
     long l;
     int i;

     i = l = f = d = 100/3; PRINT4(i,l,f,d);      (Основные типы 2.1)
     d = f = l = i = 100/3; PRINT4(i,l,f,d);      (Основные типы 2.2)
     i = l = f = d = 100/3.; PRINT4(i,l,f,d);     (Основные типы 2.3)
     d = f = l = i = (double)100/3;
     PRINT4(i,l,f,d);                             (Основные типы 2.4)


     i = l = f = d = (double)(100000/3);
     PRINT4(i,l,f,d);                             (Основные типы 2.5)

     d = f = l = i = 100000/3; PRINT4(i,l,f,d);   (Основные типы 2.6)
}
```

Результаты:

```
i = 33   l = 33   f = 33   d = 33                                  (Основные типы 2.1)
i = 33   l = 33   f = 33   d = 33                                  (Основные типы 2.2)
i = 33   l = 33   f = 33.333332   d = 33.333333                    (Основные типы 2.3)
i = 33   l = 33   f = 33   d = 33                                  (Основные типы 2.4)
i = overflow   l = 33333   f = 33333  d = 33333  (Основные типы 2.5-PDP 11)
i = overflow   l = -32203  f = -32203 d = -32203 (Основные типы 2.6-PDP 11)

i = 33333    l = 33333    f = 33333  d = 33333    (Основные типы 2.5-VAX)
i = 33333    l = 33333    f = 33333  d = 33333    (Основные типы 2.6-VAX)
```

Пояснения начинаются на с. 234.

Основные типы 3: Еще о приведении типов

Что напечатает следующая программа?

```
#include <stdio.h>

#define PR(x)  printf("x = %g\t",(double)(x))
#define NL putchar('\n')
#define PRINT1(x1)  PR(x1); NL
#define PRINT2(x1,x2)  PR(x1); PRINT1(x2)

main()
{
     double d=3.2, x;
     int i=2, y;

     x = (y=d/i)*2; PRINT2(x,y);                          (Основные типы 3.1)
     y = (x=d/i)*2; PRINT2(x,y);                          (Основные типы 3.2)

     y = d * (x=2.5/d); PRINT1(y);                        (Основные типы 3.3)
     x = d * (y = ((int)2.9+1.1)/d); PRINT2(x,y);         (Основные типы 3.4)
```

Результаты:

```
x = 2     y = 1      (Основные типы 3.1)
x = 1.6   y = 3      (Основные типы 3.2)
y = 2                (Основные типы 3.3)
x = 0     y = 0      (Основные типы 3.4)
```

Пояснения начинаются на с. 287.

Глава 3. Включение файлов

Каждая из последующих программ начинается с оператора препроцессора:

```
#include "defs.h"
```

Препроцессор заменяет приведенную выше строку на содержимое файла defs.h, что делает определения, содержащиеся в файле defs.h, доступными для использования. Ниже приводится содержимое defs.h:

```
#include <stdio.h>

#define PR(format,value) printf("value = %format\t",(value))
#define NL putchar('\n')

#define PRINT1(f,x1)  PR(f,x1), NL
#define PRINT2(f,x1,x2)  PR(f,x1), PRINT1(f,x2)
#define PRINT3(f,x1,x2,x3)  PR(f,x1), PRINT2(f,x2,x3)
#define PRINT4(f,x1,x2,x3,x4)  PR(f,x1), PRINT3(f,x2,x3,x4)
```

Файл defs.h, в свою очередь, опять начинается с оператора include, требующего поместить в программу содержимое файла stdio.h, что необходимо для пользования библиотекой стандартных программ Си. Оставшаяся часть файла defs.h содержит макроопределения для печати. Например, чтобы напечатать *5*, как целое десятичное число, нужно обратиться к PRINT1 следующим образом:

```
PRINT1(d,5)
```

это обращение заменяется на

```
PR(d,5), NL
```

что дальше заменяется на

```
printf("5 = %d\t",(5)), putchar('\n').
```

Макроопределение PRINT показывает некоторую особенность препроцессора, которая часто приводит к неприятностям. Если определяемое имя встречается внутри строки, т. е. между двойными кавычками, то оно не заменяется. Однако аргументы макроопределения будут заменяться всюду, где бы они ни встретились, даже внутри строки. Обратите внимание, что макроопределение PR использует это свойство препроцессора. Для более подробного описания макроопределений смотрите главу, посвященную препроцессору.

Глава 4. Управление

В языке Си, как и в большинстве языков программирования, есть операторы управления для условной, выборочной и циклической обработки информации. Чтобы разобраться в задачах этой главы, нужно уметь определять область действия каждого из таких операторов. В хорошо отредактированной программе область действия операторов легко определить, так как начало и конец каждого оператора выделяется соответствующим числом пробелов. Чтение же плохо отредактированной программы мучительно и провоцирует ошибки; следующие задачи убедят вас в этом.

Управление 1: Условный оператор

Что напечатает следующая программа?

```
#include "defs.h"

main()
{
     int x, y=1, z;

     if( y!=0 ) x=5;
     PRINT1(d,x);                    (Управление 1.1)

     if( y==0 ) x=3;
     else x=5;
     PRINT1(d,x);                    (Управление 1.2)
```

```
x=1;
if( y<0 ) if( y>0 ) x=3;
else x=5;
PRINT1(d,x);                    (Управление 1.3)

if( z=y<0 ) x=3;
else if( y==0 ) x=5;
else x=7;
PRINT2(d,x,z);                  (Управление 1.4)

if( z=(y==0) ) x=5; x=3;
PRINT2(d,x,z);                  (Управление 1.5)

if( x=z=y ); x=3;
PRINT2(d,x,z);                  (Управление 1.6)
```

Результаты:

```
x = 5                   (Управление 1.1)
x = 5                   (Управление 1.2)
x = 1                   (Управление 1.3)
x = 7    z = 0          (Управление 1.4)
x = 3    z = 0          (Управление 1.5)
x = 3    z = 1          (Управление 1.6)
```

Пояснения начинаются на с. 238.

Управление 2: Операторы while и for

Что напечатает следующая программа?

```
#include "defs.h"

main()
{
      int x, y, z;

      x=y=0;
      while( y<10 ) ++y; x += y;
      PRINT2(d,x,y);                    (Управление 2.1)

      x=y=0;
      while( y<10 ) x += ++y;
      PRINT2(d,x,y);                    (Управление 2.2)

      y=1;
      while( y<10 ) {
      }     x = y++; z = ++y;
      PRINT3(d,x,y,z);                  (Управление 2.3)
```

```
for( y=1; y<10; y++ ) x=y;
PRINT2(d,x,y);                          (Управление 2.4)

for( y=1; (x=y)<10; y++ ) ;
PRINT2(d,x,y);                          (Управление 2.5)

for( x=0,y=1000; y>1; x++,y/=10 )
     PRINT2(d,x,y);                     (Управление 2.6)
}
```

Результаты:

```
x = 10   y = 10              (Управление 2.1)
x = 55   y = 10              (Управление 2.2)
x = 9    y = 11   z = 11     (Управление 2.3)
x = 9    y = 10              (Управление 2.4)
x = 10   y = 10              (Управление 2.5)
x = 0    y = 1000            (Управление 2.6)
x = 1    y = 100
x = 2    y = 10
```

Пояснения начинаются на с. 240.

Управление 3: Вложенность операторов

Что напечатает следующая программа?

```
#include "defs.h"

#define ENUF 3
#define EOS '\0'
#define NEXT(i) input[i++]
#define FALSE 0
#define TRUE 1

char input[]="PI=3.14159, approximately";

main()
{
     char c;
     int done, high, i, in, low;

     i=low=in=high=0;
     while( c=NEXT(i) != EOS )
          if( c<'0' ) low++;
          else if( c>'9' ) high++;
          else in++;
     PRINT3(d,low,in,high);                    (Управление 3.1)

     i=low=in=high=0; done=FALSE;
     while( (c=NEXT(i))!=EOS && !done )
          if( c<'0' ) low++;
          else if( c>'9' ) high++;
```

```
        else in++;
        if( low>=ENUF || high>=ENUF || in>=ENUF )
            done = TRUE;
    PRINT3(d,low,in,high);                          (Управление 3.2)

    i=low=in=high=0; done=FALSE;
    while( (c=NEXT(i))!=EOS && !done )
        if( c<'0' ) done = (++low==ENUF);
        else if( c>'9' ) done = (++high==ENUF);
        else done = (++in==ENUF);
    PRINT3(d,low,in,high);                          (Управление 3.3)
}
```

Результаты:

```
low = 25    in = 0      high = 0      (Управление 3.1)
low = 3     in = 6      high = 16     (Управление 3.2)
low = 0     in = 0      high = 3      (Управление 3.3)
```

Пояснения начинаются на с. 243.

Управление 4: Переключатели и операторы разрыва и продолжения

Что напечатает следующая программа?

```
#include "defs.h"

char input[] = "SSSWILTECH1\1\11W\1WALLMP1";

main()
{
    int i, c;

    for( i=2; (c=input[i])!='\0'; i++) {
        switch(c) {
        case 'a': putchar('i'); continue;
        case '1': break;
        case 1: while( (c=input[++i])!='\1' && c!='\0' ) ;
        case 9: putchar('S');
        case 'E': case 'L': continue;
        default: putchar(c); continue;
        }
        putchar(' ');
    }
    putchar('\n');                                  (Управление 4.1)
}
```

Результаты:

```
SWITCH SWAMP        (Управление 4.1)
```

Пояснения начинаются на с. 244.

О стиле программирования писалось много; и о том, каких конструкций следует избегать, и о том, какие конструкции следует имитировать. Из кажущихся противоречивыми советов можно было бы вынести поверхностное заключение, что хороший стиль программирования в основном дело вкуса. Более же разумный вывод говорит, что хороший стиль в программировании, как и в любом деле, прежде всего дело здравого смысла. Поэтому, хотя и существует много рекомендаций в отношении хорошего стиля программирования, только немногие из них безоговорочно применимы.

Имейте все это в виду, когда будете разбирать задачи, демонстрирующие распространенные огрехи в стиле программирования. Приводимые решения в отличие от решений задач из предыдущих глав будут только одними из возможных. Если и есть универсальная рекомендация по стилю программирования, то она заключается в осознании двух необходимых шагов для построения хорошо составленной программы:

четко сформулируйте основную идею программы;

составьте программу, соответствующую по структуре этой идее (ее выражению).

Стиль программирования 1: Составьте правильно условие

С помощью перестройки улучшите структуру следующих фрагментов программ:

```
while(A) {
     if(B) continue;
     C;
}
```

(Стиль программирования 1.1)

```
do {
     if(!A) continue;
     else B;
     C;
} while(A);
```

(Стиль программирования 1.2)

```
if(A)
     if(B)
          if(C) D;
          else;
     else;
else
     if(B)
          if(C) E;
          else F;
     else;
```

(Стиль программирования 1.3)

```
while( (c=getchar())!='\n' ) {
      if( c==' ' ) continue;
      if( c=='\t' ) continue;
      if( c<'0' ) return(OTHER);
      if( c<='9' ) return(DIGIT);
      if( c<'a' ) return(OTHER);
      if( c<='z' ) return(ALPHA);
      return(OTHER);
} return(EOL);
```

(Стиль программирования 1.4)

Пояснения начинаются на с. 246.

Стиль программирования 2: Выберите подходящую конструкцию

С помощью перестройки улучшите структуру следующих фрагментов программ:

```
done=i=0;
while( i<MAXI && !done ) {
      if( (x/=2)>1 ) { i++; continue; }
      done++;
}
```

(Стиль программирования 2.1)

```
{
      if(A) { B; return; }
      if(C) { D; return; }
      if(E) { F; return; }
      G; return;
}
```

(Стиль программирования 2.2)

```
plusflg=zeroflg=negflg=0;
if( a>0 ) ++plusflg;
if( a==0 ) ++zeroflg;
else if( !plusflg ) ++negflg;
```

(Стиль программирования 2.3)

```
i=0;
while((c=getchar())!=EOF){
if(c!='\n'&&c!='\t'){s[i++]=c;continue;}
if(c=='\n')break;
if(c=='\t')c=' ';
s[i++]=c;}
```

(Стиль программирования 2.4)

```
if( x!=0 )
      if( j>k ) y=j/x;
      else y=k/x;
else
      if( j>k ) y=j/NEARZERO;
      else y=k/NEARZERO;
```

(Стиль программирования 2.5)

Пояснения начинаются на с. 247.

С каждой переменной языка Си связаны два основополагающих свойства — тип и класс памяти. Типы переменных обсуждались в предыдущей главе.

Класс памяти определяет для каждой переменной область ее действия и время существования. Областью действия переменной называется та часть программы, где эта переменная известна. Время существования переменной — период в ходе выполнения программы, в течение которого переменная имеет некоторое значение. Области действия переменной и времени ее существования ограничены блоками, функциями и файлами.

Классы памяти 1: Блоки

Что напечатает следующая программа?

```
#include "defs.h"

int i=0;

main()
{
      auto int i=1;
      PRINT1(d,i);
      {
            int i=2;
            PRINT1(d,i);
            {
                  i += 1;
                  PRINT1(d,i);
            }
            PRINT1(d,i);
      }
      PRINT1(d,i);                    (Классы памяти 1.1)
}
```

Результаты:

```
i = 1          (Классы памяти 1.1)
i = 2
i = 3
i = 3
i = 1
```

Пояснения начинаются на с. 250.

Классы памяти 2: Функции

Что напечатает следующая программа?

```
#include "defs.h"

#define LOW 0
#define HIGH 5
#define CHANGE 2

int i=LOW;

main()
{
     auto int i=HIGH;
     reset( i/2 ); PRINT1(d,i);
     reset( i=i/2 ); PRINT1(d,i);
     i = reset( i/2 ); PRINT1(d,i);

     workover(i); PRINT1(d,i);        (Классы памяти 2.1)
}

workover(i)
int i;
{
     i = (i%i) * ((i*i)/(2*i) + 4);
     PRINT1(d,i);
     return(i);
}

int reset(i)
int i;
{
     i = i<=CHANGE ? HIGH : LOW;
     return(i);
}
```

Результаты:

```
i = 5          (Классы памяти 2.1)
i = 2
i = 5
i = 0
i = 5
```

Пояснения начинаются на с. 251.

Классы памяти 3: Снова функции

Что напечатает следующая программа?

```
#include "defs.h"
int i=1;

main()
{
     auto int i, j;
     i = reset();
     for( j=1; j<=3; j++ ) {
          PRINT2(d,i,j);
          PRINT1(d,next(i));
          PRINT1(d,last(i));
          PRINT1(d,new(i+j));          (Классы памяти 3.1)
     }
}

int reset()
{
     return(i);
}

int next(j)
int j;
{
     return( j=i++ );
}

int last(j)
int j;
{
     static int i=10;
     return( j=i-- );
}

int new(i)
int i;
{
     auto int j=10;
     return( i=j+=i );
}
```

Результаты:

```
i = 1     j = 1      (Классы памяти 3.1)
next(i) = 1
last(i) = 10
new(i+j) = 12
i = 1     j = 2
next(i) = 2
```

```
last(i) = 9
new(i+j) = 13
i = 1     j = 3
next(i) = 3
last(i) = 8
new(i+j) = 14
```

Пояснения начинаются на с. 252.

Классы памяти 4: Файлы

Что напечатает следующая программа?

```
#include "defs.h"

int i=1;

main()
{
     auto int i, j;

     i = reset();
     for( j=1; j<=3; j++ ) {
          PRINT2(d,i,j);
          PRINT1(d,next(i));
          PRINT1(d,last(i));
          PRINT1(d,new(i+j));
     }
}
```

(Классы памяти 4.1)

В одном файле

```
static int i=10;

int next()
{
     return( i+=1 );
}

int last()
{
     return( i-=1 );
}

int new(i)
int i;
{
     static int j=5;
     return( i=j+=i );
}
```

В еще одном файле

```
extern int i;

reset()
{
   return(i);
}
```

Результаты:

```
i = 1     j = 1    (Классы памяти 4.1)
next(i) = 11
last(i) = 10
new(i+j) = 7
i = 1     j = 2
next(i) = 11
last(i) = 10
new(i+j) = 10
i = 1     j = 3
next(i) = 11
last(i) = 10
new(i+j) = 14
```

Пояснения начинаются на с. 254.

Глава 7. Ссылки и массивы

В течение долгого времени программисты всячески поносили ссылки, и в работах, посвященных стилю программирования, к ним относятся враждебно. В частности, применение ссылок критикуется из-за того, что в силу их природы невозможно определить, на что указывает в данный момент ссылка, если не возвращаться к тому месту, где ссылке в последний раз было присвоено значение. Это усложняет программу и делает доказательство ее правильности более трудным.

Язык Си не только не ограничивает использование ссылок, но и делает их применение естественным. Как показывают задачи, ссылки и массивы тесно связаны. Любую программу, использующую индексацию массива, можно переписать с помощью ссылок. Все предостережения относительно некорректного использования ссылок относятся к языку Си точно так же, как и ко многим другим языкам программирования.

Ссылки и массивы 1: Простые ссылки и массивы

Что напечатает следующая программа?

```
#include "defs.h"

int a[]={0,1,2,3,4};

main()
{
      int i, *p;

      for( i=0; i<=4; i++ ) PR(d,a[i]);          (Ссылки и массивы 1.1)
      NL;
      for( p= &a[0]; p<=&a[4]; p++ )
            PR(d,*p);                            (Ссылки и массивы 1.2)
      NL; NL;
```

```
    for( p= &a[0],i=1; i<=5; i++ )
         PR(d,p[i]);                                    (Ссылки и массивы 1.3)
    NL;
    for( p=a,i=0; p+i<=a+4; p++,i++ )
         PR(d,*(p+i));                                  (Ссылки и массивы 1.4)
    NL; NL;

    for( p=a+4; p>=a; p-- ) PR(d,*p);                   (Ссылки и массивы 1.5)
    NL;
    for( p=a+4,i=0; i<=4; i++ ) PR(d,p[-i]);            (Ссылки и массивы 1.6)
    NL;
    for( p=a+4; p>=a; p-- ) PR(d,a[p-a]);               (Ссылки и массивы 1.7)
    NL;
```

Результаты:

```
a[i] = 0    a[i] = 1    a[i] = 2    a[i] = 3    a[i] = 4
                                                   (Ссылки и массивы 1.1)
*p = 0      *p = 1      *p = 2      *p = 3      *p = 4
                                                   (Ссылки и массивы 1.2)

p[i] = 1    p[i] = 2    p[i] = 3    p[i] = 4    p[i] = ?
                                                   (Ссылки и массивы 1.3)
*(p+i) = 0  *(p+i) = 2  *(p+i) = 4              (Ссылки и массивы 1.4)

*p = 4      *p = 3      *p = 2      *p = 1      *p = 0
                                                   (Ссылки и массивы 1.5)
p[-i] = 4   p[-i] = 3   p[-i] = 2   p[-i] = 1   p[-i] = 0
                                                   (Ссылки и массивы 1.6)
a[p-a] = 4  a[p-a] = 3  a[p-a] = 2  a[p-a] = 1  a[p-a] = 0
                                                   (Ссылки и массивы 1.7)
```

Пояснения начинаются на с. 255.

Ссылки и массивы 2: Массив ссылок

Что напечатает следующая программа?

```
#include "defs.h"

int a[]={0,1,2,3,4};
int *p[]={a,a+1,a+2,a+3,a+4};
int **pp=p;                                        (Ссылки и массивы 2.1)

main()
{
     PRINT2(d,a,*a);
     PRINT3(d,p,*p,**p);
     PRINT3(d,pp,*pp,**pp);                        (Ссылки и массивы 2.2)
     NL;
```

```
    pp++; PRINT3(d,pp-p,*pp-a,**pp);
    *pp++; PRINT3(d,pp-p,*pp-a,**pp);
    ***pp; PRINT3(d,pp-p,*pp-a,**pp);
    ++*pp; PRINT3(d,pp-p,*pp-a,**pp);          (Ссылки и массивы 2.3)
    NL;

    pp=p;
    **pp++; PRINT3(d,pp-p,*pp-a,**pp);
    ****pp; PRINT3(d,pp-p,*pp-a,**pp);
    ++**pp; PRINT3(d,pp-p,*pp-a,**pp);         (Ссылки и массивы 2.4)
}
```

Результаты:

```
a = address of a     *a = 0                                  (Ссылки и массивы 2.2)
p = address of p     *p = address of a    **p = 0
pp = address of p    *pp = address of a   **pp = 0

pp-p = 1    *pp-a = 1   **pp = 1                             (Ссылки и массивы 2.3)
pp-p = 2    *pp-a = 2   **pp = 2
pp-p = 3    *pp-a = 3   **pp = 3
pp-p = 3    *pp-a = 4   **pp = 4

pp-p = 1    *pp-a = 1   **pp = 1                             (Ссылки и массивы 2.4)
pp-p = 1    *pp-a = 2   **pp = 2
pp-p = 1    *pp-a = 2   **pp = 3
```

Пояснения начинаются на с. 258.

Ссылки и массивы 3: Многомерные массивы

Что напечатает следующая программа?

```
#include "defs.h"

int a[3][3] = {
      { 1, 2, 3 },
      { 4, 5, 6 },
      { 7, 8, 9 }
};
int *pa[3] = {
      a[0], a[1], a[2]
};
int *p = a[0];                                    (Ссылки и массивы 3.1)

main()
{
      int i;
```

```
for( i=0; i<3; i++ )
     PRINT3(d, a[i][2-i], *a[i], *(*(a+i)+i) );
NL;                                          (Ссылки и массивы 3.2)

for( i=0; i<3; i++ )
     PRINT2(d, *pa[i], p[i] );               (Ссылки и массивы 3.3)
```

Результаты:

```
a[i][2-i] = 3    *a[i] = 1  *(*(a+i)+i)  = 1   (Ссылки и массивы 3.2)
a[i][2-i] = 5    *a[i] = 4  *(*(a+i)+i)  = 5
a[i][2-i] = 7    *a[i] = 7  *(*(a+i)+i)  = 9

*pa[i] = 1       p[i]  = 1                     (Ссылки и массивы 3.3)
*pa[i] = 4       p[i]  = 2
*pa[i] = 7       p[i]  = 3
```

Пояснения начинаются на с. 259.

Ссылки и массивы 4: Хитросплетение ссылок

Что напечатает следующая программа?

```
#include "defs.h"

char *c[] = {
     "ENTER",
     "NEW",
     "POINT",
     "FIRST"
};
char **cp[] = { c+3, c+2, c+1, c };
char ***cpp = cp;                            (Ссылки и массивы 4.1)

main()
{
     printf("%s", ***+cpp );
     printf("%s ", *--***+cpp+3 );
     printf("%s", *cpp[-2]+3 );
     printf("%s\n", cpp[-1][-1]+1 );         (Ссылки и массивы 4.2)
}
```

Результаты:

```
POINTER STEW      (Ссылки и массивы 4.1)
```

Пояснения начинаются на с. 261.

Запись, т. е. тип данных struct,— основной строительный блок данных в языке. Она предоставляет удобный способ объединения различных элементов, связанных между собой.

Записи 1: Простые записи; вложенные записи

Что напечатает следующая программа?

```
#include "defs.h"

main()
{
     static struct S1 {
          char c[4], *s;
     } s1 = { "abc", "def" };

     static struct S2 {
          char *cp;
          struct S1 ss1;
     } s2 = { "ghi", { "jkl", "mno" } };          (Записи 1.1)

     PRINT2(c, s1.c[0], *s1.s);                   (Записи 1.2)
     PRINT2(s, s1.c, s1.s);                       (Записи 1.3)

     PRINT2(s, s2.cp, s2.ss1.s);                  (Записи 1.4)
     PRINT2(s, ++s2.cp, ++s2.ss1.s);              (Записи 1.5)
}
```

Результаты:

```
s1.c[0] = a      *s1.s = d          (Записи 1.2)
s1.c = abc       s1.s = def         (Записи 1.3)
s2.cp = ghi      s2.ss1.s = mno     (Записи 1.4)
++s2.cp = hi     ++s2.ss1.s = no    (Записи 1.5)
```

Пояснения начинаются на с. 262.

Записи 2: Массив записей

Что напечатает следующая программа?

```
#include "defs.h"

struct S1 {
     char *s;
     int i;
     struct S1 *s1p;
};
```

```
main()
{
     static struct S1 a[] = {
          { "abcd", 1, a+1 },
          { "efgh", 2, a+2 },
          { "ijkl", 3, a }
     };
     struct S1 *p = a;                                    (Записи 2.1)
     int i;

     PRINT3(s, a[0].s, p->s, a[2].s1p->s);                (Записи 2.2)

     for( i=0; i<2; i++ ) {
          PR(d, --a[i].i);
          PR(c, ++a[i].s[3]);                             (Записи 2.3)
          NL;
     }

     PRINT3(s, ++(p->s), a[(++p)->i].s, a[--(p->s1p->i)].s);
                                                          (Записи 2.4)
}
```

Результаты:

```
a[0].s = abcd   p->s = abcd          a[2].s1p->s = abcd
                                                          (Записи 2.2)
--a[i].i = 0    ++a[i].s[3] = e                           (Записи 2.3)
--a[i].i = 1    ++a[i].s[3] = i
++(p->s) = bce  a[(++p)->i].s = efgi  a[--(p->s1p->i)].s = ijkl
                                                          (Записи 2.4)
```

Пояснения начинаются на с. 264.

Записи 3: Массив ссылок на записи

Что напечатает следующая программа?

```
#include "defs.h"

struct S1 {
     char *s;
     struct S1 *s1p;
};

main()
{
     static struct S1 a[] = {
          { "abcd", a+1 },
          { "efgh", a+2 },
          { "ijkl", a }
```

```
    };
    struct S1 *p[3];                                    (Записи 3.1)
    int i;

    for( i=0; i<3; i++ ) p[i] = a[i].s1p;
    PRINT3(s, p[0]->s, (*p)->s, (**p).s);               (Записи 3.2)

    swap(*p,a);
    PRINT3(s, p[0]->s, (*p)->s, (*p)->s1p->s);          (Записи 3.3)

    swap(p[0], p[0]->s1p);
    PRINT3(s, p[0]->s, (*++p[0]).s, ++(*++(*p)->s1p).s);
                                                        (Записи 3.4)
}

swap(p1,p2)
struct S1 *p1, *p2;
{
    char *temp;

    temp = p1->s;
    p1->s = p2->s;
    p2->s = temp;
}
```

Результаты:

```
p[0]->s = efgh    (*p)->s = efgh   (**p).s = efgh        (Записи 3.2)
p[0]->s = abcd    (*p)->s = abcd   (*p)->s1p->s = ijkl  (Записи 3.3)
p[0]->s = ijkl    (*++p[0]).s = abcd    ++(*++(*p)->s1p).s = jkl
                                                        (Записи 3.4)
```

Пояснения начинаются на с. 267.

Глава 9. Препроцессор

Хотя, строго говоря, препроцессор и не относится к языку Си, мало найдется программ, которые можно было бы составить без его помощи. Две наиболее важные функции препроцессора — это макроподстановка и включение файлов.

Данная глава посвящена макроподстановке. Макроподстановка, если использовать ее рассудительно,— это гибкий механизм, позволяющий повысить эффективность и наглядность программы. Если же ею пользоваться неразумно, так же как и любое другое средство языка Си, она может привести к опасным ошибкам. Чтобы разобраться в задачах этой главы, необходимо очень точно следовать правилам макроподстановки.

Препроцессор 1: Препроцессор не знает Си

Что напечатает следующая программа?

```
#include <stdio.h>
#define FUDGE(k)    k+3.14159
#define PR(a)  printf("a= %d\t",(int)(a))
#define PRINT(a)    PR(a); putchar('\n')
#define PRINT2(a,b) PR(a); PRINT(b)
#define PRINT3(a,b,c)    PR(a); PRINT2(b,c)
#define MAX(a,b)    (a<b ? b : a)

main()
{

     {
          int x=2;
          PRINT( x*FUDGE(2) );                    (Препроцессор 1.1)
     }

     {
          int cel;
          for( cel=0; cel<=100; cel+=50 )
               PRINT2( cel, 9./5*cel+32 );        (Препроцессор 1.2)
     }

     {
          int x=1, y=2;
          PRINT3( MAX(x++,y),x,y );
          PRINT3( MAX(x++,y),x,y );               (Препроцессор 1.3)
     }
}
```

Результаты:

```
x*FUDGE(2) = 7                                              (Препроцессор 1.1)
cel= 0    cel= 50    cel= 100  9./5*cel+32 = 302            (Препроцессор 1.2)
MAX(x++,y)= 2        x= 2      y = 2                        (Препроцессор 1.3)
MAX(x++,y)= 3        x= 4      y = 2
```

Пояснения начинаются на с. 270.

Препроцессор 2: Осторожность вознаграждается

Что напечатает следующая программа?

```
#include <stdio.h>
#define NEG(a)-a
#define weeks(mins) (days(mins)/7)
#define days(mins)  (hours(mins)/24)
#define hours(mins) (mins/60)
#define mins(secs)  (secs/60)
#define TAB(c,i,oi,t)    if(c=='\t')\
                           for(t=8-(i-oi-1)%8,oi=i; t; t--)\
                             putchar(' ')
```

```
#define PR(a)        printf("a= %d\t",(int)(a))
#define PRINT(a)     PR(a); putchar('\n')

main()
{
     {
          int x=1;
          PRINT( -NEG(x) );                          (Препроцессор 2.1)
     }

     {
          PRINT( weeks(10080) );
          PRINT( days(mins(86400)) );                (Препроцессор 2.2)
     }

     {
          static char input[] = "\twhich\tif?";
          char c;
          int i, oldi, temp;

          for( oldi= -1,i=0; (c=input[i])!='\0'; i++ )
               if( c<' ' ) TAB(c,i,oldi,temp);
               else putchar(c);
               putchar('\n');                        (Препроцессор 2.3)
     }
}
```

Результаты:

```
-NEG(x)= 0                    (Препроцессор 2.1)
weeks(10080) = 1              (Препроцессор 2.2)
days(mins(86400)) = 1
eleven spaces                 (Препроцессор 2.3)
```

Пояснения начинаются на с. 272.

РЕШЕНИЯ

К главе 1. Операции

Операции 1: Основные арифметические операции

Операции 1.1

```
x = - 3 + 4 * 5 - 6
```

Начнем с изучения таблицы приоритетов операций приложения 1, двигаясь от высших к низшим.

```
x = (-3) + 4 * 5 - 6
```

Наивысший приоритет в выражении имеет унарная операция —. Мы будем пользоваться скобками, чтобы показать порядок применения операций к операндам.

```
x = (-3) + (4*5) - 6
```

Следующий по порядку приоритет имеет операция *.

```
x = ((-3)+(4*5)) - 6
```

Обе операции + и — имеют один и тот же приоритет. Порядок выполнения операций, имеющих один и тот же приоритет, также задается в таблице приложения 1. Для операций + и — этот порядок слева направо, т. е. вначале выполняется +

```
x = (((-3)+(4*5))-6)
```

Затем выполняется операция —

```
(x=(((-3)+(4*5))-6))
```

Наконец, в самом низу таблицы приоритетов операций находится операция =. Теперь, когда для каждой операции определены операнды, можно вычислять выражение.

```
(x=((-3+(4*5))-6)
```

Для этого выражения вычисление начинается с самого внутреннего подвыражения.

```
  (x=((-3+20)-6)
  (x=(17-6))
  (x=11)
  11, целое
```

Заменяем каждое подвыражение на его результат.

Значение выражения, содержащего операцию присваивания, есть значение правой части выражения, приведенное к типу левой части присваивания.

О функции printf. Printf — программа форматной печати, которая входит в библиотеку стандартных программ Си. Первый аргумент printf —

это строка, задающая формат печати. Она показывает, в каком виде следует печатать все остальные аргументы функции. Символ % начинает спецификацию формата для аргумента. В нашей программе спецификация %d указывает, что следующий аргумент надо интерпретировать и печатать как десятичное число. В последующих программах мы увидим примеры других спецификаций печати. Printf может также выдавать на печать и просто написанные символы. В этой программе мы «напечатали» символ конца строки, указав его (\n) в строке спецификаций.

Операции 1.2

```
x = 3 + 4 % 5 - 6
```

Выражение очень похоже на предыдущее.

```
x = 3 + (4%5) - 6
x = (3+(4%5)) - 6
x = ((3+(4%5))-6)
(x=((3+(4%5))-6))
```

Вследствие приоритета операций и порядка их выполнения получаем такой результат. (Операция взятия остатка % дает остаток от деления 4 на 5.)

```
(x=((3+4)-6)
(x=(7-6))
(x=1)
1
```

Опять вычисление выражения начинается «изнутри».

Операции 1.3

```
x = - 3 * 4 % - 6 / 5
```

Это выражение сложнее предыдущего, но строгое следование правилу приоритетов операций и порядку их выполнения позволяет распутать его.

```
x = (-3) * 4 % (-6) / 5
```

Операции *, %, / имеют один и тот же приоритет и выполняются слева направо.

```
x = ((-3)*4) % (-6) / 5
x = (((-3)*4)%(-6)) / 5
x = ((((-3)*4)%(-6))/5)
(x=((((-3)*4)%(-6))/5))
(x=(((-3*4)%-6)/5))
(x=((-12%-6)/5))
(x=(0/5))
(x=0)
0
```

Начинаем изнутри вычислять выражение.

Операции 1.4

```
x = ( 7 + 6 ) % 5 / 2
```

Конечно, мы не всегда жестко связаны с заранее известными приоритетами операций. Если порядок выполнения нужно изменить или просто «прояснить» структуру выражения, можно использовать скобки.

```
x = (7+6) % 5 / 2
```

Вначале вычисляются подвыражения в скобках.

```
x = ((7+6)%5) / 2
```

Теперь, как и раньше, следуем правилу приоритетов операций и порядку их выполнения.

```
x = (((7+6)%5)/2)
(x=(((7+6)%5)/2))
(x=((13%5)/2))
```

Вычисляем выражение.

```
(x=(3/2))
(x=1)
1
```

При операциях с целыми числами дробная часть операнда отбрасывается.

О стиле программирования. Как говорилось в предисловии, программы этой книги не образцы для копирования, а предназначены для осмысления конструкции языка Си. Тем не менее задачи содержат определенные указания о стиле программирования. Если конструкция вынуждает вас обращаться за пояснениями к справочному руководству по языку, чтобы понять, как работают какие-то ее элементы, то или эта конструкция плохо построена, или она должна быть снабжена комментарием, содержащим недостающие элементы. Исходя из опыта решения первых задач, можно сказать, что в сложных выражениях следует использовать скобки, чтобы легче было связывать операции с соответствующими операндами.

Операции 2: Операции присваивания

Операции 2.1

вначале `x=2`

```
x *= 3 + 2
```

Опять следуем таблице приоритетов операций.

```
x *= (3+2)
```

Как мы уже видели раньше, операция присваивания имеет меньший приоритет, чем арифметические операции (*= является операцией присваивания).

```
(x*=(3+2))
(x*=5)
```

Вычисляем выражение.

```
(x = x*5)
```

Раскроем операцию присваивания.

```
(x=10)
10
```

Об операторе define. Наша программа начинается со строки

```
#define PRINTX printf("%d\n",x)
```

Каждая строка в программе на языке Си, начинающаяся символом #, есть оператор препроцессора языка. Одна из функций препроцессора заключается в замене одной строки на другую. Данный оператор define говорит препроцессору, что нужно заменять все встречающиеся в программе строки PRINTX строками printf("%d\n",x).

Операции 2.2

вначале `x=10`

```
x *= y = z = 4
```

```
x *= y = (z=4)
```

В этом выражении все операции есть операции присваивания, значит, порядок их выполнения такой, как указан в таблице приоритетов, т. е. справа налево.

```
x *= (y=(z=4))
(x*=(y=(z=4)))
(x*=(y=4))
```

Вычисляем выражение.

```
(x*=4)
40
```

Операции 2.3

вначале y=4, z=4

```
x = y == z
x = (y==z)
```

Новички, программирующие на языке Си, часто путают операцию присваивания = и сравнение на равенство = =. Из таблицы приоритетов операций видно, что приоритет операции = = выше, чем операции =.

```
(x=(y==z))
(x=(TRUE))
 (x=1)
```

Операции отношения и сравнения на равенство вырабатывают результат ИСТИНА (целое 1) или ЛОЖЬ (целое 0).

```
1
```

Операции 2.4

вначале x=1, z=4

```
x == ( y = z )
(x==(y=z))
```

В этом выражении операция присваивания благодаря действию скобок выполняется раньше, чем операция сравнения на равенство.

```
(x==4)
```

Вычисляем выражение.

```
FALSE, или 0
```

Значение выражения — 0. Заметьте, однако, что значение x не изменилось (операция = = не меняет значения ее операндов), так что PRINTX напечатает 1.

Операции 3: Логические операции и операции увеличения

Операции 3.1

вначале x=2, y=1, z=0

```
x = x && y || z
x = (x&&y) || z
x = ((x&&y)||z)
(x=((x&&y)||z))
```

Сопоставляем операции с операндами в соответствии с приоритетами операций.

```
(x=((TRUE&&TRUE)||z))
```

Логические операции выполняются слева направо. Считается, что значение операнда в логической операции есть ЛОЖЬ, если операнд равен 0, и ИСТИНА, если операнд имеет любое другое значение.

```
(x=(TRUE||z))
```

Логическая операция И (&&) вырабатывает значение ИСТИНА, если оба ее операнда имеют значение ИСТИНА, иначе она вырабатывает значение ЛОЖЬ.

```
(x=(TRUE||что-то еще)
(x=TRUE)
(x=1)
1
```

Если известно, что один из операндов логической операции ИЛИ (||) имеет значение ИСТИНА, то результат операции — ИСТИНА, независимо от значения другого операнда. Таким образом, в данном случае нет смысла продолжать вычисления дальше.

Еще об операторе define. Оператор препроцессора, начинающий эту программу, более сложный, чем в предшествующей программе. Здесь PRINT — имя макроподстановки с аргументами, а не просто строка, как ранее, которую надо заменить на другую строку. Препроцессор производит макроподстановку с аргументами в два этапа: вначале в теле подстановки формальные аргументы заменяются на фактические, затем получившаяся строка подставляется вместо обращения к макроподстановке.

Например, в этой программе PRINT имеет один формальный аргумент int. PRINT(x) — обращение PRINT с фактическим аргументом x. Таким образом, каждое вхождение int в теле подстановки заменяется на x, затем получившаяся строка—printf("%, d\n",x) подставляется вместо PRINT(x). Отметим, что аргумент int не сопоставляется с комбинацией букв int в середине идентификатора printf, так как формальный аргумент макроподстановки суть идентификатор и она сопоставляется только с идентификатором int.

Операции 3.2

```
вначале x=1, y=1, z=0
 x || ! y && z
 x || (!y) && z
 x || ((!y)&&z)
 (x||((!y)&&z))
(TRUE||((!y)&&z))
(TRUE||что-то еще)
TRUE, или 1
```

Сопоставляем в соответствии с приоритетами операций операции с операндами.

Вычисляем слева направо.

Операции 3.3

```
вначале x=1, y=1
 z = x ++ - 1
 z = (x++) - 1
 z = ((x++)-1)
 (z=((x++)-1))
```

Вследствие приоритетов операций.

```
(z=(1-1)), и x=2
```

Операция ++, стоящая справа от операнда, представляет собой постфиксную операцию. Это означает, что x увеличится на 1 после того, как значение x будет использовано в выражении.

```
(z=0)
0
```

Операции 3.4

```
вначале x=2, y=1, z=0
```

```
z += - x ++ + ++ y
z += - (x++) + (++y).
```

Унарные операции выполняются справа налево, так что операция ++ выполнится перед операцией —. (Фактически, если бы операция изменения знака выполнялась первой, выражение было бы некорректным, так как операции ++ и —— требуют, чтобы операндом было адресное значение; x есть адресное значение, а—x нет.)

```
z += (-(x++)) + (++y)
z += ((-(x++))+(++y))
(z+=((-(x++))+(++y)))
```

```
(z+=((-2)+2)), и x=3, y=2
(z+=0)
(z = 0+0)
(z=0)
0
```

Вычисление выражения начинается «изнутри».

Операции 3.5

```
вначале x=3, z=0
z = x / ++ x
z = x / (++x)
z = (x/(++x))
(z=(x/(++x)))
```

Вы могли бы поддаться искушению вычислять это выражение, как и раньше, изнутри. Вначале было бы выбрано значение x, увеличено на 1, а потом уже пойдет деление на значение x. Но возникает вопрос: какое значение x нужно использовать в качестве делимого — старое (3) или новое (4)? Иными словами, что делается раньше — выбирается ли значение x, используемое в дальнейшем в качестве делимого, или в x записывается его значение, увеличенное на 1. Описание языка не определяет, что получается, если возникают такие побочные эффекты[1]; их реализация остается за разработчиками трансляторов. Отсюда совет — избегайте выражений, вычисление которых зависит от вашего знания того, что происходит при побочных эффектах.

---

[1] Побочным эффектом называется любое изменение состояния программы как следствие выполнения некоторого оператора. Большинство побочных эффектов в Си связано с записью в память промежуточных значений переменных, которые, как выше, получаются в результате операции увеличения или в результате присваивания, встретившегося в выражении.

Операции 4: Поразрядные операции

Операции 4.1

```
вначале x=03, y=02, z=01

x ¦ y & z
```

Целые константы, начинающиеся с цифры 0 (нуль), являются восьмеричными числами. Восьмеричное представление целых чисел особенно удобно, когда приходится работать с поразрядными операциями, так как восьмеричные цифры легко переводятся в двоичные. В этой задаче числа 01, 02, 03 соответствуют числам 1, 2 и 3, так что появление восьмеричных чисел служит намеком читателю, что программа рассматривает значение x, y и z как последовательность двоичных цифр.

```
(x¦(y&z))
```

Вследствие приоритетов операций.

```
(x¦(02&01))
```

Самое внутреннее выражение вычисляется первым.

```
 0¦x
```

В двоичном представлении: 01 = 1, 02 = 10, 03 = 11

```
   10
& 01
----
   00
```

```
(03¦0)

03
```

```
   00
¦  11
----
   11
```

Операции 4.2

```
вначале x=03, y=02, z=01

x ¦ y & - z

(x¦(y&(~z)))

(x¦(y&-01))
```

Операция ~ меняет каждый разряд своего операнда на противоположный, так что 0...01 становится 1...10.

```
(x¦(02&-01))

(03¦02)
```

В двоичном представлении:

```
   0...010
& 1...110
---------
   000010
```

```
3
```

```
   10
¦  11
----
   11
```

Операции 4.3

```
вначале x=03, y=02, z=01
x ^ y & ~ z
```

```
(x^(y&(-z)))
```

Здесь выражение похоже на предыдущее, только операция включающего ИЛИ (¦) заменена на операцию исключающего ИЛИ (^).

```
(x^(02&~01))
(03^02)
1
```

В двоичном представлении:

```
  10
^ 11
----
  01
```

Операции 4.4

```
вначале x=03, y=02, z=01
x & y && z
((x&y)&&z)
((03&02)&&z)
(02&&z)
(TRUE&&z)
(TRUE&&01)
(TRUE&&TRUE)
TRUE, или 1
```

Операция && дает результат ИСТИНА, когда оба операнда имеют значение ИСТИНА.

Операции 4.5

```
вначале x=01
! x ¦ x
((!x)¦x)
((!TRUE)¦x)
(FALSE¦01)
(0¦01)
1
```

Операции 4.6

```
вначале x=01
~ x ¦ x
((~x)¦x)
(~01¦01)
-1
```

В двоичном представлении:

```
   1...110
¦  0...001
----------
   1...111, или -1
```

(Ответ будет одним и тем же для всех значений x. На самом же деле на машинах, использующих дополнительный код, таких, как PDP-11, ответ будет —1, а на машинах с обратным кодом 1...1, что можно считать 0. Для тех немногих случаев, когда это имеет значение, в нашей книге будет использоваться результат, получаемый на машинах с дополнительным кодом.)

Операции 4.7

```
вначале x=01
x ^ x
(01^01)
0
```

В двоичном представлении:

```
  0...01
^ 0...01
--------
  0...00
```

(Ответ будет одним и тем же для всех значений x.)

Операции 4.8

```
вначале x=01
x <<= 3
x = 01<<3
x=8
```

В двоичном представлении:

```
  0000...01
<<        3
-----------
     0...01000, что равно 8
```

Каждый сдвиг влево на 1 разряд равнозначен умножению на 2.

Операции 4.9

```
вначале y=-01
y <<= 3
y = -01<<3
y = -8
```

В двоичном представлении:

```
  1111...11
<<        3
-----------
     1...11000, или -8
```

Операции 4.10

```
вначале y=-08
y >>= 3
y = -08>>3
```

В этот момент может появиться искушение сказать, что y = —1. К сожалению, это будет не всегда так, поскольку на некоторых машинах при операции сдвига знак числа может не сохраниться. Язык Си не гаранти-

рует, что операция сдвига арифметически корректна, поэтому в любом случае более ясным способом деления на 8 было бы явное деление: y = y/8

Операции 5: Отношения и условия

Операции 5.1

```
вначале x=3, y=2, z=1

x < y ? y : x

(x<y) ? (y) : (x)
```

Операция условия, если не считать, что она использует три операнда, разбирается подобно любой другой операции.

```
((x<y)?(y):(x))

(FALSE?(y):(x))
```

Вначале вычисляется условие. Затем выполняется или часть операции, соответствующая истинному значению условия, или часть, соответствующая ложному значению условия, но не обе вместе.

```
((x))
```

В этой задаче значение условия — ЛОЖЬ, поэтому значением всего выражения будет значение выражения, соответствующего ложному значению условия.

```
3

(3)
```

Операции 5.2

```
вначале x=3, y=2, z=1

x < y ? x ++ : y ++

((x<y)?(x++):(y++))

(FALSE?(x++):(y++))

((y++))

(2), и y=3

2
```

Вначале вычисляем условие.
Значение условия — ЛОЖЬ, поэтому должна вычисляться часть, соответствующая ложному значению условия.

(Так как операция x + + не выполнялась, то значение x остается равным 3.)

Операции 5.3

```
вначале x=3, y=3, z=1

z += x < y ? x ++ : y ++

(z+=((x<y)?(x++):(y++)))

(z+=(FALSE?(x++):(y++)))

(z+=((y++))
```

Результат условного выражения — это результат присваивания.

```
(z+=(3)), и y=4

(z=z+3)

(z=4)

4
```

Операции 5.4

```
вначале x=3, y=4, z=4
(z >= y >= x) ? 1 : 0
(((z>=y)>=x)?(1):(0))
((TRUE>=x)?(1):(0))
```

Условие начинает вычисляться изнутри.

```
((1>=x)?(1):(0))
```

Значение самого внутреннего условия — ИСТИНА. Оно сравнивается со значением целого x. Хотя это и законно в языке Си, и со значением ИСТИНА обходятся вольно, считая его целым числом, в данном случае это 1. Однако часто, как и в этой задаче, это не приносит желаемых результатов. (Следующая задача показывает правильный способ для сравнения трех величин.)

```
(FALSE?(1):(0))
((0))
0
```

Операции 5.5

```
вначале x=3, y=4, z=4
z >= y && y >= x
((z>=y)&&(y>=x))
(TRUE&&(y>=x))
```

Вычисляем выражение слева направо.

```
(TRUE&&TRUE)
(TRUE)
1
```

Операции 6: Выполнение операций и их приоритеты

Операции 6.1

```
вначале x=1, y=1, z=1
++x || ++y && ++z
((++x)||((++y)&&(++z)))
(2||((++y)&&(++z))), и x=2
(TRUE||что-то еще)
```

Сопоставляем операнды с операциями. Вычисляем выражение слева направо. Поскольку левый операнд операции || имеет значение ИСТИНА, нет смысла проводить вычисления дальше. Фактически в языке Си гарантируется, что в подобных случаях вычисления прекращаются. Иными словами, логическое выражение вычисляется слева направо до тех пор, пока не станет известно его значение, т. е. для данной задачи y и x останутся равными 1.

```
TRUE, или 1
```

Операции 6.2

```
вначале x=1, y=1, z=1
 ++ x && ++ y || ++ z
 (((++x)&&(++y))||(++z))
 ((TRUE&&(++y))||(++z)), и x=2
```

```
((2&&2)||(++z)), и y=2
```

Вычисляем выражение слева направо.

```
(TRUE||(++z))
TRUE, или 1
```

Значение z не меняется.

О порядке выполнения операций и их приоритетах. Для большинства операций порядок их выполнения определяется приоритетами. Но, как демонстрируют задачи этого раздела, существует несколько исключений из этого правила:

префиксные операции увеличения и уменьшения всегда выполняются прежде, чем их операнд будет использован в выражении;

постфиксные операции увеличения и уменьшения всегда выполняются после того, как их операнд будет использован в выражении;

логические операции И и ИЛИ всегда выполняются слева направо условно.

Операции 6.3

```
вначале x=1, y=1, z=1
 ++ x && ++ y && ++ z
 (((++x)&&(++y))&&(++z))
 ((2&&2)&&(++z)), и x=2, y=2
 (TRUE&&(++z))
 (TRUE&&TRUE), и  z=2
 TRUE, или 1
```

Операции 6.4

```
вначале x=-1, y=-1, z=-1
 ++ x && ++ y || ++ z
 (((++x)&&(++y))||(++z))
 ((0&&(++y))||(++z)), и x=0
 ((FALSE&&(++y))||(++z))
(FALSE||(++z))
```

Здесь нет необходимости вычислять ++y, так как левый операнд логической операции && есть ЛОЖЬ. Но результат логической операции || пока еще не известен.

```
(FALSE||(0)), и z=0
 (FALSE||FALSE)
 FALSE, или 0
```

Операции 6.5

```
вначале x=-1, y=-1, z=-1
++x || ++y && ++z
((++x)||((++y)&&(++z)))
(FALSE||((++y)&&(++z))), и x=0
(FALSE||(FALSE&&(++z))), и y=0
(FALSE||FALSE)
FALSE, или 0
```

Операции 6.6

```
вначале x=-1, y=-1, z=-1
++x && ++y && ++z
(((++x)&&(++y))&&(++z)
((FALSE&&(++y))&&(++z)), и x=0
(FALSE&&(++z))
FALSE, или 0
```

Относительно побочных эффектов при вычислении логического выражения. К этому моменту вы уже без сомнения могли заметить, что вычисление логических выражений в Си может быть коварным, поскольку вычисление правой части выражения может проводиться в зависимости от значения левой его части. На самом деле такое условное вычисление является полезным свойством логических операций. Проблемы возникают только когда правая часть логического выражения содержит побочный эффект; иногда побочный эффект будет иметь место, а иногда и нет. Так что если вообще побочными эффектами нужно пользоваться с осторожностью, то в логических выражениях это следует делать сверхосторожно.

К главе 2. Основные типы

Основные типы 1: Символ, строка и целый тип

Основные типы 1.1

```
PRINT(d,"5")
```

Формат %d указывает printf, что аргумент нужно напечатать как десятичное число. Аргумент "5" представляет собой ссылку на массив символов т. е. адрес массива из двух символов '5' и '\0').

```
PRINT(d,'5')
```

%d требует напечатать десятичное значение символа '5'[1].

```
PRINT(d,5)
```

Целое число 5 печатается в десятичном виде.

```
PRINT(s,"5")
```

• Формат %s указывает printf, что аргумент является ссылкой на массив символов. Так как "5"—ссылка на массив символов, то печатается содержимое этого массива, т. е. число 5.

[1] Значение, которое используется здесь, есть значение символа в кодировке ASCII (приложение 3). Кодировка ASCII — одна из возможных кодировок для представления символов там, где необходимо значение символа.

```
PRINT(c,'5')
```

Формат %c указывает printf, что аргумент нужно рассматривать как значение некоторого символа (его код) и что этот символ нужно напечатать. Так как '5' как раз и есть значение символа, то напечатан будет символ *5*.

```
PRINT(c,53)
```

Десятичное число *53* — это код символа *5* в кодировке ASCII.

```
PRINT(d,('5'>5))
```

Последней будет напечатана 1, так как '5' имеет большее значение (53), чем целое *5*.

Основные типы 1.2

вначале `sx=-8, ux=-8`

```
PRINT(o,sx)
```

%o указывает printf, что аргумент следует напечатать как восьмеричное число.

```
PRINT(o,ux)
```

Значение —8 представляется как строка из 0 и 1, что верно как для переменных без знака, так и для переменных со знаком.

```
PRINT(o,sx>>3)
```

С этой трудностью мы сталкивались уже и раньше. В некоторых версиях языка Си сдвиг целого со знаком вправо приводит к тому, что знаковый разряд копируется в свободные старшие разряды слова, т. е. знак сохраняется. *Однако, внимание,— это свойство зависит от транслятора!*

```
PRINT(o,ux>>3)
```

Когда происходит сдвиг целого без знака вправо, то освободившиеся старшие разряды всегда заполняются нулями.

```
PRINT(d,sx>>3)
```

Сдвиг на 3 разряда вправо целого со знаком —8 дает ожидаемый результат —1 при условии, что знак сохраняется, и 8191 в противном случае (на 16-разрядной машине с дополнительным кодом).

```
PRINT(d,ux>>3)
```

Для переменной типа unsigned со значением —8 результат всегда равен 8191 (на 16-разрядной машине).

Основные типы 2: Приведение целых чисел и чисел с плавающей точкой

Основные типы 2.1

```
i = l = f = d = 100/3
```

```
(i= (l= (f= (d=.(100/3)) )))
```

```
(i= (l= (f= (d=33) )))
```

Вычисляем выражение справа налево.
Так как оба числа 100 и 3— целые, то операция деления есть деление целых, поэтому у частного отбрасывается дробная часть.

```
(i= (l= (f=(double)33) )), и d=33
```

Напоминаем, что значение, вырабатываемое операцией присваивания, есть значение правой части, преобразованное к типу, задаваемому левой частью.

```
(i= (l=(float)33) ), и f=33
(i=(long)33), и l=33
(integer)33, и i=33
33, целое
```

Основные типы 2.2

```
d = f = l = i = 100/3
(d= (f= (l= (i=(100/3)) )))
(d= (f= (l=(integer)33) )), и i=33
(d= (f=(long)33) ), и l=33
(d=(float)33), и f=33
((double)33), и d=33
33, двойной точности
```

Основные типы 2.3

```
i = l = f = d = 100/3.
(i= (l= (f= (d= (100/3.)) )))
(i= (l= (f=(double)33.333333) ))
и d=33.333333
```

Число 3. двойной точности, поэтому и частное будет двойной точности.

```
(i= (l=(float)33.333333)
и  f=33.33333x
```

В этой программе в printf используется формат %.8g, который задает печать числа с точностью до 8 значащих цифр. Но на самом деле на машинах PDP-11 и VAX максимальная точность для чисел с плавающей точкой не более 7 значащих цифр, так что точность до 8 цифр недостижима. Конечно, количество значащих цифр в числе зависит от машины.

```
(i=(long)33.33333x), и l=33
```

Преобразование чисел с плавающей точкой в длинные целые происходит с помощью отбрасывания дробной части.

```
(integer)33), и i=33
33, целое
```

Основные типы 2.4

```
d = f = l = i = (double)100/3

(d= (f= (l= (i= ((double)100) /3))))
```

Заметьте что операция приведения имеет более высокий приоритет, чем операция /.

```
(d= (f= (l= (i=33.333333) )))
(d= (f= (l=(integer)33.333333) ))
    и i=33
(d= (f=(long)33) ), и l=33
(d=(float)33), и f=33
((double)33), и d=33
```

33, двойной точности

Основные типы 2.5

```
i = l = f = d = (double)(100000/3)
(i= (l= (f= (d= ((double)(100000/3)) ))))

(i= (l= (f= (d=(double)33333) )))
```

Операндом для операции приведения служит частное от деления 100000 на 3.

```
(i= (l= (f=(double)33333) )), и d=33333
(i= (l=(float)33333) ), и f=33333
(i=(long)33333), и l=33333
((integer)33333), и i=33333 или переполнение
```

Число 33333 нельзя представить в виде 16-разрядного целого со знаком. Многие реализации языка спокойно допускают и переполнение, и потерю значимости числа. Если ваши вычисления могут в принципе превзойти границы, присущие данной машине, то будет разумным вставить явные проверки на попадание чисел в отводимый диапазон.

33333, целое или переполнение

Основные типы 2.6

```
d = f = l = i = 100000/3
(d= (f= (l= (i=100000/3) )))
(d= (f= (l=(integer)33333) ))
```

и i=33333, или переполнение

Как мы уже видели раньше, 33333 — значение, переполняющее 16-разряд-

ное целое со знаком. Если для представления целых использовать больше разрядов, то i так же, как l, f и d, получат значение 33333. Ниже мы ориентируемся на 16-разрядные целые.

```
(d= (f=(long)-32203) )
и l=-32203
```

Результат операции, приведшей к переполнению,— это обычное число, но не то, которое ожидалось. Значение 33333 будет потеряно независимо от дальнейших приведений типов.

```
(d=(float)-32203), и f=-32203
((double)-32203), и d=-32203
-32203, двойной точности
```

О числах. Работа с числами не самая сильная сторона языка Си. Язык не дает способа выявить арифметические ошибки, даже если аппаратура и имеет такие средства. Диапазон представления чисел фиксируется при написании транслятора, и в языке нет средств, чтобы задать этот диапазон. Лучшее, что можно сделать для проверки попадания числа в диапазон,— это явный контроль значений переменных в критических точках вычисления.

Основные типы 3: Еще о приведении типов

Основные типы 3.1

```
вначале d=3.2, i=2
x = (y=d/i)*2
(x= (y=3.2/2) *2)
(x= (y=1.6)*2)
```

3.2. —число двойной точности, этот тип выше типа int, типа числа 2, поэтому частное также будет двойной точности. Так как y — целое, то у значения 1.6 будет отброшена дробная часть.

```
(x=1*2), и y=1
(x=2)
2, и x=2
```

Основные типы 3.2

```
вначале d=3.2, i=2
y = (x=d/i)*2
(y= (x=1.6)*2)
(y=1.6*2), и x=1.6
```

Так как x — типа double, то и результат присваивания будет типа double.

```
(y=3.2)
```

Тип. 1.6 — double определяет и тип произведения.

```
3, и y=3
```

Так как y типа int, то у 3.2 должна быть отброшена дробная часть.

Основные типы 3.3

```
вначале d=3.2, i=2
y = d * (x=2.5/d)
(y= d* (x=2.5/d) )
(y= d*2.5/d ), и x=2.5/d
(y=2.5)
2, и y=2
```

Тип x—double, так что у 2.5/d точность сохраняется.

Тип y заставляет отбросить дробную часть у 2.5.

Основные типы 3.4

```
вначале d=3.2, i=2
x = d * (y = ((int)2.9+1.1)/d)
(x= d* (y=(2+1.1)/d) )
(x= d* (y=3.1/d) )
(x= d* (y=.нечто ) )
(x=d*0), и y=0
0., и x=0
```

Операция приведения более высокого приоритета, чем операция +.

y получит значение 0 независимо от величины «не́что», поскольку «не́что» находится между 0 и 1.

О смешивании типов. К этому моменту было уже много примеров, когда смешивание целых значений и значений с плавающей точкой приводило к неожиданным результатам в выражениях. Лучше всего избегать арифметических действий с операндами разных типов. Если же это необходимо, то применять явные приведения типов.

К главе 4. Управление

Управление 1: Условный оператор

Управление 1.1

```
вначале y=1
if( y!=0 ) x=5;
( y!=0 )
( 1!=0 )
TRUE
x = 5
```

На первом шаге вычисляются условия.

Так как условие — ИСТИНА, то выполняется первая часть условного оператора.

Управление 1.2

```
вначале y=1

if( y==0 ) x=3; else x=5;
```

```
( y==0 )
```

Вычисляем условие.

```
FALSE
```

```
x = 5
```

Выполняем вторую (ложную) часть условного оператора.

Управление 1.3

```
вначале y=1

x=1
if( y<0 ) if( y>0 ) x=3;
else x=5;
```

```
x=1
if( y<0 ) {
  if( y>0 ) x=3;
  else x=5;
}
```

Вначале x присваивается 1.
Фигурные скобки показывают вложенность операторов.

```
( y<0 )
```

```
FALSE
```

Условие первого условного оператора — ЛОЖЬ, так что часть оператора, соответствующая истинному значению условия, пропускается. Оператор, следующий за else, находится в первой части первого условного оператора, так как он относится ко второму условному оператору. В языке Си действует правило: оператор, следующий за else, относится к ближайшему if.

Управление 1.4

```
вначале y=1

if( z=y<0 ) x=3;
else if( y==0 ) x=5;
else x=7;
```

```
( z=(y<0) )
```

Начинаем с вычисления первого условия. Как и прежде, мы будем использовать скобки для обозначения соответствия операций и операндов.

```
( z=(1<0) )

( z=FALSE )

FALSE, и  z=0
```

```
( y==0 )
```

Так как условие первого if — ЛОЖЬ, выполняется вторая часть этого оператора, которая опять оказывается условным оператором; значит, вычисляется его условие.

```
FALSE
```

```
x = 7
```

Условие — ЛОЖЬ, поэтому выполняется вторая часть второго условного оператора.

Управление 1.5

```
вначале y = 1

if( z=(y==0) ) x=5; x=3

if( z=(y==0) ) { x=5; } x=3;
```

Первая часть условного оператора может быть одиночным оператором или блоком и следует сразу за условием.

```
( z=(y==0) )

( z=FALSE )

FALSE, и z=0

x = 3
```

Вычисляем условие.

Так как условный оператор не имеет второй части, то управление переходит к следующему оператору.

Управление 1.6

```
вначале y=1

if( x=z=y ); x=3;

if( x=z=y ) { ; } x=3;
```

Первая часть условного оператора есть пустой оператор.

```
( x=(z=y)) )

( x=(z=1))

( x=1 ), и z=1

TRUE, и x=1

x = 3
```

Вычисляем условие.

Условие — ИСТИНА, поэтому выполняется первая часть оператора, но поскольку там пустой оператор, то выполняется оператор, следующий за условным.

Управление 2: Операторы while и for

Управление 2.1

```
вначале x=0, y=0

while( y<10 ) ++y; x += y;

while( y<10 ) ++y;

( y<10 )

( y>=10 )

y = 0
```

Начнем с изучения факторов, управляющих выполнением оператора while.

Условие продолжения цикла. Тело цикла будет выполняться до тех пор, пока условие продолжения цикла — ИСТИНА.

Условие выхода из цикла. Это отрицание условия продолжения цикла, и в случае нормального окончания цикла оно имеет значение ИСТИНА.

Начальное значение управляющей переменной — это значение, которое имеет такая переменная при первом выполнении тела цикла.

```
++y
```

Изменение управляющей переменной при выполнении тела цикла.

```
y = 0 до 9 в цикле
```

При первом выполнении цикла y = 0. Каждый раз в теле цикла значение y увеличивается на 1.

```
y = 10 на выходе
```

Когда y = 10, условие продолжения цикла становится ложным и выполнение цикла прекращается.

```
x += y;
```

Управление передается оператору, следующему за телом цикла.

```
x = 0+10
x = 10
```

Управление 2.2

```
вначале x=0, y=0
while( y<10 ) x += ++y;
```

```
( y<10 )
```
Условие продолжения цикла.

```
( y>=10 )
```
Условие выхода из цикла.

```
y = 0
```
Начальное значение управляющей переменной.

```
++y
```
Изменение в цикле управляющей переменной.

```
y = 0 до 9 в цикле
```
Так же, как и в предыдущей задаче.

```
x += ++y
```
К x в цикле после его увеличения на 1 прибавляется значение y.

```
x = 55
```
Сумма всех целых от 1 до 10.

```
y = 10 на выходе
```

Управление 2.3

```
вначале y=1
while( y<10 ) { x = y++; z = ++y; }
```

```
( y<10 )
```
Условие продолжения цикла.

```
( y>=10 )
```
Условие выхода из цикла.

```
y = 1
```
Начальное значение управляющей переменной.

```
y++, ++y
```
Изменение в цикле управляющей переменной.

```
y = 1,3,5,7,9 в цикле
```
В первый момент в цикле y = 1, и при каждом выполнении цикла оно увеличивается на 2.

```
x = 1,3,5,7,9
```
В цикле до того, как y увеличивается на 1, его значение присваивается x.

```
z = 3,5,7,9,11
```
В цикле после того, как y увеличивается на 2, его значение присваивается z.

```
y = 11 на выходе
```

Управление 2.4

```
for( y=1; y<10; y++ ) x=y;
```

В операторе for собраны все выражения, определяющие выполнение тела цикла.

```
y<10
```

Условие продолжения цикла.

```
y>=10
```

Условие выхода из цикла.

```
y=1
```

Начальное значение.

```
y++
```

Изменение.

```
y = 1 до 9 в цикле
x = 1 до 9
```

В теле цикла x принимает значение y.

```
y = 10 на выходе
```

Управление 2.5

```
for( y=1; (x=y)<10; y++ ) ;
```

Условие продолжения цикла.

```
y<10
```

```
y>=10
```

Условие выхода из цикла.

```
y=1
```

Начальное значение.

```
y++
```

Изменение.

```
y = 1 до 9 в цикле
x = 1 до 10
```

Как раз перед вычислением условия продолжения цикла x принимает значение y. Заметим, что условие продолжения цикла вычисляется на один раз больше, чем выполняется тело.

```
y = 10 на выходе
```

Управление 2.6

```
for( x=0,y=1000; y>1; x++,y/=10 )
  PRINT2(d,x,y);
```

```
y>1
```

Условие продолжения цикла.

```
y<=1
```

Условие выхода из цикла.

```
y=1000
```

Начальное значение.

```
y/=10
```

Изменение.

```
y = 1000,100, в цикле
x = 0,1,2 в цикле
```

Перед началом цикла x = 0. После выполнения тела цикла, но перед вычислением условия продолжения цикла x увеличивается. (Оператор PRINT2 находится в теле цикла.)

```
y = 1 на выходе
x = 3 на выходе
```

Упражнение 3: Вложенность операторов

Упражнение 3.1

```
Вначале i=in=high=low=0, input="PI=3.14159, приблизительно";
```

```
while( c=(NEXT(i)!=EOS) )
```

Условие продолжения цикла заключается в том, что NEXT(i)!=EOS, где NEXT(i) последовательно принимает значения символов из input. Переменная с принимает значение логического выражения NEXT(i)!=EOS, которое по определению равно ИСТИНА в цикле и ЛОЖЬ по выходе из цикла.

```
if( 1<'0' ) low++
```

Так как в цикле с всегда имеет значение 1, то low всегда будет увеличиваться (1<060).

```
while( c=(I!=EOS) )
```

Цикл продолжается до тех пор, пока не будут прочитаны все символы из input. В качестве символа конца строки в языке Си используется нулевой символ «алфавита» ASCII, т. е. 00.

Упражнение 3.2

```
Вначале  i=in=high=low=0, done=FALSE,
         input="PI=3.14159, приблизительно"
```

```
while( (c=NEXT(i))!=EOS && !done )
```

Переменная с последовательно принимает значения символов из input.

```
if( 'P'<'0' )
```

При первом выполнении цикла с='p', следовательно, в условном операторе условие—ЛОЖЬ.

```
else if( 'P'>'9' )
```

ИСТИНА, и high увеличивается на 1.

```
while( 'I'!=EOS && !done )
```

Возвращаемся к условию продолжения цикла. Условный оператор, сравнивающий low, high и in с ENUF, находится вне тела цикла, хотя и кажется, что он в него входит. Так как done в теле не изменяется, то цикл заканчивается, когда с==EOS. В цикле счетчики low, in и high увеличиваются в зависимости от результата сравнения с с символами '0' и '9'.

Упражнение 3.3

```
Вначале i=in=high=low=0, done=FALSE,
        input="PI=3.14159, приблизительно"
```

```
while( (c=NEXT(i))!=EOS && !done ) {
```
Переменная c последовательно принимает значения символов из input.

```
if( 'P'<'0' )
```
ЛОЖЬ.

```
else if( 'P'>'9' )
```
ИСТИНА.

```
done = (++high==ENUF)
```
Переменная high после увеличения не равна ENUF, так что переменной done присваивается значение **ЛОЖЬ**. high = 1.

```
while( 'I'!=EOS && !done )
```
ИСТИНА.

```
if( 'I'<'0' )
```
ЛОЖЬ.

```
else if( 'I'>'9' )
```
ИСТИНА.

```
done = (++high==ENUF)
```
high = 2, done = **ЛОЖЬ.**

```
while( '='!=EOS && !done )
```
ИСТИНА.

```
if( '='<'0' )
```
ЛОЖЬ.

```
else if( '='>'9' )
```
ИСТИНА.

```
done = (++high==ENUF)
```
high = 3, done = **ИСТИНА.**

```
while( '3'!=EOS && !done )
```
done = **ИСТИНА**, так что !done = **ЛОЖЬ**, и цикл завершается.

Управление 4: Переключатели и операторы разрыва и продолжения

Управление 4.1

```
char input[]="SSSWILTECH1\1\11W\1WALLMP1"
```
Массив символов input инициируется строкой символов „SSS...MP1".

```
for(i=2; (c=input[2])!='\0';
```
Переменная c принимает значения символов из массива input, начиная с третьего.

```
switch('S') {
```
При первом проходе по переключателю c = 'S'.

```
default: putchar('S')
```
Так как ни одна из меток вариантов не совпала с 'S', то выбирается вариант default. Печатается S.

```
continue
```
Оператор продолжения приводит к очередному повторению самого внутреннего из объемлющих циклов, в данном случае это оператор for. Заметим, что оператор продолжения действует как оператор перехода к повторной инициации в заголовке оператора for.

```
for( ; (c=input[3])!='\0'; i++) {
```
Значение четвертого символа из input записывается в c.

```
switch('W') {
```
c = 'W'.

```
default: putchar('W'); continue
```
Как и раньше, печатается W.

```
                                    Аналогично для i=4, c='I',
switch('L') {                       i= 5, c='L'.
case 'L': continue                  Выбирается вариант 'L', ничего не печатается.
```

в цикле for

```
i=5, c='L';                         Ничего не печатается.
i=6, c='T';                         Печатается T.
i=7, c='E';                         Ничего не печатается.
i=8, c='C';                         Печатается C.
i=9, c='H';                         Печатается H.
switch('1') {                       i=10, c = '1'.
case '1': break                     Оператор разрыва приводит к выходу из ближайшего объемлющего цикла или переключателя. В данном случае он вызывает переход на оператор, следующий за переключателем.
putchar(' ')                        Печатается пробел.
for( ; (c=input[11])!='\0'; i++) {  Возвращаемся назад к началу оператора for.
switch('\1') {                      Символьная константа '\n', где n состоит не более чем из четырех восьмеричных цифр, соответствует символу с указанным восьмеричным кодом.
                                    Например, в кодировке ASCII \0 соответствует символу nul, а \101 — символу A.
case 1:                             Метками вариантов могут быть целые или символьные константы. Значение \1 совпадает с целым 1, так как в языке тип char автоматически приводится к int.
while( (c=input[++i])!='\1' && c!='\0') ;   Условие окончания цикла для оператора while есть или c=='\1', или конец строки. Каждый раз, когда проверяется условие оператора while, i увеличивается на 1, так что выполнение цикла приведет к тому, что i станет индексом символа из input, следующего за '\1' или за концом строки.
```

в цикле while

`i=12, c='\11';`	Ничего не печатается.
`i=13, c='W';`	Ничего не печатается.
`i=14, c='\1';`	Оператор while завершается.
`case 9: putchar('S')`	Операторы каждого варианта выполняются последовательно, один за другим; между ними нет никакого неявного оператора разрыва. За вариантом 1 следует вариант 9. Печатается S.
`case 'E': case 'L': continue`	За вариантом 9 следуют варианты 'E' и 'L'.
`for( ; (c=input[15]); i++) {`	Опять возвращаемся назад к началу оператора for.

в цикле for

`i=15, c='W';`	Печатается W.
`i=16, c='A';`	Печатается A.
`i=17, c='L';`	Ничего не печатается.
`i=18, c='L';`	Ничего не печатается.
`i=19, c='M';`	Печатается M.
`i=20, c='P';`	Печатается P.
`i=21, c='1';`	Печатается пробел.
`i=22, c='\0';`	Завершается оператор for.
`putchar('\n')`	

К главе 5. Стиль программирования

Стиль программирования 1: Составьте правильно условие

Стиль программирования 1.1

Если изменить условие, то часто удается избавиться от оператора продолжения. Иногда получившаяся программа становится существенно проще.

В этой задаче достаточно изменить условие в условном операторе.

```
while(A)
      if(!B) C;
```

Стиль программирования 1.2

Оператор do...while — это такая конструкция языка Си, от которой лучше избавляться. Если можно использовать либо оператор do...while, либо оператор while, то следует всегда предпочитать второй, так как в нем условие проверяется перед каждым выполнением тела цикла. Тот факт, что условие продолжения цикла не проверяется перед первым выполнением тела цикла, служит источником многих ошибок.

В этой задаче операторы do...while и условный — лишние; их можно заменить оператором while.

```
do {
     if(A) { B; C; }
  while(A);
```

Вначале избавляемся от оператора продолжения.

```
while(A) {
     B; C;
}
```

Затем заменяем операторы do...while и условный на оператор while.

Стиль программирования 1.3

Насколько трудно разобраться в конструкциях со многими вложенными условными операторами, хорошо известно программистам: когда доходишь до самого внутреннего условия, внешние условия забываешь или упускаешь из виду. Можно, конечно, полностью пояснить условия, но тогда они становятся слишком длинными и непонятными с самого начала. Во всяком случае, лучше всегда придерживаться меры. Здесь предлагаются два способа решения этой проблемы:

```
if( A && B && C ) D;
else if( !A && B && C ) E;
else if( !A && B && !C ) F;
```

или

```
if( B )
     if( A && C ) D;
     else if( !A && C ) E;
     else if( !A && !C ) F;
```

Стиль программирования 1.4

По идее, эта задача строго иерархична:
пока еще есть символы в строке;
в зависимости от вида символа используется один из нескольких вариантов:
выдается ALPHA,
выдается DIGIT,
выдается OTHER.
Это легко записать на языке Си:

```
while( (c=getchar()) != '\n' ) {
     if( c>='a' && c<='z' ) return(ALPHA);
     else if( c>='0' && c<='9' ) return(DIGIT);
     else if( c!=' ' && c!='\t' ) return(OTHER);
}
return(EOL);
```

Стиль программирования 2: Выберите подходящую конструкцию

Стиль программирования 2.1

```
done = i = 0;
while( i<MAXI && !done ) {
     if( (x/=2) > 1 ) i++;
     else done++;
}
```

Первое наблюдение — конструкция if...continue действует как if...else. Так и используйте if...else.

```
i = 0;
while( i<MAXI && (x/=2)>1 ) i++;
```

Далее становится ясно, что:
условие продолжения цикла требует, чтобы done было ЛОЖЬ;
done — ЛОЖЬ, пока условие в условном операторе ИСТИНА;
таким образом, одно из условий продолжения цикла $(x/2) > 1$.
Укажите это явно!

```
for( i=0; i<MAXI && (x/=2)>1; i++ ) ;
```

Оператор while, которому предшествует инициация и который содержит изменение управляющей переменной, есть в точности оператор for.

Стиль программирования 2.2

Обычно некоторую идею в Си можно реализовать многими способами. Полезно объединять эти идеи в некоторые группы (нечто целое). В языке есть несколько уровней представления таких групп:

самый низший уровень — идея становится выражением;
выражения объединяются в операторы;
операторы объединяются в блоки и функции.

В этой задаче существуют два уровня иерархии. На низшем уровне находятся выражения B, D, F и G. Вместе они существуют как взаимоисключающие варианты одного переключателя. Более органичным представлением переключателя общего вида будет конструкция if...else if.

```
if(A). B;
else if(C) D;
else if(E) F;
else G;
return;
```

Стиль программирования 2.3

Самое главное, что надо обнаружить в этой задаче,— это то, что используемая конструкция эквивалентна переключателю с тремя взаимоисключающими вариантами.

```
plusflg = zeroflg = negflg = 0;

if( a>0 ) ++plusflg;
else if( a==0 ) ++zeroflg;
else ++negflg;
```

Стиль программирования 2.4

```
i = 0;
while( (c=getchar())!=EOF && c!='\n' ) {
      if( c!='\n' && c!='\t' ) {
  s[i++] = c;
```

Первое, что надо сделать,— это переписать выражения, выделив вложенность

```
   continue;
      }
      if( c=='\t' ) c = ' ';
      s[i++] = c;
}
```

операторов. Затем обратить внимание на операторы разрыва и продолжения, действительно ли они нужны. От оператора разрыва легко избавиться, если добавить отрицание его условия к условию в операторе while.

```
i = 0;
while( (c=getchar())!=EOF && c!='\n' ) {
      if( c!='\t' ) {
   s[i++] = c;
   continue;
      }
      if( c=='\t' ) s[i++] = ' ';
}
```

Теперь можно убрать первое условие в условном операторе. (c !='\n' находится теперь в условии цикла, значит, в теле цикла оно всегда будет истинно.)

```
i = 0;
while( (c=getchar())!=EOF && c!='\n' )
      if( c!='\t' ) s[i++] = c;
      else s[i++] = ' ';
```

Оператор продолжения действует как конструкция if...else.

```
for( i=0; (c=getchar())!=EOF && c!='\n'; i++ )
      if( c!='\t' ) s[i] = c;
      else s[i] = ' ';
```

Наконец, очевидно, что s[i] — следующий символ, если это не табуляция; иначе же это пробел.

или

```
for( i=0; (c=getchar())!=EOF && c!='\n'; i++ )
      s[i] = c!='\t' ? c : ' ';
```

Другими словами, здесь происходит замена табуляций на пробелы. Последние два варианта показывают это достаточно ясно. Очевидно также и сходство между условным оператором и условным выражением. В этом примере в условном операторе акцент делается на проверке на табуляцию |, а в условном выражении — на присваивании s[i].

```
if( j>k ) y = j / (x!=0 ? x : NEARZERO);
else y = k / (x!=0 ? x : NEARZERO);
```

В этой задаче очевидно, что x!=0 не основное соображение; проверка просто предохраняет от деления на 0. Условное выражение естественно реализует эту проверку.

```
y = MAX(j,k) / (x!=0 ? x : NEARZERO);
```

Далее, можно прийти к выводу, что основное здесь — присваивание переменной y, которое включает обе проверки (MAX—функция, выдающая наибольший из ее двух параметров).

К главе 6. Классы памяти

Классы памяти 1: Блоки

Классы памяти 1.1

```
int i=0,
```

i.0=0
(Обозначение x.*n* используется для того, чтобы показать обращение к переменной x, определенной в блоке уровня *n*[1].) Класс памяти переменной i.0 — внешний[2]. Область действия переменной i.0 есть любая программа, загружающаяся с данным файлом. Время существования i.0 — все время выполнения этой программы.

```
main()
{
```

Теперь мы находимся в блоке уровня 1.

```
auto int i=1;
```

i.1 = 1 (i относится к уровню 1).
Класс памяти переменной i.1 — auto. Область действия i.1 — функция main. Время существования i.1 — все время выполнения main.

---

[1] В любом месте текста программы *уровнем блока* называется число левых фигурных скобок ({) минус число правых фигурных скобок (}). Другими словами, это число текстуально открытых блоков. Самый внешний уровень в программе, т. е. когда еще нет ни одного блока, — это блок уровня 0.

[2] Вы можете спросить, почему класс памяти i не описывается явно с помощью служебного слова extern. Если только явно не описывается что-то другое, то класс памяти переменных, определяемых в блоке уровня 0, всегда считается extern. Связывание переменной со служебным словом extern не есть определение переменной. Это просто указание для транслятора, что эта переменная где-то в другом месте определена в блоке уровня 0.

```
PRINT1(d,i.1);
```
Если две переменные имеют одно и то же имя, то по этому имени обращаются к внутренней переменной, внешняя непосредственно недоступна.

```
{
```
Теперь мы находимся в блоке уровня 2.

```
int i=2;
```
i.2 = 2.
Класс памяти переменной i.2 — auto; таким по умолчанию будет класс памяти для переменных блоков уровня 1 и более. Область действия i.2 — блок уровня 2, время существования — время существования этого блока.

```
PRINT1(d,i.2);
```

```
{
```
Теперь мы находимся в блоке уровня 3.

```
i.2+=1;
```
i.2=3.

```
PRINT1(d,i.2);
```
Печатается i.2, т. е. самая внутренняя переменная с именем i.

```
}
```
Возвращаемся в блок уровня 2.

```
PRINT1(d,i.2);
```
Опять печатается i.2.

```
}
```
Переходим на уровень 1. Переменная i.2 исчезает.

```
PRINT1(d,i.1);
```
С ее исчезновением самой внутренней переменной с именем i будет i.1.

```
}
```
Возвращаемся на уровень 0.

Классы памяти 2: Функции

Классы памяти 2.1

```
int i=LOW;
```
i.0 = 0.

```
main()
{
auto int i=HIGH;
```
i.1 = 5.

```
reset(i.1/2);
```
Функция reset вызывается с параметром i.1/2, т. е. 2. Ее выполнение не затрагивает i.1.

```
PRINT1(d,i.1);
```

```
reset(i.1=i.1/2);
```
Опять обращаются к reset со значением i.1/2. На этот раз побочным эффектом от вызова функции будет присваивание i.1 значения 2, а reset опять не повлияет на i.1.

```
PRINT1(d,i.1);
```

```
i.1=reset(i.1/2);
```
Переменная i.1 получит значение, выдаваемое reset при обращении к ней с параметром i.1/2. Мы вставим в это место тело функции.

```
int reset(1)
```
Тип значения, выдаваемого функцией, указывается в ее описании. Функция reset возвращает значение типа int.

```
{ (int i=1;)
```
i.reset = 1.
Параметры функции можно рассматривать, как инициированные локальные переменные. Мы выделили эти неявные присваивания с помощью скобок.

```
i.reset = i.reset<=2 ? 5
```
i.reset = 5.

```
  return(i.reset);
  }
PRINT1(d,i.1)
```
Функция reset возвращает целое 5; так что i.1 = 5.

```
workover(i.1);
```
Функции workover передается значение i.1; сама i.1 вызовом функции не затрагивается. Мы вставим здесь тело функции, так как она содержит обращение PRINT.

```
workover(5)
```
Если не указано что-то другое, то функция возвращает значение типа int.

```
{ (int i=5;)
 i.workover = 0 * whatever
```
i.workover = 5.
i.workover = 0.

```
PRINT1(d,i.workover)

   return(i.workover);

   }
```
Функция workover возвращает 0, но обращающаяся к ней программа это игнорирует.

```
PRINT1(d,i.1);
}
```

Классы памяти 3: Снова функции

Классы памяти 3.1

```
int i=1;
```
i.0 = 1.

```
main()
{
auto int i,j;
```
Переменные i.1 и j.1 определяются, но значений у них нет.

```
i.1 = reset();
  reset()
  {

return(i.0);

}
```
В качестве значения i.1 получает результат, возвращаемый reset.

Поскольку reset не имеет параметров и локальных переменных с именем i, то появление i относится к i.0. Функция reset выдает 1, так что i.1 = 1.

```
for( j.1=1; j.1<3; j.1++ ) {
```
j.1 = 1.

```
PRINT2(d,i.1,j.1);
PRINT1(d,next(i.1));
  int next(1)
```

```
{ (int i=1;)
return( j.next=i.0++ );
  }
PRINT1(d,last(i.1));
   int last(1)
{ (int j=1;)
static int i=10;
return( j.last=i.last-- );
  }
PRINT1(d,new(i.1+j.1));
  int new(2)
{ (int i=2;)
int j=10;
return(i.new=j.new+=i.new);
}
for( j.1=1; j.1<3; j.1++ ) {
PRINT2(d,i.1,j.1);
PRINT1(d,next(i.1));
```

j.next = 1.
i.0 = 2, но next выдает 1, так как операция увеличения выполняется после того, как будет взято значение i.0.
Оператор возврата ссылается на i.0, так как в next нет других переменных с именем i. После выполнения оператора возврата j.next исчезает.

j.last = 1.
i.last = 10.
В last есть локальная переменная с именем i, инициируемая значением 10. Класс памяти i — static, а это означает, что i инициируется при загрузке программы и исчезает, когда заканчивается программа.

Переменная i.last = 9, но выдается значение 10, так как уменьшение переменной происходит после использования ее значения.

После возврата j.last исчезает, но i.last остается, и когда вновь обращаемся к last, то i будет равно 9.

i.new = 2.
j.new = 10.

j.new = 12, i.new = 12 и выдается значение 12.
После возврата j.new и i.new исчезают.

j.1 = 2.

Возвращаемся к оператору for. Для этого шага опишем в общих чертах действие каждого оператора.

Выполнение тела цикла приводит к увеличению j.1 на 1, но значение i.1 не меняется.

Функция next игнорирует значение передаваемого параметра и возвращает текущее значение i.0. Побочным эффектом выполнения next будет увеличение i.0 на 1.

```
PRINT1(d,last(i.1));
```

Функция last также игнорирует значение передаваемого параметра. Выдается текущее значение локальной статической переменной i.last. Побочный эффект выполнения last—уменьшение i.last на 1.

```
PRINT1(d,new(i.1+j.1));
```

Функция new выдает значение своего параметра плюс 10. Побочных эффектов нет.

```
}
}
```

Классы памяти 4: Файлы

Классы памяти 4.1

```
int i=1;
```

i.0 = 1

```
main()
{
auto int i,j;
i.1 = reset();
  extern int i;

reset()
{
return(i.0);
}
for( j.1=1; j.1<3; j.1++ ){
PRINT2(d,i.1,j.1);
PRINT1(d,next(i.1));
```

Описание extern сообщает транслятору, что i — внешняя переменная, определенная где-то в другом месте, вероятно, в другом файле. Здесь i относятся к i.0.

i.0 — внешняя переменная i, которая используется в reset.
i.1 = 1.
j.1 = 1.

```
  static int i=10;
```

Второй файл программы начинается с определения внешней переменной i. Кажется, что это определение вступает в конфликт с описанием внешней переменной в первом файле. Однако служебное слово static говорит транслятору, что эта переменная известна только в текущем файле. Другими словами, она доступна только в функциях next, last и new. Мы будем обозначать эту переменную как i.nIn; i.nIn = 10.

```
next()
{
```

В описании функции next нет никаких параметров, поэтому значение, передаваемое main, игнорируется.

```
  return(i.nIn+=1);
  }
PRINT1(d,last(i.1));
  last()
  {
```

Переменная i.nIn = 11 и next выдает 11.

```
return(i.nln-=1);
}
```

Переменная i.nln = 10 и last возвращает 10. В last используется то же i, которое ранее было увеличено функцией next.

```
PRINT1(d,new(i.1+j.1));
  new(2)
  { (int i=2;)
  static int j=5;
  return(i.new=j.new=5+2);
  }
for( j.1=1; j.1<3; j.1++ ) {
PRINT2(d,i.1,j.1);
PRINT1(d,next(i.1))
PRINT1(d,last(i.1));
PRINT1(d,new(i.1+j.1));
}
}
```

i.new = 2.

j.new = 5.

j.new = 7, i.new = 7 и выдается 7. Переменная i.nln не изменяется, i.new исчезает после возврата и j.new при новом обращении к new будет равно 7.

j.1 = 2.
Опишем в общих чертах действие каждого оператора в цикле.

Выполнение тела цикла приводит к увеличению j.1 на 1.

Функция next увеличивает i.nln и выдает получившееся значение.

Функция last уменьшает i.nln и выдает получившееся значение.

Функция new прибавляет свой параметр к j.new и выдает получившуюся сумму.

К главе 7. Ссылки и массивы

Ссылки и массивы 1: Простые ссылки и массивы

Ссылки и массивы 1.1

```
int a[] = {0,1,2,3,4};
for( i=0; i<=4; i++ )
PR(d,a[i]):
```

Переменная a определена как массив из пяти целых с элементами a[i] = i для i от 0 до 4.

Переменная i принимает значения от 0 до 4.

a[i] последовательно пробегает все элементы массива a.

Ссылки и массивы 1.2

```
int *p;
```

Описание вида type*x сообщает транслятору, что когда *x появляется в выражении, то это значение типа type. Перемен-

ная x — ссылка на переменные типа type и принимает в качестве значения адреса переменных типа type. type — базовый тип для x. В этой задаче переменная *p* описана как ссылка на целое, т. е. базовый тип p — int.

```
for( p= &a[0];
```

&a[0] дает адрес a[0].

```
p<=&a[4];
```

Элементы массива хранятся в памяти в порядке возрастания индексов, т. е. a[0] предшествует a[1], которое предшествует a[2], и т. д. Так что p, инициируемое значением &a[0] меньше, чем &a[4].

```
PR(d,*p);
```

Выражение *p дает целое, хранящееся по адресу, содержащемуся в p. Поскольку p содержит &a[0], то *p есть a[0].

```
p++ )
```

Если к ссылке применить операцию увеличения, то она будет указывать на следующий элемент базового типа. Фактически же значение ссылки увеличивается на sizeof (*базовый тип*). В языке не предполагается проверка того, что получившийся в результате адрес действительно есть адрес доступного элемента базового типа. В этой задаче p будет указывать на следующий элемент массива a.

```
p<=&a[4]
```

Значение p проверяется на конец массива a. Цикл заканчивается, когда p будет указывать за последний элемент массива. В теле цикла p последовательно указывает на элементы массива в порядке возрастания индексов.

Ссылки и массивы 1.3

```
for( p=&a[0],i=1; i<=5; i++ )
```

Переменная p указывает на начало массива a. Переменная i пробегает значения от 1 до 5.

```
PR(d,p[i]);
```

Ссылка p[i] последовательно указывает на элементы массива a. Ссылка p[5] указывает за массив a.

О массивах и индексах. Хотя до сих пор обозначение [ ] находило свое обычное применение—индексация массива, на самом деле [ ] обозначает общую операцию индексирования. x[i] определяется как *(x+i), где x обычно является адресом и i — целое. В соответствии с правилами адресной арифметики i должно быть кратно sizeof (*базовый тип* x). (К этому моменту должно быть ясно, почему массивы индексируются начиная с 0. Имя массива по сути есть ссылка на первый элемент массива. Индекс — это смещение относительно начала массива. Смещение первого элемента относительно начала массива равно 0.)

В предыдущем фрагменте i используется для индексации ссылки p. p [i] = *(p + i) = *(a + i) = a [i]. Переменная i пробегает значения от 1 до 5. Когда i = 5, p + i указывает за массив a, т. е. значение, находящееся по адресу p + i, неизвестно. Это настолько распространенная ошибка, что стоит подчеркнуть опять: *индекс массива из n элементов принимает значения от 0 до n — 1.*

Ссылки и массивы 1.4

`for( p=a,i=0;`	Ссылке p присваивается адрес первого элемента a.
`p+i <= a+4;`	p = a, i = 0, так что p + i = a + 0, что меньше, чем a + 4.
`PR(d,*(p+i));`	*(p + i) = *(a + 0) = a[0].
`p++, i++ )`	p указывает на второй элемент a, i равно 1.
`p+i <= a+4`	p = a + 1, i = 1, значит p + i = a + 2.
`PR(d,*(p+i));`	*(p+i) = a[2].
`p++, i++`	p = a+2, i = 2.
`p+i <= a+4`	p+i = a+4.
`PR(d,*(p+i));`	*(p+i) = a[4].
`p++, i++`	p = a+3, i = 3.
`p+i <= a+4`	p+i = a+6, и цикл заканчивается

Ссылки и массивы 1.5

`for( p=a+4;`	Ссылка p указывает на пятый элемент массива a.
`p >= a;`	Цикл заканчивается, когда p указывает за a.
`PR(d,*p);`	Печатается целое, на которое указывает p.
`p--`	При уменьшении p настраивается на предыдущий элемент массива.

Ссылки и массивы 1.6

`for( p=a+4,i=0; i<=4; i++ )`	Ссылка p указывает на последний элемент массива a, i пробегает от 0 до 4.
`PR(d,p[-i]);`	Печатается i-й от конца массива элемент.

Ссылки и массивы 1.7

```
for( p=a+4; p>=a; p-- )
```

Ссылка p последовательно указывает на все элементы массива a, от последнего до первого.

```
PR(d,a[p-a]);
```

Значение p — a равно смещению элемента массива, на который указывает p, от начала. Другими словами, p — a есть индекс элемента, на который указывает p.

Ссылки и массивы 2: Массив ссылок

Ссылки и массивы 2.1

```
int a[] = {0,1,2,3,4}
```

Переменная a инициируется как массив из пяти целых.

```
int *p[] = {a,a+1,a+2,a+3,a+4};
```

Если в выражении встречается *p[ ], оно вычисляется как целое, т. е. p[ ] должно указывать на целое, и p — массив ссылок на целое. Пять элементов массива p вначале указывают на пять элементов массива a.

```
int **pp = p;
```

Выражение **pp вычисляется как целое, значит, *pp должно быть ссылкой на целое, а pp — указывать на ссылку на целое. Вначале pp указывает на p[0].

Рис. 2.1. показывает взаимосвязь между pp, p и a.

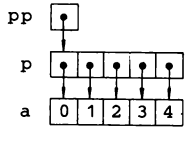

Рис. 2.1

Ссылки и массивы 2.2

```
PRINT2(d,a,*a);
```

Как уже указывалось ранее, имя массива — синоним адреса первого элемента массива. Значит, значение a — адрес начала массива a и *a эквивалентно a[0].

```
PRINT3(d,p,*p,**p);
```

Значение p считается адресом первого элемента массива p, *p дает значение первого элемента массива, т. е. p[0], а *pp выдает целое, находящееся по адресу, хранящемуся в p[0], т. е. значение a[0].

```
PRINT3(d,pp,*pp,**pp);
```

Параметр pp относится к содержимому pp, т. е. адресу p. *pp дает целое, на которое ссылается p[0], или a[0].

`pp++` — Переменная pp — это ссылка на ссылку на целое (базовый тип pp — ссылка на целое), так что pp++ настраивает pp на следующую ссылку в память. Действие операции ++ показано на рис. 2.3—1 жирной стрелкой.

`pp-p` — Ссылка pp указывает на второй элемент массива p, т. е. на p[1]. Таким образом, значение pp есть p+1. pp−p=(p+1)−p, т. е. 1.

`*pp-a` — Ссылка pp указывает на p[1] и *pp указывает на второй элемент массива a. Таким образом, значение pp есть a+1, a *pp−a=(a+1)−a.

`**pp` — Выражение *pp указывает на a[1], так что *pp дает содержимое a[1].

`*pp++` — *(pp++)
Унарные операции выполняются справа налево, поэтому вначале выполняется операция ++, а затем — косвенное обращение. Жирная стрелка на рис. 2.3—2 показывает действие операции ++.

`*++pp` — *(++pp) (рис. 2.3—3).

`++*pp` — ++(*pp) (рис. 2.3—4).

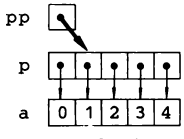

Рис. 2.3—1

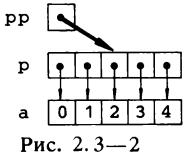

Рис. 2.3—2

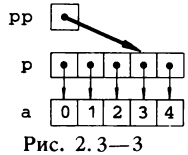

Рис. 2.3—3

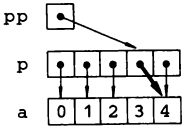

Рис. 2.3—4

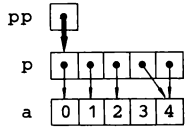

Рис. 2.4—1 pp=p

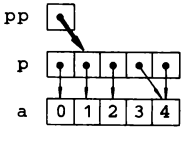

Рис. 2.4—2** (pp++))

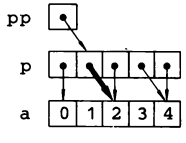

Рис. 2.4—3* (++(*pp))

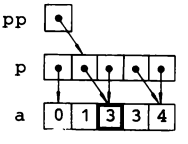

Рис. 2.4—4++ (*(*pp))

Ссылки и массивы 3: Многомерный массив


```
int a[3][3] = {
   { 1,2,3 },
   { 4,5,6 },
   { 7,8,9 }
};
```

Переменная a — матрица 3×3 со строками 1 2 3, 4 5 6 и 7 8 9.

Выражение a [i] [j] относится к целому, находящемуся на расстоянии j от начала строки i. Значение a есть адрес первой строки матрицы a. Таким образом, a — ссылка на массив целых из трех элементов и a [ ] — ссылка на целое.

```
int *pa[3] = {
   a[0],a[1],a[2]
};
```

Выражение *pa [ ] приводится к целому, значит, pa [ ] — ссылка на целое, а pa — массив ссылок на целое. Элемент pa [0] инициируется первым элементом из первой строки a, pa [1] инициируется первым элементом из второй строки, pa [2] — первым элементом третьей строки.

```
int *p = a[0];
```

Ссылка p есть ссылка на целое, первоначально указывающая на первый элемент первой строки матрицы a.

Рис. 3.1 показывает взаимосвязь между a, pa, и p.

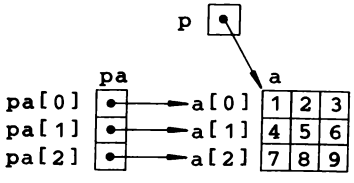

Рис. 3.1

Ссылки и массивы 3.2

```
for(i=0; i<3; i++)
a[i][2-i]
```

В цикле i принимает значения от 0 до 2. Выражение a [i] [2 — i] выбирает диагональные элементы от a [0] [2] до a [2] [0].

```
*a[i]
```

a [i] — адрес первого элемента i-й строки матрицы a.

```
*(*(a+i)+i)
```

*a [i] — значение первого элемента i-й строки.
a + i — адрес i-й строки a, *(a + i) — адрес первого элемента i-й строки, *(a + i) + i — адрес i-го элемента из i-й строки. Наконец, (* (a + i) + i) дает целое значение i-го элемента i-й строки.

Ссылки и массивы 3.3

```
for( i=0; i<3; i++ )
pa[i]
```

В цикле i принимает значения от 0 до 2. pa [i] относится к i-му элементу из pa, *pa [i] дает целое, на которое указывает i-й элемент pa.

```
p[i]
```

Ссылка p указывает на первый элемент первой строки матрицы a. Так как базовый тип ссылки p—int, то p[i] дает i-й элемент первой строки из a.

Об адресах массива. Мы уже несколько раз указывали, что адрес массива и адрес первого элемента массива есть одно и то же значение. В предыдущей задаче мы видели, что a и a [0] сводятся к одному и тому же адресу. Единственное различие между адресом массива и адресом первого элемента массива заключается в *типе* адреса, а значит, в едини-

це измерения адресной арифметики. Таким образом, поскольку тип a — ссылка на массив целых из трех элементов, базовый тип a — массив целых из трех элементов, и a + 1 указывает на следующий в памяти массив из трех чисел. В то же время тип a [0] — ссылка на целое, базовый тип a [0] — int и a [0] + 1 указывает на следующее в памяти целое.

Ссылки и массивы 4: Хитросплетение ссылок

Ссылки и массивы 4.1

```
char *c[] = {
   "ENTER",
   "NEW",
   "POINT",
   "FIRST"
};
```

Описатель *c [ ] приводится к символу, значит, c [ ] указывает на символ и c — массив ссылок на символ. Элементы c инициируются так, что они указывают на массивы символов «ENTER», «NEW», «POINT» и «FIRST».

```
char **cp[] = {
   c+3,c+2,c+1,c
};
```

Описатель **cp [ ] соответствует символу, *cp — ссылке на символ и cp [ ] — ссылке на ссылку на символ. Элементы cp инициируются так, что они указывают на элементы c.

```
char ***cpp = cp;
```

Описатель ***cpp дает символ, **cpp — ссылку на символ, *cpp — ссылку на ссылку на символ, наконец, cpp — ссылка, указывающая на ссылку на ссылку на символ.

Рис. 4.1 показывает взаимосвязь между cpp, cp и c.

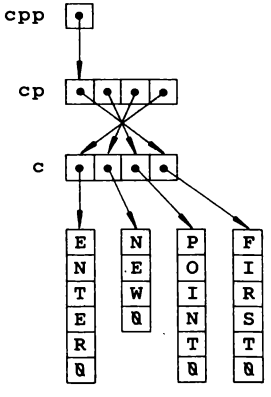

Рис. 4.1

Ссылки и массивы 4.2

```
*(*(++cpp))
```

Увеличим cpp, а затем проследим цепочку ссылок (рис. 4.2—1).

```
(*(--(*(++cpp))))+3
```

Увеличим cpp, по ссылке дойдем до cp [2], уменьшим cp [2], по ссылке дойдем до c [0] и эту ссылку проиндексируем значением 3 (рис. 4.2—2).

```
(*(cpp[(-2)]))+3
```

Индексируя cpp значением —2, получим cp [0], по ссылке дойдем до c [3] и проиндексируем его значением 3 (рис. 4.2—3).

```
((cpp[-1])[-1])+1
```

Индексируя cpp значением —1, получим cp [1], снова индексируем с —1 и доходим до c [1], эту последнюю ссылку индексируем значением 1 (рис. 4.2—4).

О ссылках. Если вы правильно решили последнюю задачу, то знаете все, что нужно знать о механизме использования ссылок. Сила ссылок заключается в их универсальности: ссылки можно связывать друг с другом, получая бесконечно разнообразные сложные структуры данных. Но опасность ссылок как раз и заключается в их силе: сложные цепочки ссылок редко бывают понятными и еще реже верными.

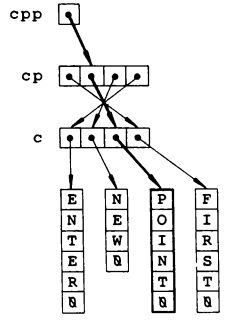

Рис. 4.2—1

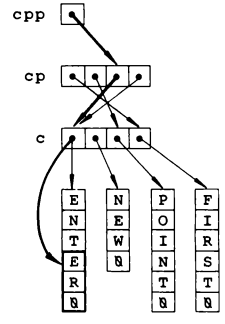

Рис. 4.2—2

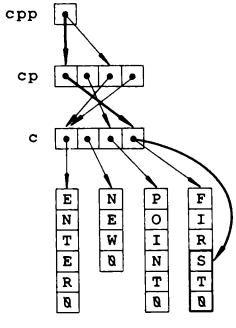

Рис. 4.2—3

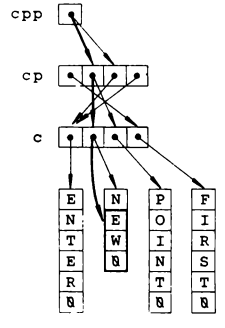

Рис. 4.2—4

К главе 8. Записи

Записи 1: Простые записи, вложенные записи

Записи 1.1

```
static struct S1 {
   char c[4], *s;
} s1 = { "abc", "def" };
```

Тип записи S1 относится к записи, содержащей массив символов с длиной 4 и ссылку на символ s. Переменная s1 — это некоторый экземп-

ляр записи S1, инициируемый следующим образом:

```
char   c[4]"abc",
       *s="def"
```

Поскольку запись была определена как static, то ее можно инициировать при определении.

```
static struct S2 {
   char *cp;
   struct S1 ss1;
} s2 = { "ghi", { "jkl", "mno" } };
```

Тип записи S2 относится к записи, содержащей ссылку на символ cp и экземпляр записи типа S1 — ss1. Переменная — запись s2 есть экземпляр записи типа S2, инициированной следующим образом:

```
char *cp="ghi";
struct S1 ss1=
   {"jkl", "mno"};
```

Рис. 1.1 изображает записи s1 и s2.

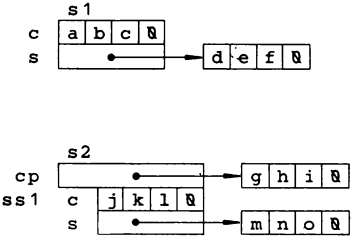

Рис. 1.1

Записи 1.2

```
PRINT2(c,
(s1.c)[0]
*(s1.s)
```

Нужно напечатать какой-то символ.

Это первый символ из элемента c записи s1 (рис. 1.2—1)
Это символ, на который указывает элемент s записи s1 (рис. 1.2—2).

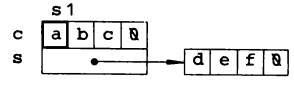

Рис. 1.2—1

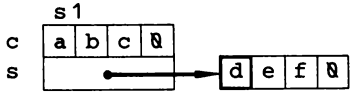

Рис. 1.2—2

Записи 1.3

```
PRINT2(s,
s1.c
```

Нужно напечатать некоторую строку.

Это строка, на которую указывает элемент c записи s1. Напомним, что c=&c[0] (рис. 1.3—1).

`s1.s` Это строка, на которую указывает элемент s записи s1 (рис. 1.3—2).

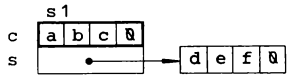

Рис. 1.3—1

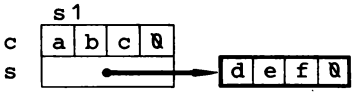

Рис. 1.3—2

Записи 1.4

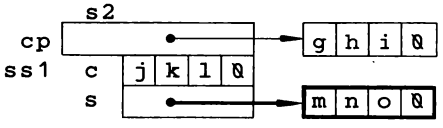

Рис. 1.4—1 s2 . cp

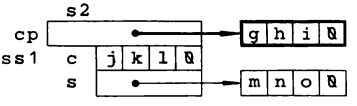

Рис. 1.4—2 (s2. ss1).s

Записи 1.5

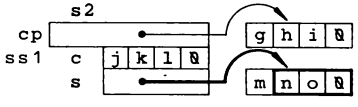

Рис. 1.5—1 + + (s2 . cp.)

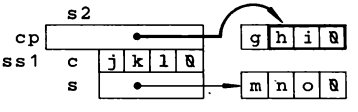

Рис. 1.5—2 + + ((s2. ss1). s)

Записи 2: Массив записей

Записи 2.1

```
struct S1  {
   char*s;
   int i;
   struct S1*s1p;
};

static struct S1 a[] = {
   { "abcd", 1, a+1 },
   { "efgh", 2, a+2 },
   { "ijkl", 3, a }
};
struct S1 *p=a;
```

Тип записи S1 относится к записи, содержащей ссылку на символ s, целое i и ссылку s1p на запись типа S1. Но это только описание, никакого экземпляра записи типа S1 не определяется.

Здесь a — массив из трех элементов типа S1, т. е. записей S1. Поскольку a определялся как static, то его можно инициировать в самом определении.

Переменная p описывается как ссылка на запись S1 и инициируется значением, указывающим на первый элемент a.

На рис. 2.1. изображены массив a и ссылка p.

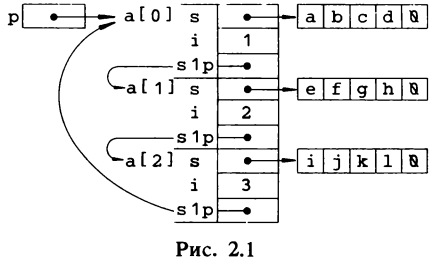

Рис. 2.1

Записи 2.2

`PRINT3(s,` — Должны быть напечатаны строки.

`(a[0]).s` — Это строка, на которую указывает элемент записи, являющейся первым элементом массива a (рис. 2.2—1).

`p->s` — Это строка, на которую указывает элемент s записи, на которую указывает p (рис. 2.2—2).

`(((a[2]).s1p)->)s` — Это строка, на которую указывает элемент s записи, на которую, в свою очередь, указывает элемент s1p записи, являющейся третьим элементом a (рис. 2.2—3).

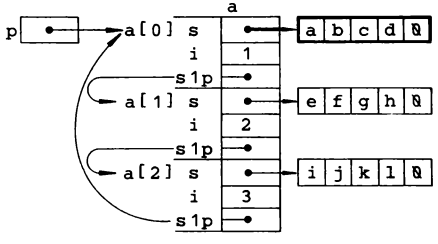

Рис. 2.2—1

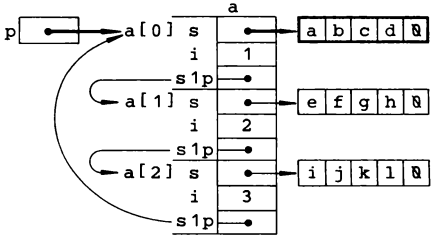

Рис. 2.2—2

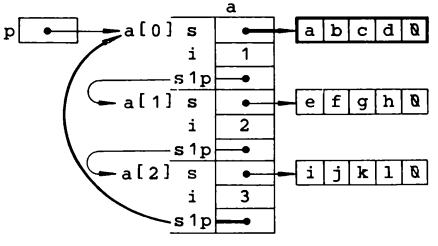

Рис. 2.2—3

Записи 2.3

`for( i=0; i<2; i++ ) {` — Переменная i принимает значения 0 и 1.

`PR(d,` — Нужно напечатать некоторое целое.

`--((a[i]).i)` — Операция уменьшения применяется к целому, хранящемуся в элементе i записи, которая есть i-й элемент a. (На рис. 2.3—1 показан случай для i=0.)

`PR(c,` — Нужно напечатать символ.

`++(((a[i]).s)[3])` — Операция увеличения применяется к четвертому символу строки, на которую указывает элемент записи, которая есть

i-й элемент a. (На рис. 2.3—2 показан случай для i=0.)

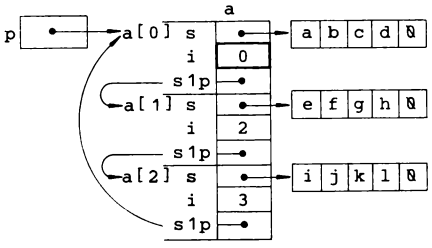

Рис. 2.3—1

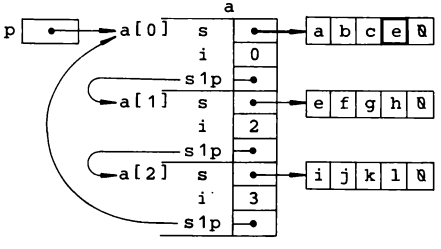

Рис. 2.3—2

Записи 2.4

`++(p->s)`

Увеличивается элемент s записи, на которую указывает p, затем печатается строка, на которую указывает этот элемент (рис. 2.4—1).

`a[((++p)->i)].s`

Вначале увеличивается p, затем печатается элемент s записи, являющейся элементом массива a с индексом p—>i (рис. 2.4—2).

`a[--((p->s1p)->i)].s`

Сначала уменьшается элемент i записи, на которую указывает элемент s1p из записи, на которую указывает p. Затем результат используется как индекс для a (рис. 2.4—3).

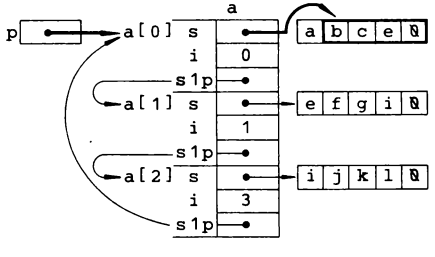

Рис. 2.4—1

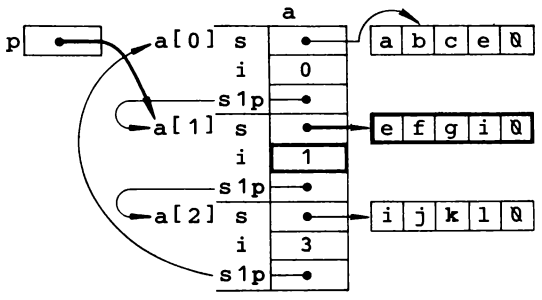

Рис. 2.4—2

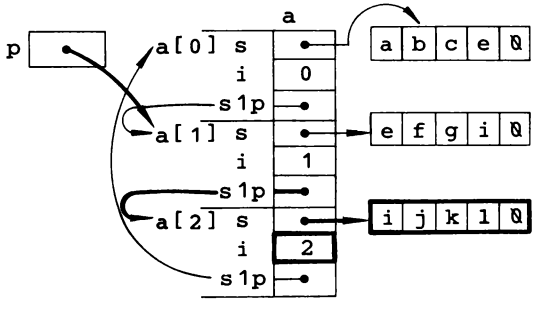

Рис. 2.4—3

Записи 3: Массив ссылок на записи

Записи 3.1

```
struct S1 {
   char *s;
   struct S1 *s1p;
};
static struct S1 a[] = {
   { "abcd", a+1 },
   { "efgh", a+2 },
   { "ijkl", a }
};
struct S1 *(p[3]);
```

Тип записи S1 относится к записи, содержащей ссылку на символ s, и ссылку на запись типа S1, s1p.

Массив a из трех элементов, представляющих собой записи типа S1, определен как статический, поэтому в определении его можно инициировать.

Выражение*(p[ ]), если оно встречается в операторе программы, относится к записи S1. Значит p[ ] — ссылка на запись S1 и p — массив из трех элементов, указывающих на записи типа S1.

На рис. 3.1 изображены массивы a и p.

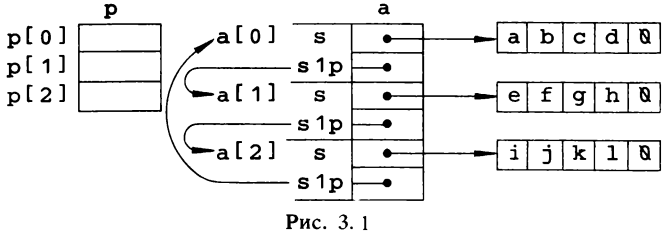

Рис. 3. 1

Записи 3.2

`for( i=0; i<3; i++ )` — i принимает значения 0, 1, 2.

`p[i] = (a[i]).s1p;` — В i-й элемент p копируется ссылка из элемента s1p записи, которая есть i-й элемент a (рис. 3.2—2).

`(p[0])->s, (*p)->s, (**p).s` — Одно и то же можно сказать по-разному (рис. 3.2—2).

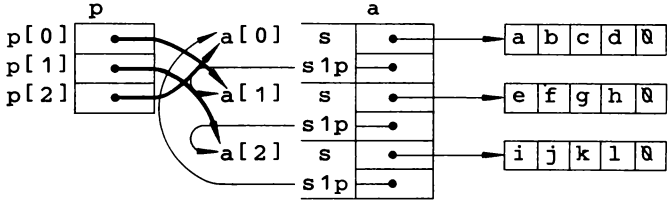

Рис. 3.2.—1

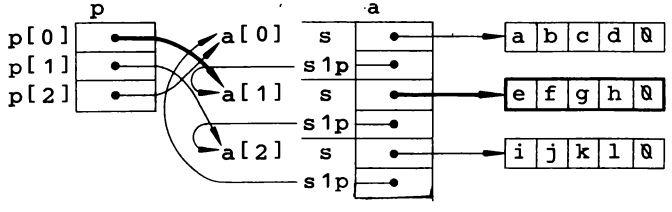

Рис. 3.2.—2

Записи 3.3

`swap(*p,a);` — Так как p указывает на p[0], то*p относится к содержимому p[0] или &a[1], а дает &a[0].

`temp = (&a[1])->s;` — Эквивалентно temp=a[1].s

`(&a[1])->s = (&a[0])->s` — Или a[1].s=a[0].s.

`(&a[0])->s = temp` — Функция swap меняет местами строки, на которые указывают элементы s ее аргументов (рис. 3.3—1).

`(p[0])->s, (*p)->s` — (рис. 3.3—2)

`((*p)->s1p)->s` — (рис. 3.3—3)

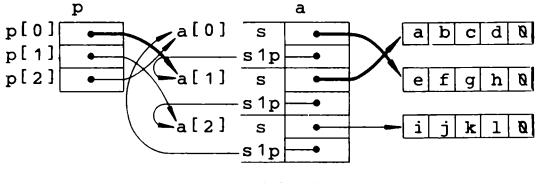

Рис. 3.3—1

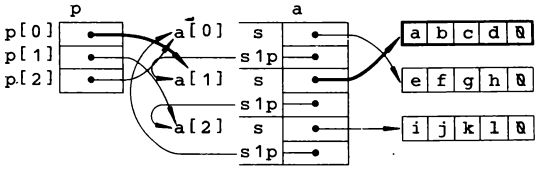

Рис. 3.3—2

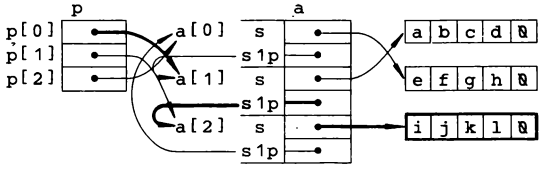

Рис. 3.3—3

Записи 3.4

```
swap(p[0], p[0]->s1p);
```

Элемент p [0] содержит &a [1], (p [0])→slp содержит &a [2] (рис. 3.4—1).

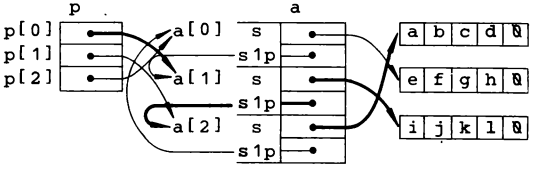

Рис. 3.4—1

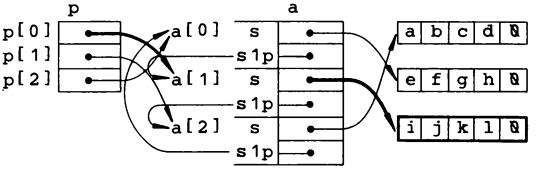

Рис. 3.4—2(p [0]) — > s

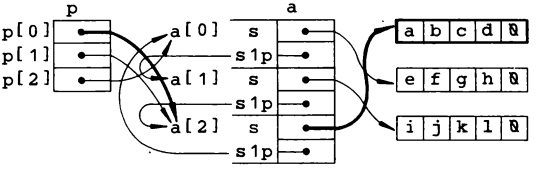

Рис. 3.4 —3 (*(++(p [0]))).s

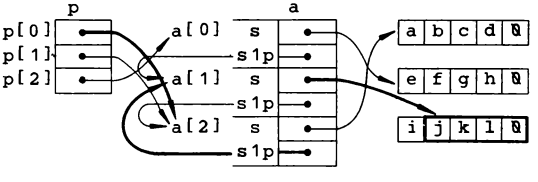

Рис. 3.4—4 ++((*(++((*p)— >slp))).s)

Препроцессор 1: Препроцессор не знает Си

Препроцессор 1.1

```
int x=2
PRINT( x*FUDGE(2) );
```

Чтобы понять эффект макроподстановки препроцессора, нужно провести ее в месте вызова.

```
PR(a); putchar('\n')
```

Всегда производится самая левая макроподстановка.

Сначала вызов заменяется строкой подстановки.

```
PR( x*FUDGE(2) ); putchar('\n')
```

Затем аргументы заменяются на соответствующие строки.

```
printf("a= %d\t",(int)(a))
```

Опять производится самая левая макроподстановка, на этот раз PR.

```
printf(" x*FUDGE(2) = %d\t",
    (int)(x*FUDGE(2)))
```

Подставляются аргументы макроподстановки.

```
printf(" x*FUDGE(2) = %d\t",
    (int)(x*k+3.1459))
```

Имя макроподстановки, встречающееся в строке, не заменяется, но аргументы макроподстановки, встречающиеся в строке подстановки, заменяться должны. Значит, a в макроподстановке PR заменяется на x*FUDGE(2), но FUDGE(2), встречающееся при задании формата печати в обращении к printf, не заменяется.

```
(int)(x*2+3.14159)
```

Заменяя формальный параметр на фактический, получаем неожиданный результат. Сначала умножаем, затем складываем и отбрасываем дробную часть.

Внимание! Макроподстановки могут быть источником трудно уловимых ошибок. Макроподстановка — это только замена одних строк на другие. Препроцессор не знает ничего о языке Си. Поэтому многих неожиданных результатов можно избежать, если строго следовать нескольким правилам.

Правило 1. Заключайте в скобки строки-подстановки, если они содержат операции.

Нежелательного взаимодействия между строкой-подстановкой и контекстом в приведенной задаче могло бы и не быть, если бы FUDGE(k) определялось как (k + 3.14159).

Препроцессор 1.2

```
for(cel=0; cel<=100; cel+=50)
     PRINT2( cel,9./5*cel+32 );
```

```
for(cel=0; cel<=100; cel+=50)
     PR( cel);
PRINT( 9./5*cel+32 );
```

Вначале производим макроподстановку PRINT2.

```
for(cel=0; cel<=100; cel+=50)
     printf(" cel= %d\t",(int)(cel));
PRINT( 9./5*cel+32 );
```

Затем производим макроподстановку PR.

```
for(cel=0; cel<=100; cel+=50)
     printf(" cel= %d\t",(int)(cel));
PR( 9./5*cel+32 ); putchar('\n');
```

Производим макроподстановку PRINT.

```
for(cel=0; cel<=100; cel+=50)
     printf(" cel= %d\t",(int)(cel));
printf(" 9./5*cel+32 =%d\t",
     (int)(9./5*cel+32));
putchar('\n');
```

Производим макроподстановку PR.

Обращение к PRINT2 выглядит как один оператор, но после макроподстановки появляются три. Только первое обращение к PR оказывается внутри цикла for. Второе обращение к PR происходит после выполнения цикла for со значением cel = 150.

Правило 2. Не давайте расползаться макроподстановке; лучше использовать выражение, а не оператор, и не несколько операторов, а один-единственный.

В данной задаче, чтобы удовлетворить этому правилу, надо использовать в теле PRINT вместо точек с запятой запятые.

Препроцессор 1.3

```
int x=1, y=2;
PRINT3( MAX(x++,y),x,y );
(a<b ? b : a),x,y
```

Макроподстановка PRINT3, конечно же, происходит прежде MAX. Однако, дабы не затемнять сути, в этой и последующей задачах подстановка PRINT не будет производиться. Тогда первый шаг состоит в подстановке строки вместо MAX.

```
(x++<y ? y : x++),x,y
```

Затем заменяем аргументы макроподстановки на формальные.

```
(1<2 ? y : x++), and x=2
(y)
2
```

Наконец, производим вычисления.

```
PRINT3( MAX(x++,y),x,y );
(x++<y ? y : x++),x,y
(2<2 ? y : x++), and x=3
(x++)
3, and x=4
```

Теперь выполняем второе обращение к PRINT3.

При обращении x++ появляется только один раз, но в расширении фигурирует уже два раза, что приводит к увеличению x иногда на 1 и иногда на 2. Тяжесть ответственности за защиту от таких нежелательных побочных эффектов можно возлагать или на того, кто определяет макроподстановку, или на того, кто ею пользуется.

Правило 3. Избегайте макроподстановок, которые могут привести к непредсказуемым или несообразным побочным эффектам.

Правило 3А. В обращениях к макроподстановкам избегайте выражений, содержащих побочные эффекты.

В общем случае проблема побочных эффектов достаточно хитрая. Следование правилу 3 часто означает копирование аргументов в локальные переменные подстановки; эта добавочная работа уменьшает преимущество (в скорости) макроподстановок перед обращением к функциям. Следование же правилу 3А требует знания того, какая подпрограмма реализована как макроподстановка, а какая как функция. В лучшем случае это нарушает представление о подпрограмме как некой абстракции, а в худшем случае может привести к ошибке, если данная подпрограмма будет реализована по-другому.

В данной задаче следование правилу 3А не затронет саму MAX.

Препроцессор 2: Осторожность вознаграждается

Препроцессор 2.1

```
int x=1;
PRINT( -NEG(x) );
```

```
--a
```

Вначале подставляется строка подстановки. (Как и прежде, макроподстановка PRINT не производится.)

```
--x, and x=0
```

Затем элемент в этой строке заменяется на аргумент обращения.

Строку подстановки образуют точно те символы, которые следуют сразу за закрывающей скобкой списка аргументов. Изюминка данной задачи состоит в том, что —a сразу следует за закрывающей скобкой. Если, следуя правилу 1, определить NEG(a) как (—a), то получим нужный результат. Кроме того, лучше начинать строку подстановки с пробела или символа табуляции.

Препроцессор 2.2

```
PRINT( weeks(10080) )
(days(10080)/7)
```

Заменяем каждое обращение на соответствующую строку. Заметим, что здесь нет никакого конфликта между параметром mins и именем макроподстановки mins.

```
((hours(10080)/24)/7)
(((10080/60)/24)/7)
1
```

Вычисляем.

```
PRINT( days(mins(86400)) )
(hours(mins(86400))/24)
((mins(86400)/60)/24)
(((86400/60)/60)/24)
1
```

Производим самую левую макроподстановку.

Вычисляем.

Препроцессор 2.3

```
static char input = "\twhich\if?";

if(c<' ') TAB(c,i,oldi,temp);
else putchar(c);


if(c<' ')
      if(c=='\t')
            for(temp=8-(i-oldi-1)%8,oldi=i; temp; temp--)
                  putchar(' ');
      else putchar(c);
```

TAB включает «открытый» условный оператор. Поэтому после подстановки он «поглощает» оператор, следующий за ближайшим else.

Правило 4. Подстановка макроопределения должна быть законченной конструкцией языка Си — выражением, оператором (без последней точки с запятой) или блоком.

В этой задаче трудность устраняется добавлением в определение TAB «пустого» оператора else.

О функциях и макроподстановках. Часто подпрограмму можно реализовать и с помощью функции, и с помощью макроподстановки. Преимущество использования макроподстановки в том, что программа работает быстрее, поскольку нет обращений к функциям. Использование же функции гарантирует, что не возникают всякие неприятные ситуации наподобие тех, которые рассматривались в предыдущих задачах. Кроме того, в случае функций, если они вызываются неоднократно, потребуется, вероятно, меньше памяти. Это подводит нас к заключительному правилу по использованию макроподстановок.

Правило 5. Макроопределение должно быть простым; если это не удается, используйте функцию.

ПРИЛОЖЕНИЯ

Приложение I. Таблица приоритетов

Операция	Порядок выполнения
первичные: ( ) [ ] — >.	слева направо
унарные: ! ~ ++ ——— *(тип)** & sizeof	справа налево
мультипликативные: * / %	слева направо
аддитивные : + —	слева направо
сдвиги: << >>	слева направо
отношения: <<= >>=	слева направо
сравнения на равенство: == !=	слева направо
поразрядные: &	слева направо
поразрядные: ^	слева направо
поразрядные: |	слева направо
логические: &&	слева направо
логические ||	слева направо
условие: ? :	справа налево
присваивание: = += —= и т. д.	справа налево
запятая: ,	слева направо

Таблица приоритетов показывает относительный приоритет операций. Приоритет определяет порядок, в котором операции сопоставляются с операндами. Операции получают свои операнды в порядке убывания их приоритетов.

Чтобы определить относительный приоритет операций в некотором выражении, нужно найти эти операции в столбце таблицы, озаглавленном «Операция». Операция, занимающая более высокую строку в этом столбце, будет иметь и более высокий приоритет. Если операции находятся на одной строке, то посмотрите графу «Порядок выполнения». Если там сказано «слева направо», то операция, стоящая в выражении левее, имеет более высокий приоритет; если сказано «справа налево» — то наоборот.

Приложение 2. Сводная таблица операций

Арифметические операции (операнды — числа или ссылки)

Аддитивные

Операция	*Результат*	*Ограничения*
x+y	сумма x и y	если один из операндов ссылка, то другой должен быть целой величиной[1]
x—y	разность x и y	если один из операндов ссылка, то другой — целая величина или ссылка с тем же базовым типом

Мультипликативные

Операция	*Результат*	*Ограничения*
x*y	произведение x на y	x и y не должны быть ссылками
x/y	частное от деления x на y	x и y не должны быть ссылками
x%y	остаток от деления x на y	x и y не должны быть типов double, float или ссылка
—x	изменение знака x	x не должно быть ссылкой

[1] Точнее, величиной типа int, char, short, long или unsigned.

Увеличения и уменьшения

Операция	Результат	Ограничения
x++(x--)	x после использования x увеличивается (уменьшается)	x относится к некоторому числовому значению или к ссылке
++x (--x)	x + 1 (x — 1) x увеличивается (уменьшается) перед использованием	x относится к некоторому числовому значению или к ссылке

Операции присваивания

Операция	Результат	Ограничения
x=y	y приводится к типу x, x получает значение y	x, y могут быть любого типа, но не массивом
x *op*=y	x *op* (y) приводится к типу x, x получает значение x *op* (y)	x, y могут быть любого типа, но не массивом или записью

Поразрядные операции (операнды — целые величины)

Логические

Операция	Результат	Ограничения
x & y	поразрядное логическое И для x на y; соответствующий разряд результата равен 1, если оба разряда x и y равны 1 и 0, иначе	
x | y	поразрядное логическое ИЛИ; соответствующий разряд результата равен 0, если оба разряда x и y равны 0 и 1, иначе	
	поразрядное логическое исключающее ИЛИ; соответствующий разряд резуль-	

x ^ y	тата равен 0, если разряды x и y одинаковы и 1, иначе
~x	логическое отрицание (дополнение); разряд, равный 1. в x, равен 0 в результате, и наоборот

Сдвиг

Операция	Результат	Ограничения
x<<y	сдвиг влево на y разрядов, младшие разряды x равны 0	y положительно и меньше разрядности машины
x>>y	сдвиг x вправо на y разрядов; старшие разряды x будут равны 0 для положительных x и равны 0 или 1 (в зависимости от транслятора) для отрицательных x	y положительно и меньше разрядности машины

Логические операции (операнды — числа и ссылки)

Операция	Результат	Ограничения
x && y	логическое И: 1, если x и y не равны 0, 0 — иначе	результат типа int
x||y	логическое ИЛИ: 0, если x и y равны 0, 1 — иначе	результат типа int
! x	логическое отрицание x: 0, если x не равен 0, 1 — иначе	результат типа int

Сравнения (операнды — числа и ссылки)

Отношения

Операция	Результат	Ограничения
x<y (x>y)	1, если x меньше (больше) y, 0—иначе	результат типа int
x<=y x>=y	1, если x меньше или равно (больше или равно) y, 0 — иначе	результат типа int

Сравнение на равенство

Операция	Результат	Ограничения
x==y (x!=y)	1, если x равно (не равно) y, 0—иначе	результат типа int

Условие

Операция	Результат	Ограничения
x ? y : z	y, если x не равно 0, z — иначе	

Адресные операции

Операция	Результат	Ограничения
* x	значение по адресу, хранящемуся в x, приведенное к базовому типу x	x должно быть ссылкой
& x	адрес x	x должно указывать на некоторое значение
x [y]	результат — значение, находящееся по адресу x+y, приведенное к типу, требуемому операндом	один операнд должен быть адресом, а другой — целой величиной
x.y	результат — значение элемента y записи x	x должно быть записью, а y — элемент этой записи
x➘ y	результат — значение элемента y записи с адресом x	x должно быть ссылкой на запись, а y — элементом этой записи

Операции работающие с типом

Операция	Результат	Ограничения
*(тип)*x	x преобразуется к типу *тип*	x — любое выражение
sizeof x	размер x в байтах	x — любое выражение
sizeof *(тип)*	размер в байтах объекта типа *тип*	x — любое выражение

Операция запятая

Операция	Результат	Ограничения
x, y	y x вычисляется прежде y	x, y — любые выражения

Приложение 3. Таблица кодов

В восьмеричном виде

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	$	045	%	046	&	047	'
050	(	051	)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174	¦	175	}	176	~	177	del

В шестнадцатеричном виде

00	nul	01	soh	02	stx	03	etx	04	eot	05	enq	06	ack	07	bel
08	bs	09	ht	0a	nl	0b	vt	0c	np	0d	cr	0e	so	0f	si
10	dle	11	dc1	12	dc2	13	dc3	14	dc4	15	nak	16	syn	17	etb
18	can	19	em	1a	sub	1b	esc	1c	fs	1d	gs	1e	rs	1f	us
20	sp	21	!	22	"	23	#	24	$	25	%	26	&	27	'
28	(	29	)	2a	*	2b	+	2c	,	2d	-	2e	.	2f	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3a	:	3b	;	3c	<	3d	=	3e	>	3f	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4a	J	4b	K	4c	L	4d	M	4e	N	4f	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5a	Z	5b	[	5c	\	5d	]	5e	^	5f	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6a	j	6b	k	6c	l	6d	m	6e	n	6f	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7a	z	7b	{	7c	¦	7d	}	7e	~	7f	del

ASCII (American Standard Code for information interchange) отображает множество управляющих и печатных символов в множество чисел, состоящих из 7 двоичных цифр. Таблицы показывают соответствие между символом и его значением. Вообще говоря, символы со значениями меньше восьмеричного 040 (шестнадцатеричного 020) считаются управляющими символами и не печатаются, хотя символы табуляции, конца строки и т. д. приведены здесь. Символы со значениями начиная с 040 — это обычные известные символы. Цифры и буквы упорядочены естественным образом, т. е. значение '1' меньше значения '2', значение 'A' меньше значения 'B' и т. д.

Приложение 4. Схема иерархии типов

```
double  ←  float
  ↑
long
  ↑
unsigned
  ↑
int  ←  char, short
```

Схема иерархии типов показывает упорядоченность арифметических типов. Тип результата каждой арифметической операции выражения есть тип того ее операнда, который имеет в соответствии с приведенной схемой более высокий тип. Аналогично, если сравниваются два значения, то значение, имеющее низший тип, преобразовывается к высшему согласно приведенной схеме. Упорядоченность типов на схеме показывают вертикальные стрелки: самый высший тип — double, самый низший — int. Горизонтальные стрелки задают порядок автоматического приведения типов в выражении. А именно операнды типа float всегда преобразуются перед использованием в значение типа double, а операнды типа char или short преобразуются в значение типа int.

ОГЛАВЛЕНИЕ

Брайан Керниган,
Деннис Ритчи,
Алан Фьюэр

Язык программирования Си. Задачи по языку Си

Книга одобрена на заседании секции редсовета
по электронной обработке данных в экономике
11 марта 1983 г.

Зав. редакцией *А. В. Павлюков*
Редактор *О. Б. Степанченко*
Мл. редактор *И. В. Щербакова*
Техн. редакторы *Г. А. Полякова, И. В. Завгородняя*
Корректоры *Я. Б. Островский,*
Е. Ю. Потапкина и И. П. Елкина
Художн. редактор *О. Н. Поленова*
Переплет художника *Б. С. Вехтера*

ИБ № 1403

Сдано в набор 1.08.84. Подписано в печать 27.02.85.
Формат 70×100 1/16. Бум. офсетная № 2. Гарнитура тип. Таймс.
Печать офсетная. Усл. п. л. 22,75. Усл. кр.-отт. 22,75.
Уч.-изд. л. 18,17. Тираж 15000 экз. Заказ 395. Цена 1 р. 50 к.

Издательство «Финансы и статистика»,
101000, Москва, ул. Чернышевского, 7.
Типография № 4 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 129041, Москва, Б. Переяславская ул., 46.