Язык программирования С

Б.В. Керниган, Д.М. Ричи.

Brian W.Kernighan and Dennis M.Ritchie
The C Programming Language

Предисловие

Язык **C** (произносится «си») – это универсальный язык программирования, для которого характерны экономичность выражения, современный поток управления и структуры данных, богатый набор операторов. Язык **C** не является ни языком «очень высокого уровня», ни «большим» языком, и не предназначается для некоторой специальной области применения, но отсутствие ограничений и общность языка делают его более удобным и эффективным для многих задач, чем языки, предположительно более мощные.

Язык **C**, первоначально предназначавшийся для написания операционной системы UNIX на ЭВМ DEC PDP-11, был разработан и реализован на этой системе Деннисом Ричи. Операционная система, компилятор с языка **C** и по существу все прикладные программы системы UNIX (включая все программное обеспечение, использованное при подготовке этой книги) написаны на **C**. Коммерческие компиляторы с языка **C** существуют также на некоторых других ЭВМ, включая IBM SYSTEM/370, HONEYWELL 6000, INTERDATA 8/32. Язык **C**, однако, не связан с какими-либо определёнными аппаратными средствами или системами, и на нем легко писать программы, которые можно пропускать без изменений на любой ЭВМ, имеющей **C**-компилятор.

Эта книга предназначена для того, чтобы помочь читателю научиться программировать на языке **C**. Она содержит учебное введение, цель которого – позволить новым пользователям начать программировать как можно быстрее, отдельные главы по всем основным особенностям языка и справочное руководство. Обучение построено в основном на чтении, написании и разборе примеров, а не голой формулировке правил. Примеры, приводимые в книге, по большей части являются законченными реальными программами, а не отдельными фрагментами. Все примеры были проверены непосредственно с текста книги, где они напечатаны в виде, пригодном для ввода в машину. Кроме указаний о том, как сделать использование языка более эффективным, мы также пытались, где это возможно, проиллюстрировать полезные алгоритмы и принципы хорошего стиля и разумной разработки.

Настоящая книга не является вводным курсом в программирование; она предполагает определённое знакомство с основными понятиями программирования такими как переменные, операторы присваивания, циклы, функции. Тем не менее и новичок в программировании должен оказаться в состоянии читать подряд и освоиться с языком, хотя при этом была бы полезной помощь более опытного коллеги.

По нашему опыту, **C** показал себя приятным, выразительным и разносторонним языком на широком множестве разнообразных программ. Его легко выучить, и он не теряет своих качеств с ростом опыта программиста. Мы надеемся, что эта книга поможет вам хорошо его использовать.

Вдумчивая критика и предложения многих наших друзей и коллег очень много добавили как для самой книги, так и для нашего удовольствия при её написании. В частности, Майк Биапси, Джим Блю, Стью Фельдман, Доуг Мак-Илрой, Билл Рум, Боб Розин и Ларри Рослер тщательно прочитали множество вариантов. Мы также обязаны Элю Ахо, Стиву Борну, Дэву Двораку, Чаку Хэлею, Дебби Хэлей, Мариону Харрису, Рику Холту, Стиву Джонсону, Джону Машею, Бобу Митцу, Ральфу Мьюа, Питеру Нельсону, Эллиоту Пинсону, Биллу Плагеру, Джерри Спиваку, Кену Томпсону и Питеру Вейнбергеру за

полезные замечания на различных этапах и Майку Лоску и Джо Осанна за неоценимую помощь при печатании книги.

Брайен В. Керниган
Деннис М. Ричи

Оглавление

Введение 1

1 Учебное введение 5
1.1 Начинаем . . . 5
1.2 Переменные и арифметика . . . 8
1.3 Оператор for . . . 12
1.4 Символические константы . . . 13
1.5 Набор полезных программ . . . 14
1.5.1 Ввод и вывод символов . . . 14
1.5.2 Копирование файла . . . 14
1.5.3 Подсчёт символов . . . 16
1.5.4 Подсчёт строк . . . 17
1.5.5 Подсчёт слов . . . 19
1.6 Массивы . . . 21
1.7 Функции . . . 23
1.8 Аргументы – вызов по значению . . . 25
1.9 Массивы символов . . . 26
1.10 Область действия: внешние переменные . . . 29
1.11 Резюме . . . 32

2 Типы, операции и выражения 33
2.1 Имена переменных . . . 33
2.2 Типы и размеры данных . . . 33
2.3 Константы . . . 34
2.3.1 Символьная константа . . . 35
2.3.2 Константное выражение . . . 35
2.3.3 Строчная константа . . . 35
2.4 Описания . . . 36
2.5 Арифметические операции . . . 37
2.6 Операции отношения и логические операции . . . 38
2.7 Преобразование типов . . . 39
2.8 Операции увеличения и уменьшения . . . 43
2.9 Побитовые логические операции . . . 45
2.10 Операции и выражения присваивания . . . 46
2.11 Условные выражения . . . 48
2.12 Старшинство и порядок вычисления . . . 49

3 Поток управления **53**
3.1 Операторы и блоки . . . 53
3.2 if – else . . . 53
3.3 else – if . . . 55
3.4 Переключатель . . . 56
3.5 Циклы – while и for . . . 58
3.6 Цикл do – while . . . 61
3.7 Оператор break . . . 62
3.8 Оператор continue . . . 63
3.9 Оператор goto и метки . . . 64

4 Функции и структура программ **67**
4.1 Основные сведения . . . 67
4.2 Функции, возвращающие нецелые значения . . . 70
4.3 Ещё об аргументах функций . . . 73
4.4 Внешние переменные . . . 73
4.5 Правила, определяющие область действия . . . 78
4.5.1 Область действия . . . 78
4.6 Статические переменные . . . 82
4.7 Регистровые переменные . . . 83
4.8 Блочная структура . . . 84
4.9 Инициализация . . . 85
4.10 Рекурсия . . . 87
4.11 Препроцессор языка **C** . . . 88
4.11.1 Включение файлов . . . 88
4.11.2 Макроподстановка . . . 89

5 Указатели и массивы **91**
5.1 Указатели и адреса . . . 91
5.2 Указатели и аргументы функций . . . 93
5.3 Указатели и массивы . . . 95
5.4 Адресная арифметика . . . 98
5.5 Указатели символов и функции . . . 101
5.6 Указатели – не целые . . . 104
5.7 Многомерные массивы . . . 105
5.8 Массивы указателей; указатели указателей . . . 107
5.9 Инициализация массивов указателей . . . 111
5.10 Указатели и многомерные массивы . . . 111
5.11 Командная строка аргументов . . . 112
5.12 Указатели на функции . . . 117

6 Структуры **121**
6.1 Основные сведения . . . 121
6.2 Структуры и функции . . . 123

6.3 Массивы структур . . . 125
6.4 Указатели на структуры . . . 130
6.5 Структуры, ссылающиеся на себя . . . 132
6.6 Поиск в таблице . . . 136
6.7 Поля . . . 138
6.8 Объединения . . . 140
6.9 Определение типа . . . 142

7 Ввод и вывод 145
7.1 Обращение к стандартной библиотеке . . . 145
7.2 Стандартный ввод и вывод – функции getchar и putchar . . . 146
7.3 Форматный вывод – функция printf . . . 147
7.4 Форматный ввод – функция scanf . . . 149
7.5 Форматное преобразование в памяти . . . 152
7.6 Доступ к файлам . . . 153
7.7 Обработка ошибок – stderr и exit . . . 155
7.8 Ввод и вывод строк . . . 157
7.9 Несколько разнообразных функций . . . 158
7.9.1 Проверка вида символов и преобразования . . . 158
7.9.2 Функция ungetc . . . 159
7.9.3 Обращение к системе . . . 159
7.9.4 Управление памятью . . . 159

8 Интерфейс системы UNIX 161
8.1 Дескрипторы файлов . . . 161
8.2 Низкоуровневый ввод-вывод – операторы read и write . . . 162
8.3 Открытие, создание, закрытие и расцепление (unlink) . . . 164
8.4 Произвольный доступ – seek и lseek . . . 166
8.5 Пример – реализация функций fopen и getc . . . 167
8.6 Пример – распечатка справочников . . . 171
8.7 Пример – распределитель памяти . . . 175

9 Приложение A: Справочное руководство по языку C 181
9.1 Введение . . . 181
9.2 Лексические соглашения . . . 181
9.2.1 Комментарии . . . 181
9.2.2 Идентификаторы (имена) . . . 181
9.2.3 Ключевые слова . . . 182
9.2.4 Константы . . . 182
9.2.5 Строки . . . 183
9.3 Характеристики аппаратных средств . . . 184
9.4 Синтаксическая нотация . . . 184
9.5 Что в имени тебе моем? . . . 184
9.6 Объекты и L-значения . . . 185

9.7 Преобразования . . . 186
9.7.1 Символы и целые . . . 186
9.7.2 Типы float и double . . . 186
9.7.3 Плавающие и целочисленные величины . . . 186
9.7.4 Указатели и целые . . . 187
9.7.5 Целое без знака . . . 187
9.7.6 Арифметические преобразования . . . 187
9.8 Выражения . . . 188
9.8.1 Первичные выражения . . . 188
9.8.2 Унарные операции . . . 190
9.8.3 Мультипликативные операции . . . 191
9.8.4 Аддитивные операции . . . 192
9.8.5 Операции сдвига . . . 193
9.8.6 Операции отношения . . . 193
9.8.7 Операции равенства . . . 193
9.8.8 Побитовая операция «И» . . . 194
9.8.9 Побитовая операция исключающего «ИЛИ» . . . 194
9.8.10 Побитовая операция включающего «ИЛИ» . . . 194
9.8.11 Логическая операция «И» . . . 194
9.8.12 Операция логического «ИЛИ» . . . 195
9.8.13 Условная операция . . . 195
9.8.14 Операция присваивания . . . 195
9.8.15 Операция запятая . . . 196
9.9 Описания . . . 197
9.9.1 Спецификаторы класса памяти . . . 197
9.9.2 Спецификаторы типа . . . 198
9.9.3 Описатели . . . 198
9.9.4 Смысл описателей . . . 199
9.9.5 Описание структур и объединений . . . 200
9.9.6 Инициализация . . . 203
9.9.7 Имена типов . . . 204
9.9.8 typedef . . . 205
9.10 Операторы . . . 206
9.10.1 Операторное выражение . . . 206
9.10.2 Составной оператор (или блок) . . . 206
9.10.3 Условные операторы . . . 207
9.10.4 Оператор while . . . 207
9.10.5 Оператор do . . . 207
9.10.6 Оператор for . . . 207
9.10.7 Оператор switch . . . 208
9.10.8 Оператор break . . . 209
9.10.9 Оператор continue . . . 209
9.10.10 Оператор возврата . . . 209
9.10.11 Оператор goto . . . 210

9.10.12 Помеченный оператор 210
9.10.13 Пустой оператор 210
9.11 Внешние определения 210
9.11.1 Внешнее определение функции 211
9.11.2 Внешние определения данных 212
9.12 Правила, определяющие область действия 212
9.12.1 Лексическая область действия 212
9.12.2 Область действия внешних идентификаторов 213
9.13 Строки управления компилятором 213
9.13.1 Замена лексем 214
9.13.2 Включение файлов 214
9.13.3 Условная компиляция 215
9.14 Неявные описания 215
9.15 Снова о типах 215
9.15.1 Структуры и объединения 216
9.15.2 Функции 216
9.15.3 Массивы, указатели и индексация 217
9.15.4 Явные преобразования указателей 217
9.16 Константные выражения 218
9.17 Соображения о переносимости 219
9.18 Анахронизмы 220
9.19 Сводка синтаксических правил 220
9.19.1 Выражения 221
9.19.2 Описания 222
9.19.3 Операторы 224
9.19.4 Внешние определения 225
9.19.5 Препроцессор 225

10 Последние изменения языка C (15 ноября 1978 г.) 227
10.1 Присваивание структуры 227
10.2 Тип перечисления 227

Введение

Язык **C** является универсальным языком программирования. Он тесно связан с операционной системой UNIX, так как был развит на этой системе и так как UNIX и её программное обеспечение написано на **C**. Сам язык, однако, не связан с какой-либо одной операционной системой или машиной; и хотя его называют языком системного программирования, так как он удобен для написания операционных систем, он с равным успехом использовался при написании больших вычислительных программ, программ для обработки текстов и баз данных.

Язык **C** – это язык относительно «низкого уровня». В такой характеристике нет ничего оскорбительного; это просто означает, что **C** имеет дело с объектами того же вида, что и большинство ЭВМ, а именно, с символами, числами и адресами. Они могут объединяться и пересылаться посредством обычных арифметических и логических операций, осуществляемых реальными ЭВМ.

В языке **C** отсутствуют операции, имеющие дело непосредственно с составными объектами, такими как строки символов, множества, списки или с массивами, рассматриваемыми как целое. Здесь, например, нет никакого аналога операциям PL/1, оперирующим с целыми массивами и строками. Язык не предоставляет никаких других возможностей распределения памяти, кроме статического определения и механизма стеков, обеспечиваемого локальными переменных функций; здесь нет ни «куч» (HEAP), ни «сборки мусора», как это предусматривается в АЛГОЛЕ-68. Наконец, сам по себе **C** не обеспечивает никаких возможностей ввода-вывода: здесь нет операторов READ или WRITE и никаких встроенных методов доступа к файлам. Все эти механизмы высокого уровня должны обеспечиваться явно вызываемыми функциями.

Аналогично, язык **C** предлагает только простые, последовательные конструкции потоков управления: проверки, циклы, группирование и подпрограммы, но не мультипрограммирование, параллельные операции, синхронизацию или сопрограммы.

Хотя отсутствие некоторых из этих средств может выглядеть как удручающая неполноценность («выходит, что я должен обращаться к функции, чтобы сравнить две строки символов?!»), но удержание языка в скромных размерах даёт реальные преимущества. Так как **C** относительно мал, он не требует много места для своего описания и может быть быстро выучен. Компилятор с **C** может быть простым и компактным. Кроме того, компиляторы легко пишутся; при использовании современной технологии можно ожидать написания компилятора для новой ЭВМ за пару месяцев и при этом окажется, что 80 процентов программы нового компилятора будет общей с программой для уже существующих компиляторов. Это обеспечивает высокую степень мобильности языка. Поскольку типы данных и структуры управления, имеющиеся в **C**, непосредственно поддерживаются большинством существующих ЭВМ, библиотека, необходимая во время прогона изолированных программ, оказывается очень маленькой. На PDP-11, например,

она содержит только программы для 32-битового умножения и деления и для выполнения программ ввода и вывода последовательностей. Конечно, каждая реализация обеспечивает исчерпывающую, совместимую библиотеку функций для выполнения операций ввода-вывода, обработки строк и распределения памяти, но так как обращение к ним осуществляется только явно, можно, если необходимо, избежать их вызова; эти функции могут быть компактно написаны на самом **C**.

Опять же из-за того, что язык **C** отражает возможности современных компьютеров, программы на **C** оказываются достаточно эффективными, так что не возникает побуждения писать вместо этого программы на языке ассемблера. Наиболее убедительным примером этого является сама операционная система UNIX, которая почти полностью написана на **C**. Из 13000 строк программы системы только около 800 строк самого низкого уровня написаны на ассемблере. Кроме того, по существу все прикладное программное обеспечение системы UNIX написано на **C**; подавляющее большинство пользователей системы UNIX(включая одного из авторов этой книги) даже не знает языка ассемблера PDP-11.

Хотя **C** соответствует возможностям многих ЭВМ, он не зависит от какой-либо конкретной архитектуры машины и в силу этого без особых усилий позволяет писать «переносимые» программы, т.е. программы, которые можно пропускать без изменений на различных аппаратных средствах. В наших кругах стал уже традицией перенос программного обеспечения, разработанного на системе UNIX, на системы ЭВМ: HONEYWELL, IBM и INTERDATA. Фактически компиляторы с **C** и программное обеспечение во время прогона программ на этих четырёх системах, по-видимому, гораздо более совместимы, чем стандартные версии фортрана американского национального института стандартов (ANSI). Сама операционная система UNIX теперь работает как на PDP-11, так и на INTERDATA 8/32. За исключением программ, которые неизбежно оказываются в некоторой степени машинно-зависимыми, таких как компилятор, ассемблер и отладчик. Написанное на языке **C** программное обеспечение идентично на обеих машинах. Внутри самой операционной системы 7000 строк программы, исключая математическое обеспечение языка ассемблера ЭВМ и управления операциями ввода-вывода, совпадают на 95 процентов.

Программистам, знакомым с другими языками, для сравнения и противопоставления может оказаться полезным упоминание нескольких исторических, технических и философских аспектов **C**.

Многие из наиболее важных идей **C** происходят от гораздо более старого, но все ещё вполне жизненного языка BCPL, разработанного Мартином Ричардсом. Косвенно язык BCPL оказал влияние на **C** через язык B, написанный Кеном Томпсоном в 1970 году для первой операционной системы UNIX на ЭВМ PDP-7.

Хотя язык **C** имеет несколько общих с BCPL характерных особенностей, он никоим образом не является диалектом последнего. И BCPL и B – «безтипные» языки; единственным видом данных для них являются машинное слово, а доступ к другим объектам реализуется специальными операторами или обращением к функциям. В языке **C** объектами основных типов данных являются символы, целые числа нескольких размеров и числа с плавающей точкой. Кроме того, имеется иерархия производных типов данных, создаваемых указателями, массивами, структурами, объединениями и функциями.

Язык **C** включает основные конструкции потока управления, требуемые для хорошо

структурированных программ: группирование операторов, принятие решений (`if`), циклы с проверкой завершения в начале (`while`, `for`) или в конце (`do`) и выбор одного из множества возможных вариантов (`switch`). Все эти возможности обеспечивались и в BCPL, хотя и при несколько отличном синтаксисе; этот язык предчувствовал наступившую через несколько лет моду на структурное программирование.

В языке **C** имеются указатели и возможность адресной арифметики. Аргументы передаются функциям посредством копирования значения аргумента, и вызванная функция не может изменить фактический аргумент в вызывающей программе. Если желательно добиться «вызова по ссылке», можно неявно передать указатель, и функция сможет изменить объект, на который этот указатель указывает. Имена массивов передаются указанием начала массивов, так что аргументы типа массивов эффективно вызываются по ссылке.

К любой функции можно обращаться рекурсивно, и её локальные переменные обычно «автоматические», т.е. создаются заново при каждом обращении. Описание одной функции не может содержаться внутри другой, но переменные могут описываться в соответствии с обычной блочной структурой. Функции в **C** программе могут транслироваться отдельно. Переменные по отношению к функции могут быть внутренними, внешними, но известными только в пределах одного исходного файла, или полностью глобальными. Внутренние переменные могут быть автоматическими или статическими. Автоматические переменные для большей эффективности можно помещать в регистры, но объявление регистра является только указанием для компилятора и никак не связано с конкретными машинными регистрами.

Язык **C** не является языком со строгими типами в смысле паскаля или алгола 68. Он сравнительно снисходителен к преобразованию данных, хотя и не будет автоматически преобразовывать типы данных с буйной непринуждённостью языка PL/1. Существующие компиляторы не предусматривают никакой проверки во время выполнения программы индексов массивов, типов аргументов и т.д.

В тех ситуациях, когда желательна строгая проверка типов, используется специальная версия компилятора. Эта программа называется `lint` очевидно потому, что она выбирает кусочки пуха из вашей программы[1]. Программа `lint` не генерирует машинного кода, а делает очень строгую проверку всех тех сторон программы, которые можно проконтролировать во время компиляции и загрузки. Она определяет несоответствие типов, несовместимость аргументов, неиспользованные или очевидным образом неинициализированные переменные, потенциальные трудности переносимости и т.д. Для программ, которые благополучно проходят через `lint`, гарантируется отсутствие ошибок типа примерно с той же полнотой, как и для программ, написанных, например, на АЛГОЛЕ-68. Другие возможности программы `lint` будут отмечены, когда представится соответствующий случай.

Наконец, язык **C**, подобно любому другому языку, имеет свои недостатки. Некоторые операции имеют неудачное старшинство; некоторые разделы синтаксиса могли бы быть лучше; существует несколько версий языка, отличающихся небольшими деталями. Тем не менее язык **C** зарекомендовал себя как исключительно эффективный и выразительный

[1] lint в переводе с английского – вата.

язык для широкого разнообразия применений программирования.

Содержание книги организовано следующим образом. Глава 1 является учебным введением в центральную часть языка **C**. Цель – позволить читателю стартовать так быстро,как только возможно, так как мы твёрдо убеждены, что единственный способ изучить новый язык – писать на нем программы. При этом, однако, предполагается рабочее владение основными элементами программирования; здесь не объясняется, что такое ЭВМ или компилятор, не поясняется смысл выражений типа `n=n+1`, хотя мы и пытались, где это возможно, продемонстрировать полезную технику программирования. Эта книга не предназначается быть справочным руководством по структурам данных и алгоритмам; там, где мы вынуждены были сделать выбор, мы концентрировались на языке.

В главах со 2-й по 6-ю различные аспекты **C** излагаются более детально и несколько более формально, чем в главе 1, хотя ударение по-прежнему делается на разборе примеров законченных, полезных программ, а не на отдельных фрагментах.

В главе 2 обсуждаются основные типы данных, операторы и выражения. В главе 3 рассматриваются управляющие операторы: `if-else`, `while`, `for` и т.д. Глава 4 охватывает функции и структуру программы – внешние переменные, правила определённых областей действия описания и т.д. В главе 5 обсуждаются указатели и адресная арифметика. Глава 6 содержит подробное описание структур и объединений.

В главе 7 описывается стандартная библиотека ввода-вывода языка **C**, которая обеспечивает стандартный интерфейс с операционной системой. Эта библиотека ввода-вывода поддерживается на всех машинах, на которых реализован **C**, так что программы, использующие её для ввода, вывода и других системных функций, могут переноситься с одной системы на другую по существу без изменений.

В главе 8 описывается интерфейс между **C**-программами и операционной системой UNIX. Упор делается на ввод-вывод, систему файлов и переносимость. Хотя некоторые части этой главы специфичны для операционной системы UNIX, программисты, не использующие UNIX, все же должны найти здесь полезный материал, в том числе некоторое представление о том, как реализована одна версия стандартной библиотеки и предложения для достижения переносимости программы.

Приложение A содержит справочное руководство по языку **C**. Оно является «официальным» изложением синтаксиса и семантики **C** и (исключая чей-либо собственный компилятор) окончательным арбитром для всех двусмысленностей и упущений в предыдущих главах.

Так как **C** является развивающимся языком, реализованным на множестве систем, часть материла настоящей книги может не соответствовать текущему состоянию разработки на какой-то конкретной системе. Мы старались избегать таких проблем и предостерегать о возможных трудностях. В сомнительных случаях, однако, мы обычно предпочитали описывать ситуацию для системы UNIX PDP-11, так как она является средой для большинства программирующих на языке **C**. В приложении A также описаны расхождения в реализациях языка **C** на основных системах.

1 Учебное введение

Давайте начнём с быстрого введения в язык **C**. Наша цель – продемонстрировать существенные элементы языка на реальных программах, не увязая при этом в деталях, формальных правилах и исключениях. В этой главе мы не пытаемся изложить язык полностью или хотя бы строго (разумеется, приводимые примеры будут корректными). Мы хотим как можно скорее довести вас до такого уровня, на котором вы были бы в состоянии писать полезные программы, и чтобы добиться этого, мы сосредотачиваемся на основном: переменных и константах, арифметике, операторах передачи управления, функциях и элементарных сведениях о вводе и выводе. Мы совершенно намеренно оставляем за пределами этой главы многие элементы языка **C**, которые имеют первостепенное значение при написании больших программ, в том числе указатели, структуры, большую часть из богатого набора операторов языка **C**, несколько операторов передачи управления и несметное количество деталей.

Такой подход имеет, конечно, свои недостатки. Самым существенным является то, что полное описание любого конкретного элемента языка не излагается в одном месте, а пояснения, в силу краткости, могут привести к неправильному истолкованию. Кроме того, из-за невозможности использовать всю мощь языка, примеры оказываются не столь краткими и элегантными, как они могли бы быть. И хотя мы старались свести эти недостатки к минимуму, все же имейте их ввиду.

Другой недостаток состоит в том, что последующие главы будут неизбежно повторять некоторые части этой главы. Мы надеемся, что такое повторение будет скорее помогать, чем раздражать.

Во всяком случае, опытные программисты должны оказаться в состоянии проэкстраполировать материал данной главы на свои собственные программистские нужды. Начинающие же должны в дополнение писать аналогичные маленькие самостоятельные программы. И те, и другие могут использовать эту главу как каркас, на который будут навешиваться более подробные описания, начинающиеся с главы 2.

1.1 Начинаем

Единственный способ освоить новый язык программирования – писать на нем программы. Первая программа, которая должна быть написана, – одна для всех языков: напечатать слова HELLO, WORLD.

Это – самый существенный барьер; чтобы преодолеть его, вы должны суметь завести где-то текст программы, успешно его скомпилировать, загрузить, прогнать и найти, где оказалась ваша выдача. Если вы научились справляться с этими техническими деталями, все остальное сравнительно просто.

Программа печати HELLO, WORLD на языке **C** имеет вид:

```
main()
{
  printf("HELLO, WORLD\n");
}
```

Как пропустить эту программу – зависит от используемой вами системы. В частности, на операционной системе UNIX вы должны завести исходную программу в файле, имя которого оканчивается на «.c», например, hello.c, и затем скомпилировать её по команде

```
cc hello.c
```

Если вы не допустили какой-либо небрежности, такой как пропуск символа или неправильное написание, компиляция пройдёт без сообщений и будет создан исполняемый файл с именем a.out. Прогон его по команде

```
./a.out
```

приведёт к выводу

```
HELLO, WORLD
```

На других системах эти правила будут иными; проконсультируйтесь с местным авторитетом.

Упражнение 1–1

Пропустите эту программу на вашей системе. Попробуйте не включать различные части программы и посмотрите какие сообщения об ошибках вы при этом получите.

Теперь некоторые пояснения к самой программе. Любая **C**-программа, каков бы ни был её размер, состоит из одной или более «функций», указывающих фактические операции компьютера, которые должны быть выполнены. Функции в языке **C** подобны функциям и подпрограммам фортрана и процедурам PL/1, паскаля и т.д. В нашем примере такой функцией является `main`. Обычно вы можете давать функциям любые имена по вашему усмотрению, но `main` – это особое имя; выполнение вашей программы начинается сначала с функции `main`. Это означает, что каждая программа должна в каком-то месте содержать функцию с именем `main`. Для выполнения определённых действий функция `main` обычно обращается к другим функциям, часть из которых находится в той же самой программе, а часть – в библиотеках, содержащих ранее написанные функции.

Одним способом обмена данными между функциями является передача посредством аргументов. Круглые скобки, следующие за именем функции, заключают в себе список аргументов; здесь `main` – функция без аргументов, что указывается как (). Операторы, составляющие функцию, заключаются в фигурные скобки { и }, которые аналогичны DO-END в PL/1 или BEGIN-END в алголе, паскале и т.д. Обращение к функции осуществляется указанием её имени, за которым следует заключённый в круглые скобки

список аргументов. Здесь нет никаких операторов CALL, как в фортране или PL/1. Круглые скобки должны присутствовать и в том случае, когда функция не имеет аргументов. Строка

```
printf("HELLO, WORLD\n");
```

является обращением к функции, которое вызывает функцию с именем `printf` и аргументом `''HELLO, WORLD\n''`. Функция `printf` является библиотечной функцией, которая выдаёт выходные данные на терминал (если только не указано какое-то другое место назначения). В данном случае печатается строка символов, являющаяся аргументом функции.

Последовательность из любого количества символов, заключённых в удвоенные кавычки "...", называется «символьной строкой» или «строчной константой». Пока мы будем использовать символьные строки только в качестве аргументов для `printf` и других функций.

Последовательность \n в приведённой строке является обозначением на языке **C** для «символа новой строки», который служит указанием для перехода на терминале к левому краю следующей строки. Если вы не включите \n (полезный эксперимент), то обнаружите, что ваша выдача не закончится переходом терминала на новую строку. Использование последовательности \n – единственный способ введения символа новой строки в аргумент функции `printf`; если вы попробуете что-нибудь вроде

```
printf("HELLO, WORLD
");
```

то **C**-компилятор будет печатать злорадные диагностические сообщения о недостающих кавычках.

Функция `printf` не обеспечивает автоматического перехода на новую строку, так что многократное обращение к ней можно использовать для поэтапной сборки выходной строки. Наша первая программа, печатающая идентичную выдачу, с точно таким же успехом могла бы быть написана в виде

```
main()
{
  printf("HELLO, ");
  printf("WORLD");
  printf("\n");
}
```

Подчеркнём, что \n представляет только один символ. Условные «последовательности», подобные \n, дают общий и допускающий расширение механизм для представления трудных для печати или невидимых символов. Среди прочих символов в языке **C** предусмотрены следующие:

\t – для табуляции;
\b – для возврата на одну позицию;
\" – для двойной кавычки;
\\ – для самой обратной косой черты.

Упражнение 1–2

Проведите эксперименты для того, чтобы узнать что произойдёт, если в строке, являющейся аргументом функции `printf` будет содержаться \X, где X – некоторый символ, не входящий в вышеприведённый список.

1.2 Переменные и арифметика

Следующая программа печатает приведённую ниже таблицу температур по Фаренгейту и их эквивалентов по стоградусной шкале Цельсия, используя для перевода формулу

```
      C = (5/9)*(F-32).
 0      -17.8
20       -6.7
40        4.4
60       15.6
...       ...
260     126.7
280     137.8
300     140.9
```

Теперь сама программа:

```
/* print fahrenheit-celsius table
   for f = 0, 20, ..., 300 */
main()
{
  int lower, upper, step;
  float fahr, celsius;
  lower = 0;    /* lower limit of temperature table */
  upper = 300;  /* upper limit */
  step = 20;    /* step size */
  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

```
/* print fahrenheit-celsius table
   for  f = 0, 20, ..., 300 */
```

являются комментарием, который в данном случае кратко поясняет, что делает программа. Любые символы между `/*` и `*/` игнорируются компилятором; можно свободно пользоваться комментариями для облегчения понимания программы. Комментарии могут появляться в любом месте, где возможен пробел или переход на новую строку.

В языке **C** все переменные должны быть описаны до их использования, обычно это делается в начале функции до первого выполняемого оператора. Если вы забудете вставить описание, то получите диагностическое сообщение от компилятора. Описание состоит из типа и списка переменных, имеющих этот тип, как в

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

Тип `int` означает, что все переменные списка целые; тип `float` предназначен для чисел с плавающей точкой, т.е. для чисел, которые могут иметь дробную часть. Точность как `int`, так и `float` зависит от конкретной машины, на которой вы работаете. На PDP-11, например, тип `int` соответствует 16-битовому числу со знаком, т.е. числу, лежащему между −32768 и +32767. Число типа `float` – это 32-битовое число, имеющее около семи значащих цифр и лежащее в диапазоне от 10^{-38} до 10^{+38}. В главе 2 приводится список размеров для других машин.

В языке **C** предусмотрено несколько других основных типов данных, кроме `int` и `float`, это:

`char`	–	символ – один байт;
`short`	–	короткое целое;
`long`	–	длинное целое;
`double`	–	плавающее с двойной точностью.

Размеры этих объектов тоже машинно-независимы; детали приведены в главе 2. Имеются также массивы, структуры и объединения этих основных типов, указатели на них и функции,которые их возвращают; со всеми ними мы встретимся в своё время.

Фактически вычисления в программе перевода температур начинаются с операторов присваивания

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

которые придают переменным их начальные значения. Каждый отдельный оператор заканчивается точкой с запятой.

Каждая строка таблицы вычисляется одинаковым образом, так что мы используем цикл, повторяющийся один раз на строку. В этом назначение оператора `while`:

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

проверяется условие в круглых скобках. Если оно истинно (fahr меньше или равно upper), то выполняется тело цикла (все операторы, заключённые в фигурные скобки { и }). Затем вновь проверяется это условие и, если оно истинно, опять выполняется тело цикла. Если же условие не выполняется (fahr превосходит upper), цикл заканчивается и происходит переход к выполнению оператора, следующего за оператором цикла. Так как в настоящей программе нет никаких последующих операторов, то выполнение программы завершается.

Тело оператора `while` может состоять из одного или более операторов, заключённых в фигурные скобки, как в программе перевода температур, или из одного оператора без скобок, как, например, в

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

В обоих случаях операторы, управляемые оператором `while`, сдвинуты на одну табуляцию, чтобы вы могли с первого взгляда видеть, какие операторы находятся внутри цикла. Такой сдвиг подчёркивает логическую структуру программы. Хотя в языке **C** допускается совершенно произвольное расположение операторов в строке, подходящий сдвиг и использование пробелов значительно облегчают чтение программ. Мы рекомендуем писать только один оператор на строке и (обычно) оставлять пробелы вокруг операторов. Расположение фигурных скобок менее существенно; мы выбрали один из нескольких популярных стилей. Выберите подходящий для вас стиль и затем используйте его последовательно.

Основная часть работы выполняется в теле цикла. Температура по Цельсию вычисляется и присваивается переменной celsius оператором

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

причина использования выражения 5.0/9.0 вместо выглядящего проще 5/9 заключается в том, что в языке **C**, как и во многих других языках, при делении целых происходит усечение, состоящее в отбрасывании дробной части результата. Таким образом, результат операции 5/9 равен нулю, и, конечно, в этом случае все температуры оказались бы равными нулю. Десятичная точка в константе указывает, что она имеет тип с плавающей точкой, так что, как мы и хотели, 5.0/9.0 равно 0.5555...

Мы также писали 32.0 вместо 32, несмотря на то, что так как переменная fahr имеет тип `float`, целое 32 автоматически бы преобразовалось к типу `float` (в 32.0) перед вычитанием. С точки зрения стиля разумно писать плавающие константы с явной десятичной точкой даже тогда, когда они имеют целые значения; это подчёркивает их плавающую природу для просматривающего программу и обеспечивает то, что компилятор будет смотреть на вещи так же, как и Вы.

Подробные правила о том, в каком случае целые преобразуются к типу с плавающей точкой, приведены в главе 2. Сейчас же отметим, что присваивание

```
fahr = lower;
```

проверка

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

работают, как ожидается, – перед выполнением операций целые преобразуются в плавающую форму.

Этот же пример сообщает чуть больше о том, как работает `printf`. Функция `printf` фактически является универсальной функцией форматных преобразований, которая будет полностью описана в главе 7. Её первым аргументом является строка символов, которая должна быть напечатана, причём каждый знак % указывает, куда должен подставляться каждый из остальных аргументов (второй, третий, ...) и в какой форме он должен печататься. Например, в операторе

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

спецификация преобразования %4.0f говорит, что число с плавающей точкой должно быть напечатано в поле шириной по крайней мере в четыре символа без цифр после десятичной точки. спецификация %6.1f описывает другое число, которое должно занимать по крайней мере шесть позиций с одной цифрой после десятичной точки, аналогично спецификациям F6.1 в фортране или F(6,1) в PL/1. Различные части спецификации могут быть опущены: спецификация %6f говорит, что число будет шириной по крайней мере в шесть символов; спецификация %2 требует двух позиций после десятичной точки, но ширина при этом не ограничивается; спецификация %f говорит только о том, что нужно напечатать число с плавающей точкой. Функция `printf` также распознает следующие спецификации:

%d	–	для десятичного целого;
%o	–	для восьмеричного числа;
%x	–	для шестнадцатеричного;
%c	–	для символа;
%s	–	для символьной строки;
%%	–	для самого символа %.

Каждая конструкция с символом % в первом аргументе функции `printf` сочетается с соответствующим вторым, третьим, и т.д. аргументами; они должны согласовываться по числу и типу; в противном случае вы получите бессмысленные результаты.

Между прочим, функция `printf` не является частью языка **C**; в самом языке **C** не определены операции ввода-вывода. Нет ничего таинственного и в функции `printf`; это – просто полезная функция, являющаяся частью стандартной библиотеки подпрограмм, которая обычно доступна **C**-программам. Чтобы сосредоточиться на самом языке, мы не будем подробно останавливаться на операциях ввода-вывода до главы 7. В частности, мы до тех пор отложим форматный ввод. Если вам надо ввести числа – прочитайте описание функции `scanf` в главе 7, раздел 7.4. Функция `scanf` во многом сходна с `printf`, но она считывает входные данные, а не печатает выходные.

Упражнение 1–3

Преобразуйте программу перевода температур таким образом, чтобы она печатала заголовок к таблице.

Упражнение 1–4

Напишите программы печати соответствующей таблицы перехода от градусов Цельсия к градусам Фаренгейта.

1.3 Оператор for

Как и можно было ожидать, имеется множество различных способов написания каждой программы. Давайте рассмотрим такой вариант программы перевода температур:

```
main()
{       /* fahrenheit-celsius table */
  int fahr;
  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
    printf("%4d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32.0));
}
```

Эта программа выдаёт те же самые результаты, но выглядит безусловно по-другому. Главное изменение – исключение большинства переменных; осталась только переменная fahr, причём типа `int` (это сделано для того, чтобы продемонстрировать преобразование %d в функции `printf`). Нижняя и верхняя границы и размер шага появляются только как константы в операторе `for`, который сам является новой конструкцией, а выражение, вычисляющее температуру по Цельсию, входит теперь в виде третьего аргумента функции `printf`, а не в виде отдельного оператора присваивания.

Последнее изменение является примером вполне общего правила языка **C** – в любом контексте, в котором допускается использование значения переменной некоторого типа, вы можете использовать выражение этого типа. Так как третий аргумент функции `printf` должен иметь значение с плавающей точкой, чтобы соответствовать спецификации %6.1f, то в этом месте может встретиться любое выражение плавающего типа.

Сам оператор `for` – это оператор цикла, обобщающий оператор `while`. Его функционирование должно стать ясным, если вы сравните его с ранее описанным оператором `while`. Оператор `for` содержит три части, разделяемые точкой с запятой. Первая часть

```
fahr = 0
```

выполняется один раз перед входом в сам цикл. Вторая часть проверка, или условие, которое управляет циклом:

```
fahr <= 300
```

это условие проверяется и, если оно истинно, то выполняется тело цикла (в данном случае только функция `printf`). Затем выполняется шаг реинициализации

```
fahr = fahr + 20
```

и условие проверяется снова. Цикл завершается, когда это условие становится ложным. Так же, как и в случае оператора `while`, тело цикла может состоять из одного оператора или из группы операторов, заключённых в фигурные скобки. Инициализирующая и реинициализирующая части могут быть любыми отдельными выражениями.

Выбор между операторами `while` и `for` произволен и основывается на том, что выглядит яснее. Оператор `for` обычно удобен для циклов, в которых инициализация и реинициализация логически связаны и каждая задаётся одним оператором, так как в этом случае запись более компактна, чем при использовании оператора `while`, а операторы управления циклом сосредотачиваются вместе в одном месте.

Упражнение 1–5

Модифицируйте программу перевода температур таким образом, чтобы она печатала таблицу в обратном порядке, т.е. от 300 градусов до 0.

1.4 Символические константы

Последнее замечание, прежде чем мы навсегда оставим программу перевода температур. Прятать «магические числа», такие как 300 и 20, внутрь программы – это неудачная практика; они дают мало информации тем, кто, возможно, должен будет разбираться в этой программе позднее, и их трудно изменять систематическим образом. К счастью в языке **C** предусмотрен способ, позволяющий избежать таких «магических чисел». Используя конструкцию `#define`, вы можете в начале программы определить символическое имя или символическую константу, которая будет конкретной строкой символов. Впоследствии компилятор заменит все не заключённые в кавычки появления этого имени на соответствующую строку. Фактически это имя может быть заменено абсолютно произвольным текстом, не обязательно цифрами

```
#define  LOWER 0        /* lower limit of table */
#define  UPPER 300      /* upper limit */
#define  STEP  20       /* step size */

main()
{       /* fahrenheit-celsius table */
  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 являются константами и поэтому они не указываются в описаниях. Символические имена обычно пишут прописными буквами, чтобы их было легко отличить от написанных строчными буквами имён переменных. Отметим, что в конце определения не ставится точка с запятой. Так как подставляется вся строка, следующая за определённым именем, то это привело бы к слишком большому числу точек с запятой в операторе `for`.

1.5 Набор полезных программ

Теперь мы собираемся рассмотреть семейство родственных программ, предназначенных для выполнения простых операций над символьными данными. В дальнейшем вы обнаружите, что многие программы являются просто расширенными версиями тех прототипов, которые мы здесь обсуждаем.

1.5.1 Ввод и вывод символов

Стандартная библиотека включает функции для чтения и записи по одному символу за один раз. Функция `getchar()` извлекает следующий вводимый символ каждый раз, как к ней обращаются, и возвращает этот символ в качестве своего значения. Это значит, что после

```
c = getchar()
```

переменная с содержит следующий символ из входных данных. Символы обычно поступают с терминала, но это не должно нас касаться до главы 7.

Функция `putchar(c)` является дополнением к `getchar`: в результате обращения

```
putchar(c)
```

содержимое переменной с выдаётся на некоторый выходной носитель, обычно опять на терминал. Обращение к функциям `putchar` и `printf` могут перемежаться; выдача будет появляться в том порядке, в котором происходят обращения.

Как и функция `printf`, функции `getchar` и `putchar` не содержат ничего экстраординарного. Они не входят в состав языка **C**, но к ним всегда можно обратиться.

1.5.2 Копирование файла

Имея в своём распоряжении только функции `getchar` и `putchar` вы можете, не зная ничего более об операциях ввода-вывода, написать удивительное количество полезных программ. Простейшим примером может служить программа посимвольного копирования вводного файла в выводной. Общая схема имеет вид:

```
ввести символ
while (символ не является признаком конца файла) {
  вывести только что прочитанный символ;
  ввести новый символ
}
```

Программа, написанная на языке **C**, выглядит следующим образом:

```
main()
{       /* copy input to output; 1st version */
  int c;
```

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

Оператор отношения `!=` означает «не равно».

Основная проблема заключается в том, чтобы зафиксировать конец файла ввода. Обычно, когда функция `gtchar` наталкивается на конец файла ввода, она возвращает значение, не являющееся действительным символом; таким образом, программа может установить, что файл ввода исчерпан. Единственное осложнение, являющееся значительным неудобством, заключается в существовании двух общеупотребительных соглашений о том, какое значение фактически является признаком конца файла. Мы отсрочим решение этого вопроса, использовав символическое имя `EOF` для этого значения, каким бы оно ни было. На практике `EOF` будет либо -1, либо 0, так что для правильной работы перед программой должно стоять собственно либо

```
#define  EOF  -1
```

либо

```
#define  EOF   0
```

Использовав символическую константу `EOF` для представления значения, возвращаемого функцией `getchar` при выходе на конец файла, мы обеспечили, что только одна величина в программе зависит от конкретного численного значения.

Мы также описали переменную с как `int`, а не `char`, с тем чтобы она могла хранить значение, возвращаемое `getchar`. как мы увидим в главе 2, эта величина действительно `int`, так как она должна быть в состоянии в дополнение ко всем возможным символам представлять и `EOF`.

Программистом, имеющим опыт работы на **C**, программа копирования была бы написана более сжато. В языке **C** любое присваивание, такое как

```
c = getchar()
```

может быть использовано в выражении; его значение – просто значение, присваиваемое левой части. Если присваивание символа переменной с поместить внутрь проверочной части оператора `while`, то программа копирования файла запишется в виде:

```
main()
{       /* copy input to output; 2nd version */
  int c;
  while ((c = getchar()) != EOF)
    putchar(c);
}
```

Программа извлекает символ, присваивает его переменной c и затем проверяет, не является ли этот символ признаком конца файла. Если нет – выполняется тело оператора `while`, выводящее этот символ. Затем цикл `while` повторяется. Когда, наконец, будет достигнут конец файла ввода, оператор `while` завершается, а вместе с ним заканчивается выполнение и функции `main`.

В этой версии централизуется ввод – в программе только одно обращение к функции `getchar` – и ужимается программа. Вложение присваивания в проверяемое условие – это одно из тех мест языка **C**, которое приводит к значительному сокращению программ. Однако, на этом пути можно увлечься и начать писать недоступные для понимания программы. Эту тенденцию мы будем пытаться сдерживать.

Важно понять, что круглые скобки вокруг присваивания в условном выражении действительно необходимы. Старшинство операции != выше, чем операции присваивания =, а это означает, что в отсутствие круглых скобок проверка условия != будет выполнена до присваивания =. Таким образом, оператор

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

эквивалентен оператору

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

Это, вопреки нашему желанию, приведёт к тому, что c будет принимать значение 0 или 1 в зависимости от того, натолкнётся или нет `getchar` на признак конца файла.[1]

1.5.3 Подсчёт символов

Следующая программа подсчитывает число символов; она представляет собой небольшое развитие программы копирования.

```
main()
{       /* count characters in input */
  long nc;

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

Оператор

```
++nc;
```

демонстрирует новую операцию, ++, которая означает увеличение на единицу. Вы могли бы написать nc = nc + 1, но ++nc более кратко и зачастую более эффективно. Имеется соответствующая операция −− уменьшение на единицу. Операции ++ и −− могут быть либо

[1] Подробнее об этом будет сказано в главе 2.

префиксными (++nc), либо постфиксными (nc++); эти две формы, как будет показано в главе 2, имеют в выражениях различные значения, но как ++nc, так и nc++ увеличивают nc. Пока мы будем придерживаться префиксных операций.

Программа подсчёта символов накапливает их количество в переменной типа `long`, а не `int`. На PDP-11 максимальное значение равно 32767, и если описать счётчик как `int`, то он будет переполняться даже при сравнительно малом файле ввода; на языке **C** для HONEYWELL и IBM типы `long` и `int` являются синонимами и имеют значительно больший размер. Спецификация преобразования %ld указывает `printf`, что соответствующий аргумент является целым типа `long`.

Чтобы справиться с ещё большими числами, вы можете использовать тип `double`[1] мы также используем оператор `for` вместо `while` с тем, чтобы проиллюстрировать другой способ записи цикла.

```
main()
{       /* count characters in input */
  double nc;

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

Функция `printf` использует спецификацию %f как для `float`, так и для `double`; спецификация %.0f подавляет печать несуществующей дробной части.

Тело оператора цикла `for` здесь пусто, так как вся работа выполняется в проверочной и реинициализационной частях. Но грамматические правила языка **C** требуют, чтобы оператор `for` имел тело. Изолированная точка с запятой, соответствующая пустому оператору, появляется здесь, чтобы удовлетворить этому требованию. Мы выделили её на отдельную строку, чтобы сделать её более заметной.

Прежде чем мы распростимся с программой подсчёта символов, отметим, что если файл ввода не содержит никаких символов, то условие в `while` или `for` не выполнится при самом первом обращении к `getchar`, и, следовательно, программа выдаст нуль, т.е. правильный ответ. Это важное замечание. Одним из приятных свойств операторов `while` и `for` является то, что они проверяют условие в начале цикла, т.е. до выполнения тела. Если делать ничего не надо, то ничего не будет сделано, даже если это означает, что тело цикла никогда не будет выполняться. программы должны действовать разумно, когда они обращаются с файлами типа «никаких символов». Операторы `while` и `for` помогают обеспечить правильное поведение программ при граничных значениях проверяемых условий.

1.5.4 Подсчёт строк

Следующая программа подсчитывает количество строк в файле ввода. Предполагается, что строки ввода заканчиваются символом новой строки \n, скрупулёзно добавлен-

[1] `float` двойной длины.

ным к каждой выписанной строке.

```
main()
{       /* count lines in input */
  int c, nl;

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

Тело `while` теперь содержит оператор `if`, который в свою очередь управляет оператором увеличения ++nl. Оператор `if` проверяет заключённое в круглые скобки условие и, если оно истинно, выполняет следующий за ним оператор (или группу операторов, заключённых в фигурные скобки). Мы опять использовали сдвиг вправо, чтобы показать, что чем управляет.

Удвоенный знак равенства == является обозначением в языке **C** для «равно» (аналогично `.EQ.` в фортране). Этот символ введён для того, чтобы отличать проверку на равенство от одиночного =, используемого при присваивании. Поскольку в типичных **C**-программах знак присваивания встречается примерно в два раза чаще, чем проверка на равенство, то естественно, чтобы знак оператора был вполовину короче.

Любой отдельный символ может быть записан внутри одиночных кавычек, и при этом ему соответствует значение, равное численному значению этого символа в машинном наборе символов; это называется символьной константой. Так, например, 'A' – символьная константа; её значение в наборе символов ASCII (американский стандартный код для обмена информацией) равно 65, внутреннему представлению символа A. Конечно, 'A' предпочтительнее, чем 65: его смысл очевиден и он не зависит от конкретного машинного набора символов.

Условные последовательности, используемые в символьных строках, также занимают законное место среди символьных констант. Так в проверках и арифметических выражениях '\n' представляет значение символа новой строки. Вы должны твёрдо уяснить, что '\n' – отдельный символ, который в выражениях эквивалентен одиночному целому; с другой стороны "\n" – это символьная строка, которая содержит только один символ. Вопрос о сопоставлении строк и символов обсуждается в главе 2.

Упражнение 1–6

Напишите программу для подсчёта пробелов, табуляций и новых строк.

Упражнение 1–7

Напишите программу, которая копирует ввод на вывод, заменяя при этом каждую последовательность из одного или более пробелов на один пробел.

1.5.5 Подсчёт слов

Четвёртая программа из нашей серии полезных программ подсчитывает количество строк, слов и символов, используя при этом весьма широкое определение, что словом является любая последовательность символов, не содержащая пробелов, табуляций или новых строк.[1]

```
#define  YES  1
#define  NO   0

main()
{       /* count lines, words, chars in input */
  int c, nl, nw, 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);
}
```

Каждый раз, когда программа встречает первый символ слова, она увеличивает счётчик числа слов на единицу. Переменная inword следит за тем, находится ли программа в настоящий момент внутри слова или нет; сначала этой переменной присваивается «не в слове», чему соответствует значение NO. Мы предпочитаем символические константы YES и NO литерным значениям 1 и 0, потому что они делают программу более удобной для чтения. Конечно, в такой крошечной программе, как эта, это не приводит к заметной разнице, но в больших программах увеличение ясности вполне стоит тех скромных дополнительных усилий, которых требует следование этому принципу с самого начала. Вы также обнаружите, что существенные изменения гораздо легче вносить в те программы, где числа фигурируют только в качестве символьных констант.

Строка

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

полагает все три переменные равными нулю. Это не особый случай, а следствие того обстоятельства, что оператору присваивания соответствует некоторое значение и при-

[1] Это – упрощённая версия утилиты wc системы UNIX.

сваивания проводятся последовательно справа налево. Таким образом, дело обстоит так, как если бы мы написали

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

операция || означает OR, так что строка

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

говорит «если c – пробел, или c – символ новой строки, или c – табуляция»[1]

Имеется соответствующая операция && для AND. Выражения, связанные операциями && или ||, рассматриваются слева на право, и при этом гарантируется, что оценивание выражений будет прекращено, как только станет ясно, является ли все выражение истинным или ложным. Так, если c оказывается пробелом, то нет никакой необходимости проверять, является ли c символом новой строки или табуляции, и такие проверки действительно не делаются. В данном случае это не имеет принципиального значения, но, как мы скоро увидим, в более сложных ситуациях эта особенность языка весьма существенна.

Этот пример также демонстрирует оператор `else` языка **C**, который указывает то действие, которое должно выполняться, если условие, содержащееся в операторе `if`, окажется ложным.

Общая форма такова:

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

Выполняется один и только один из двух операторов, связанных с конструкцией `if-else`. Если выражение истинно, выполняется оператор-1; если нет – выполняется оператор-2. Фактически каждый оператор может быть довольно сложным. В программе подсчёта слов оператор, следующий за `else`, является оператором `if`, который управляет двумя операторами в фигурных скобках.

Упражнение 1–8

Как бы вы стали проверять программу подсчёта слов? Какие имеются ограничения?

Упражнение 1–9

Напишите программу, которая будет печатать слова из файла ввода, причём по одному на строку.

Упражнение 1–10

Переделайте программу подсчёта слов, используя лучшее определение «слова»; считайте, например словом последовательность букв, цифр и апострофов, начинающуюся с буквы.

[1] Условная последовательность '\t' является изображением символа табуляции.

1.6 Массивы

Давайте напишем программу подсчёта числа появлений каждой цифры, символов пустых промежутков (пробел, табуляции, новая строка) и всех остальных символов. Конечно, такая задача несколько искусственна, но она позволит нам проиллюстрировать в одной программе сразу несколько аспектов языка **C**.

Мы разбили вводимые символы на двенадцать категорий, и нам удобнее использовать массив для хранения числа появлений каждой цифры, а не десять отдельных переменных. Вот один из вариантов программы:

```
main()
{       /* count digits, white space, others */
  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];
```

объявляет, что ndigit является массивом из десяти целых. В языке **C** индексы массива всегда начинаются с нуля (а не с 1, как в фортране или PL/1), так что элементами массива являются ndigit[0], ndigit[1],..., ndigit[9]. Эта особенность отражена в циклах `for`, которые инициализируют и печатают массив.

Индекс может быть любым целым выражением, которое, конечно, может включать целые переменные, такие как i, и целые константы.

Эта конкретная программа сильно опирается на свойства символьного представления цифр. Так, например, в программе проверка

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

определяет, является ли символ в с цифрой, и если это так, то численное значение этой цифры определяется по формуле `(c - '0')`. Такой способ работает только в том случае, если значения символьных констант '0', '1' и т.д. положительны, расположены в порядке возрастания и нет ничего, кроме цифр, между константами '0' и '9'. К счастью, это верно для всех общепринятых наборов символов.

По определению перед проведением арифметических операций, вовлекающих переменные типа `char` и `int`, все они преобразуются к типу `int`, так что в арифметических выражениях переменные типа `char` по существу идентичны переменным типа `int`. Это вполне естественно и удобно; например, `c - '0'` есть целое выражение со значением между 0 и 9 в соответствии с тем, какой символ от '0' до '9' хранится в с, и, следовательно, оно является подходящим индексом для массива ndigit.

Выяснение вопроса, является ли данный символ цифрой, символом пустого промежутка или чем-либо ещё, осуществляется последовательностью операторов

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

Конструкция

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

часто встречаются в программах как средство выражения ситуаций, в которых осуществляется выбор одного из нескольких возможных решений.

Программа просто движется сверху вниз до тех пор, пока не удовлетворится какое-нибудь условие; тогда выполняется соответствующий оператор, и вся конструкция завершается. (Конечно, «оператор» может состоять из нескольких операторов, заключённых в фигурные скобки). Если ни одно из условий не удовлетворяется, то выполняется оператор, стоящий после заключительного `else`, если оно присутствует. Если последнее `else` и соответствующий оператор опущены (как в программе подсчёта слов), то никаких действий не производится. Между начальным `if` и конечным `else` может помещаться произвольное количество групп

```
else if (условие)
  оператор
```

С точки зрения стиля целесообразно записывать эту конструкцию так, как мы показали, с тем чтобы длинные выражения не залезали за правый край страницы.

Оператор `switch` (переключатель), который рассматривается в главе 3, представляет другую возможность для записи разветвления на несколько вариантов. Этот оператор особенно удобен, когда проверяемое выражение является либо просто некоторым целым, либо символьным выражением, совпадающим с одной из некоторого набора констант. Версия этой программы, использующая оператор `switch`, будет для сравнения приведена в главе 3.

Упражнение 1–11

Напишите программу, печатающую гистограмму длин слов из файла ввода. Самое лёгкое – начертить гистограмму горизонтально; вертикальная ориентация требует больших усилий.

1.7 Функции

В языке **C** функции эквивалентны подпрограммам или функциям в фортране или процедурам в PL/1, паскале и т.д. Функции дают удобный способ заключения некоторой части вычислений в чёрный ящик, который в дальнейшем можно использовать, не интересуясь его внутренним содержанием. Использование функций является фактически единственным способом справиться с потенциальной сложностью больших программ. Если функции организованы должным образом, то можно игнорировать то, как делается работа; достаточно знание того, что делается. Язык **C** разработан таким образом, чтобы сделать использование функций лёгким, удобным и эффективным. Вам будут часто встречаться функции длиной всего в несколько строчек, вызываемые только один раз, и они используются только потому, что это проясняет некоторую часть программы.

До сих пор мы использовали только предоставленные нам функции типа `printf`, `getchar` и `putchar`; теперь пора написать несколько наших собственных. Так как в **C** нет операции возведения в степень, подобной операции `**` в фортране или PL/1, давайте проиллюстрируем механику определения функции на примере функции `power`(m,n), возводящей целое m в целую положительную степень n. Так значение `power`(2,5) равно 32. Конечно, эта функция не выполняет всей работы операции `**`, поскольку она действует только с положительными степенями небольших чисел, но лучше не создавать дополнительных затруднений, смешивая несколько различных вопросов.

Ниже приводится функция `power` и использующая её основная программа, так что вы можете видеть целиком всю структуру.

```
main()
{       /* test power function */
  int i;

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

```
}

power(x, n)     /* raise x n-th power; n > 0 */
int x, n;
{
  int i, p;
  p = 1;
  for (i = 1; i <= n; ++i)
    p = p * x;
  return (p);
}
```

Все функции имеют одинаковый вид:

```
имя (список аргументов, если они имеются)
описание аргументов, если они имеются
{
  описания
  операторы
}
```

Эти функции могут быть записаны в любом порядке и находиться в одном или двух исходных файлах. Конечно, если исходная программа размещается в двух файлах, вам придётся дать больше указаний при компиляции и загрузке, чем если бы она находилась в одном, но это дело операционной системы, а не атрибут языка. В данный момент, для того чтобы все полученные сведения о прогоне **C**-программ, не изменились в дальнейшем, мы будем предполагать, что обе функции находятся в одном и том же файле.

Функция `power` вызывается дважды в строке

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

при каждом обращении функция `power`, получив два аргумента, развращает целое значение, которое печатается в заданном формате. В выражениях `power(2,i)` является точно таким же целым, как 2 и `i`.[1].

Аргументы функции `power` должны быть описаны соответствующим образом, так как их типы известны. Это сделано в строке

```
int x, n;
```

которая следует за именем функции.

Описания аргументов помещаются между списком аргументов и открывающейся левой фигурной скобкой; каждое описание заканчивается точкой с запятой. Имена, использованные для аргументов функции `power`, являются чисто локальными и недоступны никаким другим функциям: другие процедуры могут использовать те же самые имена без возникновения конфликта. Это верно и для переменных `i` и `p`; `i` в функции `power` никак не связано с `i` в функции `main`.

[1] Не все функции выдают целое значение; мы займёмся этим вопросом в главе 4

Значение, вычисленное функцией `power`, передаются в `main` с помощью оператора `return`, точно такого же, как в PL/1. Внутри круглых скобок можно написать любое выражение. Функция не обязана возвращать какое-либо значение; оператор `return`, не содержащий никакого выражения, приводит к такой же передаче управления, как «сваливание на конец» функции при достижении конечной правой фигурной скобки, но при этом в вызывающую функцию не возвращается никакого полезного значения.

Упражнение 1–12

Напишите программу преобразования прописных букв из файла ввода в строчные, используя при этом функцию `lower(c)`, которая возвращает значение c, если c – не буква, и значение соответствующей строчной буквы, если c – буква.

1.8 Аргументы – вызов по значению

Один аспект в **C** может оказаться непривычным для программистов, которые использовали другие языки, в частности, фортран и PL/1. В языке **C** все аргументы функций передаются «по значению». Это означает, что вызванная функция получает значения своих аргументов с помощью временных переменных (фактически через стек), а не их адреса. Это приводит к некоторым особенностям, отличным от тех, с которыми мы сталкивались в языках типа фортрана и PL/1, использующих «вызов по ссылке», где вызванная процедура работает с адресом аргумента, а не с его значением.

Главное отличие состоит в том, что в **C** вызванная функция не может изменить переменную из вызывающей функции; она может менять только свою собственную временную копию.

Вызов по значению, однако, не помеха, а весьма ценное качество. Оно обычно приводит к более компактным программам, содержащим меньше не относящихся к делу переменных, потому что с аргументами можно обращаться как с удобно инициализированными локальными переменными вызванной процедуры. Вот, например, вариант функции `power` использующей это обстоятельство

```
power(x, n)     /* raise x n-th power; n > 0; version 2 */
int x, n;
{
  int p;

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

Аргумент n используется как временная переменная; из него вычитается единица до тех пор, пока он не станет нулём. Переменная i здесь больше не нужна. Чтобы ни происходило с n внутри `power` это никак не влияет на аргумент, с которым первоначально обратились к функции `power`.

При необходимости все же можно добиться, чтобы функция изменила переменную из вызывающей программы. Эта программа должна обеспечить установление адреса переменной (технически, через указатель на переменную), а в вызываемой функции надо описать соответствующий аргумент как указатель и ссылаться к фактической переменной косвенно через него. Мы рассмотрим это подробно в главе 5.

Когда в качестве аргумента выступает имя массива, то фактическим значением, передаваемым функции, является адрес начала массива. (Здесь нет никакого копирования элементов массива). С помощью индексации и адреса начала функция может найти и изменить любой элемент массива. Это – тема следующего раздела.

1.9 Массивы символов

По-видимому самым общим типом массива в **C** является массив символов. Чтобы проиллюстрировать использование массивов символов и обрабатывающих их функций, давайте напишем программу, которая читает набор строк и печатает самую длинную из них. Основная схема программы достаточно проста:

```
while (имеется ещё строка)
  if (эта строка длиннее самой длинной из предыдущих)
    запомнить эту строку и её длину
напечатать самую длинную строку
```

По этой схеме ясно, что программа естественным образом распадается на несколько частей. Одна часть читает новую строку, другая проверяет её, третья запоминает, а остальные части программы управляют этим процессом.

Поскольку все так прекрасно делится, было бы хорошо и написать программу соответствующим образом. Давайте сначала напишем отдельную функцию `getline`, которая будет извлекать следующую строку из файла ввода; это – обобщение функции `getchar`. Мы попытаемся сделать эту функцию по возможности более гибкой, чтобы она была полезной и в других ситуациях. Как минимум `getline` должна передавать сигнал о возможном появлении конца файла; более общий полезный вариант мог бы передавать длину строки или нуль, если встретится конец файла. Нуль не может быть длиной строки, так как каждая строка содержит по крайней мере один символ; даже строка, содержащая только символ новой строки, имеет длину 1.

Когда мы находим строку, которая длиннее самой длинной из предыдущих, то её надо где-то запомнить. Это наводит на мысль о другой функции, `copy`, которая будет копировать новую строку в место хранения.

Наконец, нам нужна основная программа для управления функциями `getline` и `copy`. Вот результат:

```
#define  MAXLINE  1000  /* maximum input line size */

main()
{       /* find longest line */
  int len;      /* current line length */
```

```
  int max;      /* maximum length seen so far */
  char line[MAXLINE];   /* current input line */
  char save[MAXLINE];   /* longest line, saved */

  max = 0;
  while ((len = getline(line, MAXLINE)) > 0)
    if (len > max) {
      max = len;
      copy(line, save);
    }
  if (max > 0)  /* there was a line */
    printf("%s", save);
}

getline(s, lim) /* get line into s,return length */
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)    /* copy s1 to s2; assume s2 big enough */
char s1[], s2[];
{
  int i;

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

Функция `main` и `getline` общаются как через пару аргументов, так и через возвращаемое значение. Аргументы `getline` описаны в строках

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

которые указывают, что первый аргумент является массивом, а второй – целым.

Длина массива s не указана, так как она определена в `main`. функция `getline` использует оператор `return` для передачи значения назад в вызывающую программу точно так же, как это делала функция `power`. Одни функции возвращают некоторое нужное значение; другие, подобно `copy`, используются из-за их действия и не возвращают никакого значения.

Чтобы пометить конец строки символов, функция `getline` помещает в конец создаваемого ей массива символ \0 (нулевой символ, значение которого равно нулю). Это соглашение используется также компилятором с языка **C**: когда в **C** программе встречается строчная константа типа

```
"HELLO\n"
```

то компилятор создаёт массив символов, содержащий символы этой строки, и заканчивает его символом \0, с тем чтобы функции, подобные `printf`, могли зафиксировать конец массива:

H	E	L	L	O	\n	\0

Спецификация формата %s указывает, что `printf` ожидает строку, представленную в такой форме. Проанализировав функцию `copy`, вы обнаружите, что и она опирается на тот факт, что её входной аргумент оканчивается символом \0, и копирует этот символ в выходной аргумент s2.[1]

Между прочим, стоит отметить, что даже в такой маленькой программе, как эта, возникает несколько неприятных организационных проблем. Например, что должна делать `main`, если она встретит строку, превышающую её максимально возможный размер? Функция `getline` поступает разумно: при заполнении массива она прекращает дальнейшее извлечение символов, даже если не встречает символа новой строки. Проверив полученную длину и последний символ, функция `main` может установить, не была ли эта строка слишком длинной, и поступить затем, как она сочтёт нужным. Ради краткости мы опустили эту проблему.

Пользователь функции `getline` никак не может заранее узнать, насколько длинной окажется вводимая строка. Поэтому в `getline` включён контроль переполнения. В то же время пользователь функции `copy` уже знает (или может узнать), каков размер строк, так что мы предпочли не включать в эту функцию дополнительный контроль.

Упражнение 1–13

Переделайте ведущую часть программы поиска самой длинной строки таким образом, чтобы она правильно печатала длины сколь угодно длинных вводимых строк и возможно больший текст.

Упражнение 1–14

Напишите программу печати всех строк длиннее 80 символов.

[1] Все это подразумевает, что символ \0 не является частью нормального текста.

Упражнение 1–15

Напишите программу, которая будет удалять из каждой строки стоящие в конце пробелы и табуляции, а также строки, целиком состоящие из пробелов.

Упражнение 1–16

Напишите функцию `reverse(s)`, которая располагает символьную строку s в обратном порядке. С её помощью напишите программу, которая обратит каждую строку из файла ввода.

1.10 Область действия: внешние переменные

Переменные в `main`(line, save и т.д.) являются внутренними или локальными по отношению к функции `main`, потому что они описаны внутри `main` и никакая другая функция не имеет к ним прямого доступа. Это же верно и относительно переменных в других функциях; например, переменная i в функции `getline` никак не связана с i в `copy`. Каждая локальная переменная существует только тогда, когда произошло обращение к соответствующей функции, и исчезает, как только закончится выполнение этой функции. По этой причине такие переменные, следуя терминологии других языков, обычно называют автоматическими. Мы впредь будем использовать термин автоматические при ссылке на эти динамические локальные переменные.[1]

Поскольку автоматические переменные появляются и исчезают вместе с обращением к функции, они не сохраняют своих значений в промежутке от одного вызова до другого, в силу чего им при каждом входе нужно явно присваивать значения. Если этого не сделать, то они будут содержать мусор.

В качестве альтернативы к автоматическим переменным можно определить переменные, которые будут внешними для всех функций, т.е. глобальными переменными, к которым может обратиться по имени любая функция, которая пожелает это сделать[2]. Так как внешние переменные доступны всюду, их можно использовать вместо списка аргументов для передачи данных между функциями. Кроме того, поскольку внешние переменные существуют постоянно, а не появляются и исчезают вместе с вызываемыми функциями, они сохраняют свои значения и после того, как функции, присвоившие им эти значения, завершат свою работу.

Внешняя переменная должна быть определена вне всех функций; при этом ей выделяется фактическое место в памяти. Такая переменная должна быть также описана в каждой функции, которая собирается её использовать; это можно сделать либо явным описанием `extern`, либо неявным по контексту. Чтобы сделать обсуждение более конкретным, давайте перепишем программу поиска самой длинной строки, сделав line, save и max внешними переменными. Это потребует изменения описаний и тел всех трёх функций, а также обращений к ним.

[1] В главе 4 обсуждается класс статической памяти, когда локальные переменные все же оказываются в состоянии сохранить свои значения между обращениями к функциям.

[2] Этот механизм весьма сходен с COMMON в фортране и EXTERNAL в PL/1.

```
#define MAXLINE 1000    /* max. input line size */

char line[MAXLINE];     /* input line */
char save[MAXLINE];     /* longest line saved here */
int max;        /* length of longest line seen so far */

main()
{       /* find longest line; specialized version */
  int len;
  extern int max;
  extern char save[];
  max = 0;

  while ((len = getline()) > 0)
    if (len > max) {
      max = len;
      copy();
    }
  if (max > 0)  /* there was a line */
    printf("%s", save);
}

getline()
{       /* specialized version */
  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] = '\0';
    ++i;
  }
  return (i);
}

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

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

Внешние переменные для функций `main`, `getline` и `copy` определены в первых строчках приведённого выше примера, которыми указывается их тип и вызывается отведение для них памяти. Синтаксически внешние описания точно такие же, как описания, которые мы использовали ранее, но так как они расположены вне функций, соответствующие переменные являются внешними. Чтобы функция могла использовать внешнюю переменную, ей надо сообщить её имя. Один способ сделать это включить в функцию описание `extern`; это описание отличается от предыдущих только добавлением ключевого слова `extern`.

В определённых ситуациях описание `extern` может быть опущено: если внешнее определение переменной находится в том же исходном файле, раньше её использования в некоторой конкретной функции, то не обязательно включать описание `extern` для этой переменной в саму функцию. Описания `extern` в функциях `main`, `getline` и `copy` являются, таким образом, излишними. Фактически, обычная практика заключается в помещении определений всех внешних переменных в начале исходного файла и последующем опускании всех описаний `extern`.

Если программа находится в нескольких исходных файлах, и некоторая переменная определена, скажем в файле 1, а используется в файле 2, то чтобы связать эти два вхождения переменной, необходимо в файле 2 использовать описание `extern`. Этот вопрос подробно обсуждается в главе 4.

Вы должно быть заметили, что мы в этом разделе при ссылке на внешние переменные очень аккуратно используем слова описание и определение. «Определение» относится к тому месту, где переменная фактически заводится и ей выделяется память; «описание» относится к тем местам, где указывается природа переменной, но никакой памяти не отводится.

Между прочим, существует тенденция объявлять все, что ни попадётся, внешними переменными, поскольку кажется, что это упрощает связи, – списки аргументов становятся короче и переменные всегда присутствуют, когда бы вам они ни понадобились. Но внешние переменные присутствуют и тогда, когда вы в них не нуждаетесь. Такой стиль программирования чреват опасностью, так как он приводит к программам, связи данных внутри которых не вполне очевидны. Переменные при этом могут изменяться неожиданным и даже неумышленным образом, а программы становится трудно модифицировать, когда возникает такая необходимость. Вторая версия программы поиска самой длинной строки уступает первой отчасти по этим причинам, а отчасти потому, что она лишила универсальности две весьма полезные функции, введя в них имена переменных, с которыми они будут манипулировать.

Упражнение 1–17

Проверка в операторе `for` функции `getline` довольно неуклюжа. Перепишите программу таким образом, чтобы сделать эту проверку более ясной, но сохраните при этом то же самое поведение в конце файла и при переполнении буфера. Является ли это поведение самым разумным?

1.11 Резюме

На данном этапе мы обсудили то, что можно бы назвать традиционным ядром языка **C**. Имея эту горсть строительных блоков, можно писать полезные программы весьма значительного размера, и было бы вероятно неплохой идеей, если бы вы задержались здесь на какое-то время и поступили таким образом: следующие ниже упражнения предлагают вам ряд программ несколько большей сложности, чем те, которые были приведены в этой главе.

После того как вы овладеете этой частью **C**, приступайте к чтению следующих нескольких глав. Усилия, которые вы при этом затратите, полностью окупятся, потому что в этих главах обсуждаются именно те стороны **C**, где мощь и выразительность языка начинает становиться очевидной.

Упражнение 1–18

Напишите программу `detab`, которая заменяет табуляции во вводе на нужное число пробелов так, чтобы промежуток достигал следующей табуляционной остановки. Предположите фиксированный набор табуляционных остановок, например, через каждые n позиций.

Упражнение 1–19

Напишите программу `entab`, которая заменяет строки пробелов минимальным числом табуляций и пробелов, достигая при этом тех же самых промежутков. Используйте те же табуляционные остановки, как и в `detab`.

Упражнение 1–20

Напишите программу для «сгибания» длинных вводимых строк после последнего отличного от пробела символа, стоящего до столбца n ввода, где n – параметр. Убедитесь, что ваша программа делает что-то разумное с очень длинными строками и в случае, когда перед указанным столбцом нет ни табуляций, ни пробелов.

Упражнение 1–21

Напишите программу удаления из **C**-программы всех комментариев. Не забывайте аккуратно обращаться с «закавыченными» строками и символьными константами.

Упражнение 1–22

Напишите программу проверки **C**-программы на элементарные синтаксические ошибки, такие как несоответствие круглых, квадратных и фигурных скобок. Не забудьте о кавычках, как одиночных, так и двойных, и о комментариях. (Эта программа весьма сложна, если вы будете писать её для самого общего случая).

2 Типы, операции и выражения

Переменные и константы являются основными объектами, с которыми оперирует программа. Описания перечисляют переменные, которые будут использоваться, указывают их тип и, возможно, их начальные значения. Операции определяют, что с ними будет сделано. Выражения объединяют переменные и константы для получения новых значений. Все это – темы настоящей главы.

2.1 Имена переменных

Хотя мы этого сразу прямо не сказали, существуют некоторые ограничения на имена переменных и символических констант. Имена составляются из букв и цифр; первый символ должен быть буквой. Подчёркивание «_» тоже считается буквой; это полезно для удобочитаемости длинных имён переменных. Прописные и строчные буквы различаются; традиционная практика в **C** – использовать строчные буквы для имён переменных, а прописные – для символических констант.

Играют роль только первые восемь символов внутреннего имени, хотя использовать можно и больше. Для внешних имён, таких как имена функций и внешних переменных, это число может оказаться меньше восьми, так как внешние имена используются различными ассемблерами и загрузчиками.[1] Кроме того, такие ключевые слова как `if`, `else`, `int`, `float` и т.д., зарезервированы: вы не можете использовать их в качестве имён переменных.

Конечно, разумно выбирать имена переменных таким образом, чтобы они означали нечто, относящееся к назначению переменных, и чтобы было менее вероятно спутать их при написании.

2.2 Типы и размеры данных

Языке **C** имеется только несколько основных типов данных:

`char`	–	один байт, в котором может находиться один символ из внутреннего набора символов;
`int`	–	целое, обычно соответствующее естественному размеру целых в используемой машине;
`float`	–	число с плавающей точкой одинарной точности;
`double`	–	число с плавающей точкой двойной точности.

[1] Детали приводятся в приложении А.

Кроме того имеется ряд квалификаторов, которые можно использовать с типом `int`: `short` (короткое), `long` (длинное) и `unsigned` (без знака). Квалификаторы `short` и `long` указывают на различные размеры целых. Числа без знака подчиняются законам арифметики по модулю 2 в степени n, где n – число битов в `int`; числа без знаков всегда положительны. Описания с квалификаторами имеют вид:

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

Слово `int` в таких ситуациях может быть опущено, что обычно и делается.

Количество битов, отводимых под эти объекты зависит от имеющейся машины; в таблице ниже приведены некоторые характерные значения.

	DEC PDP–11	HONEYWELL 6000	IBM 370	INTERDATA 8/32
`char`	ASCII 8-BITS	ASCII 9-BITS	EBCDIC 8-BITS	ASCII 8-BITS
`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

Цель состоит в том, чтобы `short` и `long` давали возможность в зависимости от практических нужд использовать различные длины целых; тип `int` отражает наиболее «естественный» размер конкретной машины. Как вы видите, каждый компилятор свободно интерпретирует `short` и `long` в соответствии со своими аппаратными средствами. Все, на что вы можете твёрдо полагаться, это то, что `short` не длиннее, чем `long`.

2.3 Константы

Константы типа `int` и `float` мы уже рассмотрели. Отметим ещё только, что как обычная `123.456e-7`, так и «научная» запись `0.12e3` для `float` является законной.

Каждая константа с плавающей точкой считается имеющей тип `double`, так что обозначение «e» служит как для `float`, так и для `double`.

Длинные константы записываются в виде 123L. Обычная целая константа, которая слишком длинна для типа `int`, рассматривается как `long`.

Существует система обозначений для восьмеричных и шестнадцатеричных констант: лидирующий 0(нуль) в константе типа `int` указывает на восьмеричную константу, а стоящие впереди `0x` соответствуют шестнадцатеричной константе. Например, десятичное число 31 можно записать как 037 в восьмеричной форме и как `0x1F` в шестнадцатеричной. Шестнадцатеричные и восьмеричные константы могут также заканчиваться буквой L, что делает их относящимися к типу `long`.

2.3.1 Символьная константа

Символьная константа – это один символ, заключённый в одинарные кавычки, как, например, 'x'. Значением символьной константы является численное значение этого символа во внутреннем машинном наборе символов. Например, в наборе символов ASCII символьный нуль, или '0', имеет значение 48, а в коде EBCDIC – 240, и оба эти значения совершенно отличны от числа 0. Написание '0' вместо численного значения, такого как 48 или 240, делает программу не зависящей от конкретного численного представления этого символа в данной машине. Символьные константы точно так же участвуют в численных операциях, как и любые другие числа, хотя наиболее часто они используются в сравнении с другими символами. Правила преобразования будут изложены позднее.

Некоторые неграфические символы могут быть представлены как символьные константы с помощью условных последовательностей, как, например, \n (новая строка), \t (табуляция), \0 (нулевой символ), \\ (обратная косая черта), \' (одинарная кавычка) и т.д. Хотя они выглядят как два символа, на самом деле являются одним. Кроме того, можно сгенерировать произвольную последовательность двоичных знаков размером в байт, если написать

```
'\ddd'
```

где ddd – от одной до трёх восьмеричных цифр, как в

```
#define  FORMFEED  '\014'       /* form feed */
```

Символьная константа '\0', изображающая символ со значением 0, часто записывается вместо целой константы 0, чтобы подчеркнуть символьную природу некоторого выражения.

2.3.2 Константное выражение

Константное выражение – это выражение, состоящее из одних констант. Такие выражения обрабатываются во время компиляции, а не при прогоне программы, и соответственно могут быть использованы в любом месте, где можно использовать константу, как, например в

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

или

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

2.3.3 Строчная константа

Строчная константа – это последовательность, состоящая из нуля или более символов, заключённых в двойные кавычки, как, например,

```
"I am a string" /* я - строка */
```

или

```
""      /* нуль-строка */
```

Кавычки не являются частью строки, а служат только для её ограничения. Те же самые условные последовательности, которые использовались в символьных константах, применяются и в строках; символ двойной кавычки изображается как \".

С технической точки зрения строка представляет собой массив, элементами которого являются отдельные символы. Чтобы программам было удобно определять конец строки, компилятор автоматически помещает в конец каждой строки нуль-символ \0. Такое представление означает, что не накладывается конкретного ограничения на то, какую длину может иметь строка, и чтобы определить эту длину, программы должны просматривать строку полностью. При этом для физического хранения строки требуется на одну ячейку памяти больше, чем число заключённых в кавычки символов. Следующая функция `strlen(s)` вычисляет длину символьной строки s не считая конечный символ \0.

```
strlen(s)       /* return length of s */
char s[];
{
  int i;

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

Будьте внимательны и не путайте символьную константу со строкой, содержащей один символ: 'X' – это не то же самое, что "X". Первое – это отдельный символ, использованный с целью получения численного значения, соответствующего букве X в машинном наборе символов. Второе – символьная строка, состоящая из одного символа (буква X) и \0.

2.4 Описания

Все переменные должны быть описаны до их использования, хотя некоторые описания делаются неявно, по контексту. Описание состоит из спецификатора типа и следующего за ним списка переменных, имеющих этот тип, как, например,

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

Переменные можно распределять по описаниям любым образом; приведённые выше списки можно с тем же успехом записать в виде

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

Такая форма занимает больше места, но она удобна для добавления комментария к каждому описанию и для последующих модификаций.

Переменным могут быть присвоены начальные значения внутри их описания, хотя здесь имеются некоторые ограничения. Если за именем переменной следуют знак равенства и константа, то эта константа служит в качестве инициализатора, как, например:

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

Если рассматриваемая переменная является внешней или статической, то инициализация проводится только один раз, согласно концепции до начала выполнения программы. Инициализируемым явно автоматическим переменным начальные значения присваиваются при каждом обращении к функции, в которой они описаны. Автоматические переменные, не инициализируемые явно, имеют неопределённые значения, (т.е. мусор). Внешние и статические переменные по умолчанию инициализируются нулём, но, тем не менее, их явная инициализация является признаком хорошего стиля.

Мы продолжим обсуждение вопросов инициализации, когда будем описывать новые типы данных.

2.5 Арифметические операции

Бинарными арифметическими операциями являются +, −, *, / и % (операция деления по модулю). Имеется унарная операция −, но не существует унарной операции +.

При делении целых дробная часть отбрасывается. Выражение

```
x % y
```

даёт остаток от деления x на y и, следовательно, равно нулю, когда x делится на y точно. Например, год является високосным, если он делится на 4, но не делится на 100, исключая то, что делящиеся на 400 годы тоже являются високосными. Поэтому

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

Операцию % нельзя использовать с типами `float` или `double`.

Операции + и − имеют одинаковое старшинство, которое младше одинакового уровня старшинства операций *, / и %, которые в свою очередь младше унарного минуса.

Арифметические операции группируются слева направо.[1] Порядок выполнения ассоциативных и коммутативных операций типа + и − не фиксируется; компилятор может перегруппировывать даже заключённые в круглые скобки выражения, связанные такими операциями. Таким образом, a+(b+c) может быть вычислено как (a+b)+c. Это редко приводит к какому-либо расхождению, но если необходимо обеспечить строго определённый порядок, то нужно использовать явные промежуточные переменные.

Действия, предпринимаемые при переполнении и антипереполнении (т.е. при получении слишком маленького по абсолютной величине числа), зависят от используемой машины.

2.6 Операции отношения и логические операции

Операциями отношения являются

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

все они имеют одинаковое старшинство. Непосредственно за ними по уровню старшинства следуют операции равенства и неравенства:

```
==   !=
```

которые тоже имеют одинаковое старшинство. Операции отношения младше арифметических операций, так что выражения типа `i < lim-1` понимаются как `i < (lim-1)`, как и предполагается.

Логические связки && и || более интересны. Выражения, связанные операциями && и ||, вычисляются слева направо, причём их рассмотрение прекращается сразу же как только становится ясно, будет ли результат истиной или ложью. Учёт этих свойств очень существенен для написания правильно работающих программ. Рассмотрим, например, оператор цикла в считывающей строку функции `getline`, которую мы написали в главе 1.

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

Ясно, что перед считыванием нового символа необходимо проверить, имеется ли ещё место в массиве s, так что условие `i<lim-1` должно проверяться первым. И если это условие не выполняется, мы не должны считывать следующий символ.

Так же неудачным было бы сравнение c с `EOF` до обращения к функции `getchar`: прежде чем проверять символ, его нужно считать.

Старшинство операции && выше, чем у ||, и обе они младше операций отношения и равенства. Поэтому такие выражения, как

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

[1] Сведения о старшинстве и ассоциативности всех операций собраны в таблице в конце этой главы.

не нуждаются в дополнительных круглых скобках. Но так как операция != старше операции присваивания, то для достижения правильного результата в выражении

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

скобки необходимы.

Унарная операция отрицания ! преобразует ненулевой или истинный операнд в 0, а нулевой или ложный операнд в 1. Обычное использование операции ! заключается в записи

```
if (!inword)
```

Вместо

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

Трудно сказать, какая форма лучше. Конструкции типа `(!inword)` читаются довольно удобно («если не в слове»). Но в более сложных случаях они могут оказаться трудными для понимания.

Упражнение 2–1

Напишите оператор цикла, эквивалентный приведённому выше оператору `for`, не используя операции &&.

2.7 Преобразование типов

Если в выражениях встречаются операнды различных типов, то они преобразуются к общему типу в соответствии с небольшим набором правил. В общем, автоматически производятся только преобразования, имеющие смысл, такие как, например, преобразование целого в плавающее в выражениях типа f+i. Выражения же, лишённые смысла, такие как использование переменной типа `float` в качестве индекса, запрещены.

Во-первых, типы `char` и `int` могут свободно смешиваться в арифметических выражениях: каждая переменная типа `char` автоматически преобразуется в `int`. Это обеспечивает значительную гибкость при проведении определённых преобразований символов. Примером может служить функция atoi, которая ставит в соответствие строке цифр её численный эквивалент.

```
atoi(s) /* convert s to integer */
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 уже обсуждалось в главе 1, выражение

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

имеет численное значение находящегося в s[i] символа, потому что значение символов '0', '1' и т.д. образуют возрастающую последовательность расположенных подряд целых положительных чисел.

Другой пример преобразования `char` в `int` даёт функция `lower`, преобразующая данную прописную букву в строчную. Если выступающий в качестве аргумента символ не является прописной буквой, то `lower` возвращает его неизменным. Приводимая ниже программа справедлива только для набора символов ASCII.

```
lower(c)        /* convert c to lower case; ascii only */
int c;
{
  if (c >= 'A' && c <= 'Z')
    return (c + 'a' - 'A');     /* 'a' записано вместо 'A' строчного */
  else
    return (c);
}
```

Эта функция правильно работает при коде ASCII, потому что численные значения, соответствующие в этом коде прописным и строчным буквам, отличаются на постоянную величину, а каждый алфавит является сплошным – между A и Z нет ничего, кроме букв. Это последнее замечание для набора символов EBCDIC систем IBM 360/370 оказывается несправедливым, в силу чего эта программа на таких системах работает неправильно – она преобразует не только буквы.

При преобразовании символьных переменных в целые возникает один тонкий момент. Дело в том, что сам язык не указывает, должны ли переменным типа `char` соответствовать численные значения со знаком или без знака. Может ли при преобразовании `char` в `int` получиться отрицательное целое? К сожалению, ответ на этот вопрос меняется от машины к машине, отражая расхождения в их архитектуре. На некоторых машинах (PDP-11, например) переменная типа `char`, крайний левый бит которой содержит 1, преобразуется в отрицательное целое («знаковое расширение»). На других машинах такое преобразование сопровождается добавлением нулей с левого края, в результате чего всегда получается положительное число.

Определение языка **C** гарантирует, что любой символ из стандартного набора символов машины никогда не даст отрицательного числа, так что эти символы можно свободно использовать в выражениях как положительные величины. Но произвольные комбинации двоичных знаков, хранящиеся как символьные переменные на некоторых машинах, могут дать отрицательные значения, а на других положительные.

Наиболее типичным примером возникновения такой ситуации является случай, когда значение -1 используется в качестве `EOF`. Рассмотрим программу

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

На машине, которая не осуществляет знакового расширения, переменная с всегда положительна, поскольку она описана как `char`, а так как `EOF` отрицательно, то условие никогда не выполняется. Чтобы избежать такой ситуации, мы всегда предусмотрительно использовали `int` вместо `char` для любой переменной, получающей значение от `getchar`.

Основная же причина использования `int` вместо `char` не связана с каким-либо вопросом о возможном знаковом расширении, просто функция `getchar` должна передавать все возможные символы (чтобы её можно было использовать для произвольного ввода) и, кроме того, отличающееся значение `EOF`. Следовательно значение `EOF` не может быть представлено как `char`, а должно храниться как `int`.

Другой полезной формой автоматического преобразования типов является то, что выражения отношения, подобные `i>j`, и логические выражения, связанные операциями && и ||, по определению имеют значение 1, если они истинны, и 0, если они ложны. Таким образом, присваивание

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

полагает ifdigit равным 1, если c – цифра, и равным 0 в противном случае.[1]

Неявные арифметические преобразования работают в основном, как и ожидается. В общих чертах, если операция типа + или *, которая связывает два операнда (бинарная операция), имеет операнды разных типов, то перед выполнением операции «низший» тип преобразуется к «высшему» и получается результат «высшего» типа. Более точно, к каждой арифметической операции применяется следующая последовательность правил преобразования.

- Типы `char` и `short` преобразуются в `int`, а `float` в `double`.
- Затем, если один из операндов имеет тип `double`, то другой преобразуется в `double`, и результат имеет тип `double`.
- В противном случае, если один из операндов имеет тип `long`, то другой преобразуется в `long`, и результат имеет тип `long`.
- В противном случае, если один из операндов имеет тип `unsigned`, то другой преобразуется в `unsigned` и результат имеет тип `unsigned`.
- В противном случае операнды должны быть типа `int`, и результат имеет тип `int`. Подчеркнём, что все переменные типа `float` в выражениях преобразуются в `double`; в **C** вся плавающая арифметика выполняется с двойной точностью.

Преобразования возникают и при присваиваниях; значение правой части преобразуется к типу левой, который и является типом результата. Символьные переменные преобразуются в целые либо со знаковым расширением, либо без него, как описано выше. Обратное преобразование `int` в `char` ведёт себя хорошо лишние биты высокого порядка просто отбрасываются. Таким образом

[1] В проверочной части операторов `if`, `while`, `for` и т.д. «истинно» просто означает «не нуль».

```
int i;
char c;
i = c;
c = i;
```

значение c не изменяется. Это верно независимо от того, вовлекается ли знаковое расширение или нет.

Если x типа `float`, а i типа `int`, то как

```
x = i;
```

так и

```
i = x;
```

приводят к преобразованиям; при этом `float` преобразуется в `int` отбрасыванием дробной части. Тип `double` преобразуется во `float` округлением. Длинные целые преобразуются в более короткие целые и в переменные типа `char` посредством отбрасывания лишних битов высокого порядка.

Так как аргумент функции является выражением, то при передаче функциям аргументов также происходит преобразование типов: в частности, `char` и `short` становятся `int`, а `float` становится `double`. Именно поэтому мы описывали аргументы функций как `int` и `double` даже тогда, когда обращались к ним с переменными типа `char` и `float`.

Наконец, в любом выражении может быть осуществлено («принуждено») явное преобразование типа с помощью конструкции, называемой перевод (cast). В этой конструкции, имеющей вид

```
(имя типа) выражение
```

Выражение преобразуется к указанному типу по правилам преобразования, изложенным выше. Фактически точный смысл операции перевода можно описать следующим образом: выражение как бы присваивается некоторой переменной указанного типа, которая затем используется вместо всей конструкции. Например, библиотечная процедура `sqrt` ожидает аргумента типа `double` и выдаст бессмысленный ответ, если к ней по небрежности обратятся с чем-нибудь иным. Таким образом, если n целое, то выражение

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

до передачи аргумента функции `sqrt` преобразует n к типу `double`.[1] Операция преобразования типа (cast) имеет тот же уровень старшинства, что и другие унарные операции, как указывается в таблице в конце этой главы.

Упражнение 2–2

Составьте программу для функции htoi(s), которая преобразует строку шестнадцатеричных цифр в эквивалентное ей целое значение. При этом допустимыми цифрами являются цифры от 0 до 9 и буквы от A до F.

[1] Отметим, что операция перевод преобразует значение n в надлежащий тип; фактическое содержание переменной n при этом не изменяется.

2.8 Операции увеличения и уменьшения

В языке **C** предусмотрены две необычные операции для увеличения и уменьшения значений переменных. Операция увеличения ++ добавляет 1 к своему операнду, а операция уменьшения −− вычитает 1. Мы часто использовали операцию ++ для увеличения переменных, как, например, в

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

Необычный аспект заключается в том, что ++ и −− можно использовать либо как префиксные операции (перед переменной, как в ++n), либо как постфиксные (после переменной: n++). Эффект в обоих случаях состоит в увеличении n. Но выражение ++n увеличивает переменную n до использования её значения, в то время как n++ увеличивает переменную n после того, как её значение было использовано. Это означает, что в контексте, где используется значение переменной, а не только эффект увеличения, использование ++n и n++ приводит к разным результатам. Если n = 5, то

```
x = n++;
```

устанавливает x равным 5, а

```
x = ++n;
```

полагает x равным 6. В обоих случаях n становится равным 6. Операции увеличения и уменьшения можно применять только к переменным; выражения типа x=(i+j)++ являются незаконными.

В случаях, где нужен только эффект увеличения, а само значение не используется, как, например, в

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

выбор префиксной или постфиксной операции является делом вкуса, но встречаются ситуации, где нужно использовать именно ту или другую операцию. Рассмотрим, например, функцию squeeze(s,c), которая удаляет символ c из строки S, каждый раз, как он встречается.

```
squeeze(s, c)   /* delete all c from s */
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';
}
```

Каждый раз, как встречается символ, отличный от c, он копируется в текущую позицию j, и только после этого j увеличивается на 1, чтобы быть готовым для поступления следующего символа. Это в точности эквивалентно записи

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

Другой пример подобной конструкции даёт функция `getline`, которую мы запрограммировали в главе 1, где можно заменить

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

более компактной записью

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

В качестве третьего примера рассмотрим функцию strcat(s,t), которая приписывает строку t в конец строки s, образуя конкатенацию строк s и t. При этом предполагается, что в s достаточно места для хранения полученной комбинации.

```
strcat(s, t)    /* concatenate t to end of s */
char s[], t[];  /* s must be big enough */
{
  int i, j;

  i = j = 0;
  while (s[i] != '\0')  /* find end of s */
    i++;
  while ((s[i++] = t[j++]) != '\0')     /* copy t */
    ;
}
```

Так как из t в s копируется каждый символ, то для подготовки к следующему прохождению цикла постфиксная операция ++ применяется к обеим переменным i и j.

Упражнение 2–3

Напишите другой вариант функции squeeze(s1,s2), который удаляет из строки s1 каждый символ, совпадающий с каким-либо символом строки s2.

Упражнение 2–4

Напишите программу для функции any(s1,s2), которая находит место первого появления в строке s1 какого-либо символа из строки s2 и, если строка s1 не содержит символов строки s2, возвращает значение `-1`.

2.9 Побитовые логические операции

В языке предусмотрен ряд операций для работы с битами; эти операции нельзя применять к переменным типа `float` или `double`.

&	побитовое AND;
|	побитовое включающее OR;
^	побитовое исключающее OR;
≪	сдвиг влево;
≫	сдвиг вправо;
~	дополнение (унарная операция).

Побитовая операция & часто используется для маскирования некоторого множества битов; например, оператор

```
c = n & 0177
```

передаёт в c семь младших битов n, полагая остальные равными нулю. Операция | побитового OR используется для включения битов:

```
c = x | MASK
```

устанавливает на единицу те биты в x, которые равны единице в MASK.

Следует быть внимательным и отличать побитовые операции & и | от логических связок && и ||, которые подразумевают вычисление значения истинности слева направо. Например, если x=1, а y=2, то значение x&y равно нулю, в то время как значение x&&y равно единице. (Почему?)

Операции сдвига ≪ и ≫ осуществляют соответственно сдвиг влево и вправо своего левого операнда на число битовых позиций, задаваемых правым операндом. Таким образом, x≪2 сдвигает x влево на две позиции, заполняя освобождающиеся биты нулями, что эквивалентно умножению на 4. Сдвиг вправо величины без знака заполняет освобождающиеся биты на некоторых машинах, таких как PDP-11, заполняются содержанием знакового бита («арифметический сдвиг»), а на других – нулём («логический сдвиг»).

Унарная операция ~ даёт дополнение к целому; это означает, что каждый бит со значением 1 получает значение 0 и наоборот. Эта операция обычно оказывается полезной в выражениях типа

```
x & ~077
```

где последние шесть битов x маскируются нулём. Подчеркнём, что выражение x&~077 не зависит от длины слова и поэтому предпочтительнее, чем, например, x&0177700, где предполагается, что x занимает 16 битов. Такая переносимая форма не требует никаких дополнительных затрат, поскольку ~077 является константным выражением и, следовательно, обрабатывается во время компиляции.

Чтобы проиллюстрировать использование некоторых операций с битами, рассмотрим функцию getbits(x,p,n), которая возвращает поле в n битов, вырезанное из x, начиная с позиции p и сдвигая его к правому краю. Мы предполагаем, что крайний правый бит имеет номер 0, и что n и p – разумно заданные положительные числа. Например, getbits(x,4,3) возвращает сдвинутыми к правому краю биты, занимающие позиции 4,3 и 2.

```
getbits(x, p, n)        /* get n bits from position p */
unsigned x, p, n;
{
  return ((x >> (p + 1 - n)) & ~(\0 << n));
}
```

Операция `x >> (p+1-n)` сдвигает желаемое поле в правый конец слова. Описание аргумента x как `unsigned` гарантирует, что при сдвиге вправо освобождающиеся биты будут заполняться нулями, а не содержимым знакового бита, независимо от того, на какой машине пропускается программа. Все биты константного выражения \0 равны 1; сдвиг его на n позиций влево с помощью операции `\0 << n` создаёт маску с нулями в n крайних правых битах и единицами в остальных; дополнение '~' создаёт маску с единицами в n крайних правых битах.

Упражнение 2–5

Переделайте getbits таким образом, чтобы биты отсчитывались слева направо.

Упражнение 2–6

Напишите программу для функции wordlength(), вычисляющей длину слова используемой машины, т.е. число битов в переменной типа `int`. Функция должна быть переносимой, т.е. одна и та же исходная программа должна правильно работать на любой машине.

Упражнение 2–7

Напишите программу для функции rightrot(n,b), сдвигающей циклически целое n вправо на b битовых позиций.

Упражнение 2–8

Напишите программу для функции invert(x,p,n), которая инвертирует (т.е. заменяет 1 на 0 и наоборот) n битов x, начинающихся с позиции p, оставляя другие биты неизмененными.

2.10 Операции и выражения присваивания

Такие выражения, как

```
i = i + 2
```

в которых левая часть повторяется в правой части могут быть записаны в сжатой форме

```
i += 2
```

используя операцию присваивания вида +=.

Большинству бинарных операций (операций подобных +, которые имеют левый и правый операнд) соответствует операция присваивания вида оп=, где оп – одна из операций

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

Если e1 и e2 – выражения, то

```
e1 оп= e2
```

эквивалентно

```
e1 = (e1) оп (e2)
```

за исключением того, что выражение e1 вычисляется только один раз. Обратите внимание на круглые скобки вокруг e2:

```
x *= y + 1
```

то

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

не

```
x = x * y + 1
```

В качестве примера приведём функцию bitcount, которая подсчитывает число равных 1 битов у целого аргумента.

```
bitcount(n)     /* count 1 bits in n */
unsigned n;
{
  int b;
  for (b = 0; n != 0; n >>= 1)
    if (n & 01)
      b++;
  return (b);
}
```

Не говоря уже о краткости, такие операторы присваивания имеют то преимущество, что они лучше соответствуют образу человеческого мышления. Мы говорим: «прибавить 2 к i» или «увеличить i на 2», но не «взять i, прибавить 2 и поместить результат опять в i». Итак, i += 2. Кроме того, в громоздких выражениях, подобных

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

Такая операция присваивания облегчает понимание программы, так как читатель не должен скрупулёзно проверять, являются ли два длинных выражения действительно одинаковыми, или задумываться, почему они не совпадают. Такая операция присваивания может даже помочь компилятору получить более эффективную программу.

Мы уже использовали тот факт, что операция присваивания имеет некоторое значение и может входить в выражения; самый типичный пример

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

присваивания, использующие другие операции присваивания (`+=`, `-=` и т.д.) также могут входить в выражения, хотя это случается реже.

Типом выражения присваивания является тип его левого операнда.

Упражнение 2–9

В двоичной системе счисления операция x&(x-1) обнуляет самый правый равный 1 бит переменной x.(Почему?) Используйте это замечание для написания более быстрой версии функции bitcount.

2.11 Условные выражения

Операторы

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

конечно вычисляют z как максимум из a и b. Условное выражение, записанное с помощью тернарной операции « ? : », предоставляет другую возможность для записи этой и аналогичных конструкций. В выражении

```
e1 ? e2 : e3
```

сначала вычисляется выражение e1. Если оно отлично от нуля (истинно), то вычисляется выражение e2, которое и становится значением условного выражения. В противном случае вычисляется e3, и оно становится значением условного выражения. Каждый раз вычисляется только одно из выражения e2 и e3. Таким образом, чтобы положить z равным максимуму из a и b, можно написать

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

Следует подчеркнуть, что условное выражение действительно является выражением и может использоваться точно так же, как любое другое выражение. Если e2 и e3 имеют разные типы, то тип результата определяется по правилам преобразования, рассмотренным ранее в этой главе. Например, если f имеет тип `float`, а n – тип `int`, то выражение

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

Имеет тип `double` независимо от того, положительно ли n или нет.

Так как уровень старшинства операции « ? : » очень низок, прямо над присваиванием, то первое выражение в условном выражении можно не заключать в круглые скобки. Однако, мы все же рекомендуем это делать, так как скобки делают условную часть выражения более заметной.

Использование условных выражений часто приводит к коротким программам. Например, следующий ниже оператор цикла печатает n элементов массива, по 10 в строке, разделяя каждый столбец одним пробелом и заканчивая каждую строку (включая последнюю) одним символом перевода строки.

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

Символ перевода строки записывается после каждого десятого элемента и после n-го элемента. За всеми остальными элементами следует один пробел. Хотя, возможно, это выглядит мудрёным, было бы поучительным попытаться записать это, не используя условного выражения.

Упражнение 2–10

Перепишите программу для функции `lower`, которая переводит прописные буквы в строчные, используя вместо конструкции `if– else` условное выражение.

2.12 Старшинство и порядок вычисления

В приводимой ниже таблице сведены правила старшинства и ассоциативности всех операций, включая и те, которые мы ещё не обсуждали. Операции, расположенные в одной строке, имеют один и тот же уровень старшинства; строки расположены в порядке убывания старшинства. Так, например, операции `*,/` и % имеют одинаковый уровень старшинства, который выше, чем уровень операций + и −.

Операции −> и « . » используются для доступа к элементам структур; они будут описаны в главе 6 вместе с sizeof (размер объекта). В главе 5 обсуждаются операции * (косвенная адресация) и & (адрес). Отметим, что уровень старшинства побитовых логических операций &, ^ и ~ ниже уровня операций == и `!=`. Это приводит к тому, что осуществляющие побитовую проверку выражения, подобные

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

для получения правильных результатов должны заключаться в круглые скобки.

Как уже отмечалось ранее, выражения, в которые входит одна из ассоциативных и коммутативных операций (*, +, &, ^, ~), могут перегруппировываться, даже если они заключены в круглые скобки. В большинстве случаев это не приводит к каким бы то ни

operator	associativity
`() [ ] -> .`	left to right
`! ~ ++ -- + - (cast) * & sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`? :`	right to left
`= += -=` etc.	right to left
`,`	left to right

было расхождениям; в ситуациях, где такие расхождения все же возможны, для обеспечения нужного порядка вычислений можно использовать явные промежуточные переменные.

В языке **C**, как и в большинстве языков, не фиксируется порядок вычисления операндов в операторе. Например в операторе вида

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

сначала может быть вычислено f, а потом g, и наоборот; поэтому, если либо f, либо g изменяют внешнюю переменную, от которой зависит другой операнд, то значение x может зависеть от порядка вычислений. Для обеспечения нужной последовательности промежуточные результаты можно опять запоминать во временных переменных.

Подобным же образом не фиксируется порядок вычисления аргументов функции, так что оператор

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

может давать (и действительно даёт) на разных машинах разные результаты в зависимости от того, увеличивается ли n до или после обращения к функции `power`. Правильным решением, конечно, является запись

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

Обращения к функциям, вложенные операции присваивания, операции увеличения и уменьшения приводят к так называемым «побочным эффектам» – некоторые переменные изменяются как побочный результат вычисления выражений. В любом выражении, в котором возникают побочные эффекты, могут существовать очень тонкие зависимости

от порядка, в котором определяются входящие в него переменные. Примером типичной неудачной ситуации является оператор

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

Возникает вопрос, старое или новое значение i служит в качестве индекса. Компилятор может поступать разными способами и в зависимости от своей интерпретации выдавать разные результаты. Тот случай, когда происходят побочные эффекты (присваивание фактическим переменным), – оставляется на усмотрение компилятора, так как наилучший порядок сильно зависит от архитектуры машины.

Из этих рассуждений вытекает такая мораль: написание программ, зависящих от порядка вычислений, является плохим методом программирования на любом языке. Конечно, необходимо знать, чего следует избегать, но если вы не в курсе, как некоторые вещи реализованы на разных машинах, это неведение может предохранить вас от неприятностей.[1]

[1] Отладочная программа `lint` укажет большинство мест, зависящих от порядка вычислений.

3 Поток управления

Управляющие операторы языка определяют порядок вычислений. В приведённых ранее примерах мы уже встречались с наиболее употребительными управляющими конструкциями языка **C**; здесь мы опишем остальные операторы управления и уточним действия операторов, обсуждавшихся ранее.

3.1 Операторы и блоки

Такие выражения, как x=0, или i++, или `printf(...)`, становятся операторами, если за ними следует точка с запятой, как, например,

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

В языке **C** точка с запятой является признаком конца оператора, а не разделителем операторов, как в языках типа алгола.

Фигурные скобки { и } используются для объединения описаний и операторов в составной оператор или блок, так что они оказываются синтаксически эквивалентны одному оператору. Один явный пример такого типа дают фигурные скобки, в которые заключаются операторы, составляющие функцию, другой фигурные скобки вокруг группы операторов в конструкциях `if`, `else`, `while` и `for`.[1] Точка с запятой никогда не ставится после первой фигурной скобки, которая завершает блок.

3.2 if – else

Оператор `if-else` используется при необходимости сделать выбор. Формально синтаксис имеет вид

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

где часть `else` является необязательной. Сначала вычисляется выражение; если оно «истинно» (т.е. значение выражения отлично от нуля), то выполняется оператор-1. Если оно

[1] На самом деле переменные могут быть описаны внутри любого блока; мы поговорим об этом в главе 4.

ложно (значение выражения равно нулю), и если есть часть с `else`, то вместо оператора-1 выполняется оператор-2.

Так как `if` просто проверяет численное значение выражения, то возможно некоторое сокращение записи. Самой очевидной возможностью является запись

```
if (выражение)
```

вместо

```
if (выражение!=0)
```

иногда такая запись является ясной и естественной, но временами она становится загадочной.

То, что часть `else` в конструкции `if-else` является необязательной, приводит к двусмысленности в случае, когда `else` опускается во вложенной последовательности операторов `if`. Эта неоднозначность разрешается обычным образом – `else` связывается с ближайшим предыдущим `if`, не содержащим `else`. Например, в

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

конструкция `else` относится к внутреннему `if`, как мы и показали, сдвинув `else` под соответствующий `if`. Если это не то, что вы хотите, то для получения нужного соответствия необходимо использовать фигурные скобки:

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

Такая двусмысленность особенно пагубна в ситуациях типа

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

Запись `else` под `if` ясно показывает, чего вы хотите, но компилятор не получит соответствующего указания и свяжет `else` с внутренним `if`. Ошибки такого рода очень трудно обнаруживаются.

Между прочим, обратите внимание, что в

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

после z=a стоит точка с запятой. Дело в том, что согласно грамматическим правилам за if должен следовать оператор, а выражение типа z=a, являющееся оператором, всегда заканчивается точкой с запятой.

3.3 else – if

Конструкция

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

встречается настолько часто, что заслуживает отдельного краткого рассмотрения. Такая последовательность операторов if является наиболее распространённым способом программирования выбора из нескольких возможных вариантов. Выражения просматриваются последовательно; если какое-то выражение оказывается истинным, то выполняется относящийся к нему оператор, и этим вся цепочка заканчивается. Каждый оператор может быть либо отдельным оператором, либо группой операторов в фигурных скобках.

Последняя часть с else имеет дело со случаем, когда ни одно из проверяемых условий не выполняется. Иногда при этом не надо предпринимать никаких явных действий; в этом случае хвост

```
else
  оператор
```

может быть опущен, или его можно использовать для контроля, чтобы засечь «невозможное» условие.

Для иллюстрации выбора из трёх возможных вариантов приведём программу функции, которая методом половинного деления определяет, находится ли данное значение x в отсортированном массиве v. Элементы массива v должны быть расположены в порядке возрастания. Функция возвращает номер позиции (число между 0 и n-1), в которой значение x находится в v, и -1, если x не содержится в v.

```
binary(x, v, n) /* find x in 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        /* found match */
      return (mid);
  }
  return (-1);
}
```

Основной частью каждого шага алгоритма является проверка, будет ли x меньше, больше или равен среднему элементу v[mid]; использование конструкции `else-if` здесь вполне естественно.

3.4 Переключатель

Оператор `switch` даёт специальный способ выбора одного из многих вариантов, который заключается в проверке совпадения значения данного выражения с одной из заданных констант и соответствующем ветвлении. В главе 1 мы привели программу подсчёта числа вхождений каждой цифры, символов пустых промежутков и всех остальных символов, использующую последовательность `if...else if...else`. Вот та же самая программа с переключателем.

```
main()
{       /* count digits,white space, others */
  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);
  }
```

Переключатель вычисляет целое выражение в круглых скобках (в данной программе – значение символа c) и сравнивает его значение со всеми случаями (`case`). Каждый случай должен быть помечен либо целым, либо символьной константой, либо константным выражением. Если значение константного выражения, стоящего после вариантного префикса `case`, совпадает со значением целого выражения, то выполнение начинается с этого случая. Если ни один из случаев не подходит, то выполняется оператор после префикса `default`. Префикс `default` является необязательным, если его нет, и ни один из случаев не подходит, то вообще никакие действия не выполняются. Случаи и выбор по умолчанию могут располагаться в любом порядке. Все случаи должны быть различными.

Оператор `break` приводит к немедленному выходу из переключателя. Поскольку случаи служат только в качестве меток, то если вы не предпримите явных действий после выполнения операторов, соответствующих одному случаю, вы провалитесь на следующий случай. Операторы `break` и `return` являются самым обычным способом выхода из переключателя. Как мы обсудим позже в этой главе, оператор `break` можно использовать и для немедленного выхода из операторов цикла `while`, `for` и `do`.

Проваливание сквозь случаи имеет как свои достоинства, так и недостатки. К положительным качествам можно отнести то, что оно позволяет связать несколько случаев с одним действием, как было с пробелом, табуляцией и новой строкой в нашем примере. Но в то же время оно обычно приводит к необходимости заканчивать каждый случай оператором `break`, чтобы избежать перехода к следующему случаю. Проваливание с одного случая на другой обычно бывает неустойчивым, так как оно склонно к расщеплению при модификации программы. За исключением, когда одному вычислению соответствуют несколько меток, проваливание следует использовать умеренно.

Заведите привычку ставить оператор `break` после последнего случая (в данном примере после `default`), даже если это не является логически необходимым. В один пре-

красный день, когда вы добавите в конец ещё один случай, эта маленькая мера предосторожности избавит вас от неприятностей.

Упражнение 3–1

Напишите программу для функции expand(s,t), которая копирует строку s в t, заменяя при этом символы табуляции и новой строки на видимые условные последовательности, как \n и \t. Используйте переключатель.

3.5 Циклы – while и for

Мы уже сталкивались с операторами цикла `while` и `for`. В конструкции

```
while (выражение)
  оператор
```

вычисляется выражение. Если его значение отлично от нуля, то выполняется оператор и выражение вычисляется снова. Этот цикл продолжается до тех пор, пока значение выражения не станет нулём, после чего выполнение программы продолжается с места после оператора.

Оператор

```
for (выражение1; выражение2; выражение3)
  оператор
```

эквивалентен последовательности

```
выражение 1;
while (выражение 2) {
   оператор
   выражение 3;
}
```

Грамматически все три компонента в `for` являются выражениями. Наиболее распространённым является случай, когда выражение 1 и выражение 3 являются присваиваниями или обращениями к функциям, а выражение 2 – условным выражением. Любая из трёх частей может быть опущена, хотя точки с запятой при этом должны оставаться. Если отсутствует выражение 1 или выражение 3, то оно просто выпадает из расширения. Если же отсутствует проверка, выражение 2, то считается, как будто оно всегда истинно, так что

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

является бесконечным циклом, о котором предполагается, что он будет прерван другими средствами (такими как `break` или `return`).

Использовать ли `while` или `for` – это, в основном дело вкуса. Например в

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
  ;    /* skip white space characters */
```

нет ни инициализации, ни реинициализации, так что цикл `while` выглядит самым естественным.

Цикл `for`, очевидно, предпочтительнее там, где имеется простая инициализация и реинициализация, поскольку при этом управляющие циклом операторы наглядным образом оказываются вместе в начале цикла. Это наиболее очевидно в конструкции

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

которая является идиомой языка **C** для обработки первых n элементов массива, аналогичной оператору цикла `do` в фортране и PL/1. Аналогия, однако, не полная, так как границы цикла могут быть изменены внутри цикла, а управляющая переменная сохраняет своё значение после выхода из цикла, какова бы ни была причина этого выхода. Поскольку компонентами `for` могут быть произвольные выражения, они не ограничиваются только арифметическими прогрессиями. Тем не менее является плохим стилем включать в `for` вычисления, которые не относятся к управлению циклом, лучше поместить их в управляемые циклом операторы.

В качестве большего по размеру примера приведём другой вариант функции atoi, преобразующей строку в её численный эквивалент. Этот вариант является более общим; он допускает присутствие в начале символов пустых промежутков и знака + или −.[1]

Общая схема программы отражает форму поступающих данных:

- пропустить пустой промежуток, если он имеется;
- извлечь знак, если он имеется;
- извлечь целую часть и преобразовать её.

Каждый шаг выполняет свою часть работы и оставляет все в подготовленном состоянии для следующей части. Весь процесс заканчивается на первом символе, который не может быть частью числа.

```
atoi(s) /* convert s to integer */
char s[];
{
  int i, n, sign;
  for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++)
    ; /* skip white space */
  sign = 1;
  if (s[i] == '+' || s[i] == '-')       /* sign */
    sign = (s[i++] == '+') ? 1 : -1;
  for (n = 0; s[i] >= '0' && s[i] <= '9'; i++)
    n = 10 * n + s[i] - '0';
  return (sign * n);
}
```

[1] В главе 4 приведена функция atof, которая выполняет то же самое преобразование для чисел с плавающей точкой.

Преимущества централизации управления циклом становятся ещё более очевидными, когда имеется несколько вложенных циклов. Следующая функция сортирует массив целых чисел по методу Шелла. Основная идея сортировки по Шеллу заключается в том, что сначала сравниваются удалённые элементы, а не смежные, как в обычном методе сортировки. Это приводит к быстрому устранению большой части неупорядоченности и сокращает последующую работу. Интервал между элементами постепенно сокращается до единицы, когда сортировка фактически превращается в метод перестановки соседних элементов.

```
shell(v, n)     /* sort v[0]...v[n-1] into increasing order */
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 имеются три вложенных цикла. Самый внешний цикл управляет интервалом между сравниваемыми элементами, уменьшая его от n/2 вдвое при каждом проходе, пока он не станет равным нулю. Средний цикл сравнивает каждую пару элементов, разделённых на величину интервала; самый внутренний цикл переставляет любую неупорядоченную пару. Так как интервал в конце концов сводится к единице, все элементы в результате упорядочиваются правильно. Отметим, что в силу общности конструкции `for` внешний цикл укладывается в ту же самую форму, что и остальные, хотя он и не является арифметической прогрессией.

Последней операцией языка **C** является запятая « , », которая чаще всего используется в операторе `for`. Два выражения, разделённые запятой, вычисляются слева направо, причём типом и значением результата являются тип и значение правого операнда. Таким образом, в различные части оператора `for` можно включить несколько выражений, например, для параллельного изменения двух индексов. Это иллюстрируется функцией reverse(s), которая располагает строку s в обратном порядке на том же месте.

```
reverse(s)      /* reverse string s in place */
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;
  }
}
```

Запятые, которые разделяют аргументы функций, переменные в описаниях и т.д., не имеют отношения к операции запятая и не обеспечивают вычислений слева направо.

Упражнение 3–2

Составьте программу для функции expand(s1,s2), которая расширяет сокращённые обозначения вида a–Z из строки s1 в эквивалентный полный список abc...XYZ в s2. Допускаются сокращения для строчных и прописных букв и цифр. Будьте готовы иметь дело со случаями типа a–b–c, a–Z0–9 и –a–Z. (Полезное соглашение состоит в том, что символ –, стоящий в начале или конце, воспринимается буквально).

3.6 Цикл do – while

Как уже отмечалось в главе 1, циклы `while` и `for` обладают тем приятным свойством, что в них проверка окончания осуществляется в начале, а не в конце цикла. Третий оператор цикла языка **C**, `do-while`, проверяет условие окончания в конце, после каждого прохода через тело цикла; тело цикла всегда выполняется по крайней мере один раз. Синтаксис этого оператора имеет вид:

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

Сначала выполняется оператор, затем вычисляется выражение. Если оно истинно, то оператор выполняется снова и т.д. Если выражение становится ложным, цикл заканчивается.

Как и можно было ожидать, цикл `do-while` используется значительно реже, чем `while` и `for`, составляя примерно пять процентов от всех циклов. Тем не менее, иногда он оказывается полезным, как, например, в следующей функции itoa, которая преобразует число в символьную строку (обратная функции atoi). Эта задача оказывается несколько более сложной, чем может показаться сначала. Дело в том, что простые методы выделения цифр генерируют их в неправильном порядке. Мы предпочли получить строку в обратном порядке, а затем обратить её.

```
itoa(n, s)      /* convert n to characters in s */
char s[];
int n;
{
  int i, sign;
```

```
    if ((sign = n) < 0)   /* record sign */
      n = -n;     /* make n positive */
    i = 0;
    do {  /* generate digits in reverse order */
      s[i++] = n % 10 + '0';      /* get next digit */
    } while ((n /= 10) > 0);      /* delete it */
    if (sign < 0)
      s[i++] = '-';
    s[i] = '\0';
    reverse(s);
  }
```

Цикл `do-while` здесь необходим, или по крайней мере удобен, поскольку, каково бы ни было значение n, массив s должен содержать хотя бы один символ. Мы заключили в фигурные скобки один оператор, составляющий тело `do-while`, хотя это и не обязательно, для того, чтобы торопливый читатель не принял часть `while` за начало оператора цикла `while`.

Упражнение 3–3

При представлении чисел в двоичном дополнительном коде наш вариант itoa не справляется с наибольшим отрицательным числом, т.е. со значением n равным $2^m - 1$, где m размер слова. Объясните почему. Измените программу так, чтобы она правильно печатала это значение на любой машине.

Упражнение 3–4

Напишите аналогичную функцию itob(n,s), которая преобразует целое без знака n в его двоичное символьное представление в s. Запрограммируйте функцию itoh, которая преобразует целое в шестнадцатеричное представление.

Упражнение 3–5

Напишите вариант `itoa`, который имеет три, а не два аргумента. Третий аргумент – минимальная ширина поля; преобразованное число должно, если это необходимо, дополняться слева пробелами, так чтобы оно имело достаточную ширину.

3.7 Оператор break

Иногда бывает удобным иметь возможность управлять выходом из цикла иначе, чем проверкой условия в начале или в конце. Оператор `break` позволяет выйти из операторов `for`, `while` и `do` до окончания цикла точно так же, как и из переключателя. Оператор

break приводит к немедленному выходу из самого внутреннего охватывающего его цикла (или переключателя).

Следующая программа удаляет хвостовые пробелы и табуляции из конца каждой строки файла ввода. Она использует оператор break для выхода из цикла, когда найден крайний правый отличный от пробела и табуляции символ.

```
#define MAXLINE 1000
main()
{       /* remove trailing blanks and tabs */
  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);
  }
}
```

Функция getline возвращает длину строки. Внутренний цикл начинается с последнего символа line (напомним, что --n уменьшает n до использования его значения) и движется в обратном направлении в поиске первого символа, который отличен от пробела, табуляции или новой строки. Цикл прерывается, когда либо найден такой символ, либо n становится отрицательным (т.е., когда просмотрена вся строка). Советуем вам убедиться, что такое поведение правильно и в том случае, когда строка состоит только из символов пустых промежутков.

В качестве альтернативы к break можно ввести проверку в сам цикл:

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

Это уступает предыдущему варианту, так как проверка становится труднее для понимания. Проверок, которые требуют переплетения &&,||,! и круглых скобок, по возможности следует избегать.

3.8 Оператор continue

Оператор continue родственен оператору break, но используется реже; он приводит к началу следующей итерации охватывающего цикла (for, while, do). В циклах while и do это означает непосредственный переход к выполнению проверочной части; в цикле for управление передаётся на шаг реинициализации. (Оператор continue применяется

только в циклах, но не в переключателях. Оператор `continue` внутри переключателя внутри цикла вызывает выполнение следующей итерации цикла).

В качестве примера приведём фрагмент, который обрабатывает только положительные элементы массива a; отрицательные значения пропускаются.

```
for (i = 0; i < n; i++) {
  if (a[i] < 0) /* skip negative elements */
    continue;
  ...   /* do positive elements */
}
```

Оператор `continue` часто используется, когда последующая часть цикла оказывается слишком сложной, так что рассмотрение условия, обратного проверяемому, приводит к слишком глубокому уровню вложенности программы.

Упражнение 3–6

Напишите программу копирования ввода на вывод, с тем исключением, что из каждой группы последовательных одинаковых строк выводится только одна.[1]

3.9 Оператор goto и метки

В языке **C** предусмотрен и оператор `goto`, которым бесконечно злоупотребляют, и метки для ветвления. С формальной точки зрения оператор `goto` никогда не является необходимым, и на практике почти всегда можно обойтись без него. Мы не использовали `goto` в этой книге.

Тем не менее, мы укажем несколько ситуаций, где оператор `goto` может найти своё место. Наиболее характерным является его использование тогда, когда нужно прервать выполнение в некоторой глубоко вложенной структуре, например, выйти сразу из двух циклов. Здесь нельзя непосредственно использовать оператор `break`, так как он прерывает только самый внутренний цикл. Поэтому:

```
for ( ... ) {
  for ( ... ) {
    ...
    if (disaster)
      goto error;
  }
}
...
error:
  clean up the mess
```

[1] Это простой вариант утилиты uniq систем UNIX.

Если программа обработки ошибок нетривиальна и ошибки могут возникать в нескольких местах, то такая организация оказывается удобной. Метка имеет такую же форму, что и имя переменной, и за ней всегда следует двоеточие. Метка может быть приписана к любому оператору той же функции, в которой находится оператор `goto`.

В качестве другого примера рассмотрим задачу нахождения первого отрицательного элемента в двумерном массиве.[1] Вот одна из возможностей:

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    if (v[i][j] < 0)
      goto found;
  /* didn't find */
  ...
found:
  /* found one at position i, j */
...
```

Программа, использующая оператор `goto`, всегда может быть написана без него, хотя, возможно, за счёт повторения некоторых проверок и введения дополнительных переменных. Например, программа поиска в массиве примет вид:

```
found = 0;
for (i = 0; i < n && !found; i++)
  for (j = 0; j < m && !found; j++)
    found = v[i][j] < 0;
if (found)
  /* it was at i-1, j-1 */
  ...
  else
  /* not found */
  ...
```

Хотя мы не являемся в этом вопросе догматиками, нам все же кажется, что если и нужно использовать оператор `goto`, то весьма умеренно.

[1] Многомерные массивы рассматриваются в главе 5.

4 Функции и структура программ

Функции разбивают большие вычислительные задачи на маленькие подзадачи и позволяют использовать в работе то, что уже сделано другими, а не начинать каждый раз с пустого места. Соответствующие функции часто могут скрывать в себе детали проводимых в разных частях программы операций, знать которые нет необходимости, проясняя тем самым всю программу, как целое, и облегчая мучения при внесении изменений.

Язык **C** разрабатывался со стремлением сделать функции эффективными и удобными для использования; **C**-программы обычно состоят из большого числа маленьких функций, а не из нескольких больших. Программа может размещаться в одном или нескольких исходных файлах любым удобным образом; исходные файлы могут компилироваться отдельно и загружаться вместе наряду со скомпилированными ранее функциями из библиотек. Мы здесь не будем вдаваться в детали этого процесса, поскольку они зависят от используемой системы.

Большинство программистов хорошо знакомы с «библиотечными» функциями для ввода и вывода (`getchar`, `putchar`) и для численных расчётов (`sin`, `cos`, `sqrt`). В этой главе мы сообщим больше о написании новых функций.

4.1 Основные сведения

Для начала давайте разработаем и составим программу печати каждой строки ввода, которая содержит определённую комбинацию символов.[1] Например, при поиске комбинации «the» в наборе строк

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

в качестве выхода получим

```
now is the time
men to come to the aid
of their party
```

основная схема выполнения задания чётко разделяется на три части:

```
while (имеется ещё строка)
if (строка содержит нужную комбинацию)
  вывод этой строки
```

[1] Это – специальный случай утилиты grep системы UNIX.

Конечно, возможно запрограммировать все действия в виде одной основной процедуры, но лучше использовать естественную структуру задачи и представить каждую часть в виде отдельной функции. С тремя маленькими кусками легче иметь дело, чем с одним большим, потому что отдельные не относящиеся к существу дела детали можно включить в функции и уменьшить возможность нежелательных взаимодействий. Кроме того, эти куски могут оказаться полезными сами по себе.

«Пока имеется ещё строка» – это `getline`, функция, которую мы запрограммировали в главе 1, а «вывод этой строки» это функция `printf`, которую уже кто-то подготовил для нас. Это значит, что нам осталось только написать процедуру для определения, содержит ли строка данную комбинацию символов или нет. Мы можем решить эту проблему, позаимствовав разработку из PL/1: функция `index(s,t)` возвращает позицию, или индекс, строки s, где начинается строка t, и -1, если s не содержит t. В качестве начальной позиции мы используем 0, а не 1, потому что в языке **C** массивы начинаются с позиции нуль. Когда нам в дальнейшем понадобится проверять на совпадение более сложные конструкции, нам придётся заменить только функцию `index`; остальная часть программы останется той же самой.

После того, как мы потратили столько усилий на разработку, написание программы в деталях не представляет затруднений. Ниже приводится целиком вся программа, так что вы можете видеть, как соединяются вместе отдельные части. Комбинация символов, по которой производится поиск, выступает пока в качестве символьной строки в аргументе функции `index`, что не является самым общим механизмом. Мы скоро вернёмся к обсуждению вопроса об инициализации символьных массивов и в главе 5 покажем, как сделать комбинацию символов параметром, которому присваивается значение в ходе выполнения программы. Программа также содержит новый вариант функции `getline`; вам может оказаться полезным сравнить его с вариантом из главы 1.

```
#define  MAXLINE  1000

main()
{       /* find all lines matching a pattern */
  char line[MAXLINE];

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

getline(s, lim) /* get line into s, return length */
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)     /* return index of t in s,-1 if none */
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 вид:

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

Как и указывается, некоторые части могут отсутствовать; минимальной функцией является

```
dummy(){}
```

которая не совершает никаких действий.[1] Если функция возвращает что-либо отличное от целого значения, то перед её именем может стоять указатель типа; этот вопрос обсуждается в следующем разделе.

Программой является просто набор определений отдельных функций. Связь между функциями осуществляется через аргументы и возвращаемые функциями значения; её можно также осуществлять через внешние переменные. Функции могут располагаться в исходном файле в любом порядке, а сама исходная программа может размещаться на нескольких файлах, но так, чтобы ни одна функция не расщеплялась.

Оператор `return` служит механизмом для возвращения значения из вызванной функции в функцию, которая к ней обратилась. За `return` может следовать любое выражение:

```
return (выражение)
```

[1] Такая ничего не делающая функция иногда оказывается удобной для сохранения места для дальнейшего развития программы.

Вызывающая функция может игнорировать возвращаемое значение, если она этого пожелает. Более того, после `return` может не быть вообще никакого выражения; в этом случае в вызывающую программу не передаётся никакого значения. Управление также возвращается в вызывающую программу без передачи какого-либо значения и в том случае, когда при выполнении мы «проваливаемся» на конец функции, достигая закрывающейся правой фигурной скобки. Если функция возвращает значение из одного места и не возвращает никакого значения из другого места, это не является незаконным, но может быть признаком каких-то неприятностей. В любом случае «значением» функции, которая не возвращает значения, несомненно будет мусор. Отладочная программа `lint` проверяет такие ошибки.

Механика компиляции и загрузки **C**-программ, расположенных в нескольких исходных файлах, меняется от системы к системе. В системе UNIX, например, эту работу выполняет команда cc, упомянутая в главе 1. Предположим, что три функции находятся в трёх различных файлах с именами main.c, getline.c и index.c. Тогда команда

```
cc main.c getline.c index.c
```

компилирует эти три файла, помещает полученный настраиваемый объектный код в файлы main.o, getline.o и index.o и загружает их всех в выполняемый файл, называемый a.out.

Если имеется какая-то ошибка, скажем в main.c, то этот файл можно перекомпилировать отдельно и загрузить вместе с предыдущими объектными файлами по команде

```
cc main.c getlin.o index.o
```

Команда cc использует соглашение о наименовании с «.c» и «.o» для того, чтобы отличить исходные файлы от объектных.

Упражнение 4–1

Составьте программу для функции `rindex(s,t)`, которая возвращает позицию самого правого вхождения t в s и -1, если s не содержит t.

4.2 Функции, возвращающие нецелые значения

До сих пор ни одна из наших программ не содержала какого-либо описания типа функции. Дело в том, что по умолчанию функция неявно описывается своим появлением в выражении или операторе, как, например, в

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

Если некоторое имя, которое не было описано ранее, появляется в выражении и за ним следует левая круглая скобка, то оно по контексту считается именем некоторой функции. Кроме того, по умолчанию предполагается, что эта функция возвращает значение типа `int`. Так как в выражениях `char` преобразуется в `int`, то нет необходимости описывать функции, возвращающие `char`. Эти предположения покрывают большинство случаев, включая все приведённые до сих пор примеры.

Но что происходит, если функция должна возвратить значение какого-то другого типа? Многие численные функции, такие как `sqrt`, `sin` и `cos` возвращают `double`; другие специальные функции возвращают значения других типов. Чтобы показать, как поступать в этом случае, давайте напишем и используем функцию `atof(s)`, которая преобразует строку s в эквивалентное ей плавающее число двойной точности. Функция `atof` является расширением `atoi`, варианты которой мы написали в главах 2 и 3; она обрабатывает необязательно знак и десятичную точку, а также целую и дробную часть, каждая из которых может как присутствовать, так и отсутствовать.[1]

Во-первых, сама `atof` должна описывать тип возвращаемого ею значения, поскольку он отличен от `int`. Так как в выражениях тип `float` преобразуется в `double`, то нет никакого смысла в том, чтобы `atof` возвращала `float`; мы можем с равным успехом воспользоваться дополнительной точностью, так что мы полагаем, что возвращаемое значение типа `double`. Имя типа должно стоять перед именем функции, как показывается ниже:

```
double atof(s)  /* convert string s to double */
char s[];
{
  double val, power;
  int i, sign;

  for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++);
    ;       /* skip white space */
  sign = 1;
  if (s[i] == '+' || s[i] == '-')       /* sign */
    sign = (s[i++] == '+') ? 1 : -1;
  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);
}
```

Вторым, но столь же важным, является то, что вызывающая функция должна объявить о том, что `atof` возвращает значение, отличное от `int` типа. Такое объявление демонстрируется на примере следующего примитивного настольного калькулятора (едва пригодного для подведения баланса в чековой книжке), который считывает по одному числу на строку, причём это число может иметь знак, и складывает все числа, печатая сумму после каждого ввода.

[1] Эта процедура преобразования ввода не очень высокого качества; иначе она бы заняла больше места, чем нам хотелось бы.

```
#define   MAXLINE   100
main()
{       /* rudimentary desk calkulator */
  double sum, atof();
  char line[MAXLINE];

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

Описание

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

говорит, что sum является переменной типа `double`, и что `atof` является функцией, возвращающей значение типа `double`. Эта мнемоника означает, что значениями как sum, так и `atof()` являются плавающие числа двойной точности.

Если функция `atof` не будет описана явно в обоих местах, то в **C** предполагается, что она возвращает целое значение, и вы получите бессмысленный ответ. Если сама `atof` и обращение к ней в `main` имеют несовместимые типы и находятся в одном и том же файле, то это будет обнаружено компилятором. Но если `atof` была скомпилирована отдельно (что более вероятно), то это несоответствие не будет зафиксировано, так что `atof` будет возвращать значения типа `double`, с которым `main` будет обращаться, как с `int`, что приведёт к бессмысленным результатам.[1]

Имея `atof`, мы, в принципе, могли бы с её помощью написать `atoi` (преобразование строки в `int`):

```
atoi(s) /* convert string s to integer */
char s[];
{
  double atof();

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

Обратите внимание на структуру описаний и оператор `return`. Значение выражения в

```
return (выражение)
```

всегда преобразуется к типу функции перед выполнением самого возвращения. Поэтому при появлении в операторе `return` значение функции `atof`, имеющее тип `double`, автоматически преобразуется в `int`, поскольку функция `atoi` возвращает `int`.[2]

[1] Программа `lint` вылавливает эту ошибку.

[2] Как обсуждалось в главе 2, преобразование значения с плавающей точкой к типу `int` осуществляется посредством отбрасывания дробной части.

Упражнение 4–2

Расширьте `atof` таким образом, чтобы она могла работать с числами вида `123.45e-6` где за числом с плавающей точкой может следовать e и показатель экспоненты, возможно со знаком.

4.3 Ещё об аргументах функций

В главе 1 мы уже обсуждали тот факт, что аргументы функций передаются по значению, т.е. вызванная функция получает свою временную копию каждого аргумента, а не его адрес. Это означает, что вызванная функция не может воздействовать на исходный аргумент в вызывающей функции. Внутри функции каждый аргумент по существу является локальной переменной, которая инициализируется тем значением, с которым к этой функции обратились.

Если в качестве аргумента функции выступает имя массива, то передаётся адрес начала этого массива; сами элементы не копируются. Функция может изменять элементы массива, используя индексацию и адрес начала. Таким образом, массив передаётся по ссылке. В главе 5 мы обсудим, как использование указателей позволяет функциям воздействовать на отличные от массивов переменные в вызывающих функциях.

Между прочим, не существует полностью удовлетворительного способа написания переносимой функции с переменным числом аргументов. Дело в том, что нет переносимого способа, с помощью которого вызванная функция могла бы определить, сколько аргументов было фактически передано ей в данном обращении. Таким образом, вы, например, не можете написать действительно переносимую функцию, которая будет вычислять максимум от произвольного числа аргументов, как делают встроенные функции MAX в фортране и PL/1.

Обычно со случаем переменного числа аргументов безопасно иметь дело, если вызванная функция не использует аргументов, которые ей на самом деле не были переданы, и если типы согласуются. Самая распространённая в языке **C** функция с переменным числом – `printf`. Она получает из первого аргумента информацию, позволяющую определить количество остальных аргументов и их типы. Функция `printf` работает совершенно неправильно, если вызывающая функция передаёт ей недостаточное количество аргументов, или если их типы не согласуются с типами, указанными в первом аргументе. Эта функция не является переносимой и должна модифицироваться при использовании в различных условиях.

Если же типы аргументов известны, то конец списка аргументов можно отметить, используя какое-то соглашение; например, считая, что некоторое специальное значение аргумента (часто нуль) является признаком конца аргументов.

4.4 Внешние переменные

Программа на языке **C** состоит из набора внешних объектов, которые являются либо переменными, либо функциями. Термин «внешний» используется главным образом в

противопоставление термину «внутренний», которым описываются аргументы и автоматические переменные, определённые внутри функций. Внешние переменные определены вне какой-либо функции и, таким образом, потенциально доступны для многих функций. Сами функции всегда являются внешними, потому что правила языка **C** не разрешают определять одни функции внутри других. По умолчанию внешние переменные являются также и «глобальными», так что все ссылки на такую переменную, использующие одно и то же имя (даже из функций, скомпилированных независимо), будут ссылками на одно и то же. В этом смысле внешние переменные аналогичны переменным COMMON в фортране и EXTERNAL в PL/1. Позднее мы покажем, как определить внешние переменные и функции таким образом, чтобы они были доступны не глобально, а только в пределах одного исходного файла.

В силу своей глобальной доступности внешние переменные предоставляют другую, отличную от аргументов и возвращаемых значений, возможность для обмена данными между функциями. Если имя внешней переменной каким-либо образом описано, то любая функция имеет доступ к этой переменной, ссылаясь к ней по этому имени.

В случаях, когда связь между функциями осуществляется с помощью большого числа переменных, внешние переменные оказываются более удобными и эффективными, чем использование длинных списков аргументов. Как, однако, отмечалось в главе 1, это соображение следует использовать с определённой осторожностью, так как оно может плохо отразиться на структуре программ и приводить к программам с большим числом связей по данным между функциями.

Вторая причина использования внешних переменных связана с инициализацией. В частности, внешние массивы могут быть инициализированы а автоматические нет. Мы рассмотрим вопрос об инициализации в конце этой главы.

Третья причина использования внешних переменных обусловлена их областью действия и временем существования. Автоматические переменные являются внутренними по отношению к функциям; они возникают при входе в функцию и исчезают при выходе из неё. Внешние переменные, напротив, существуют постоянно. Они не появляются и не исчезают, так что могут сохранять свои значения в период от одного обращения к функции до другого. В силу этого, если две функции используют некоторые общие данные, причём ни одна из них не обращается к другой, то часто наиболее удобным оказывается хранить эти общие данные в виде внешних переменных, а не передавать их в функцию и обратно с помощью аргументов.

Давайте продолжим обсуждение этого вопроса на большом примере. Задача будет состоять в написании другой программы для калькулятора, лучшей, чем предыдущая. Здесь допускаются операции `+`, `-`, `*`, `/` и знак `=` (для выдачи ответа). Вместо инфиксного представления калькулятор будет использовать обратную польскую нотацию, поскольку её несколько легче реализовать. В обратной польской нотации знак следует за операндами; инфиксное выражение типа

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

записывается в виде

```
12-45+*=
```

круглые скобки при этом не нужны

Реализация оказывается весьма простой, каждый операнд помещается в стек; когда поступает знак операции, нужное число операндов (два для бинарных операций) вынимается, к ним применяется операция и результат направляется обратно в стек. Так в приведённом выше примере 1 и 2 помещаются в стек и затем заменяются их разностью, −1. После этого 4 и 5 вводятся в стек и затем заменяются своей суммой, 9. Далее числа −1 и 9 заменяются в стеке на их произведение, равное −9. Операция = печатает верхний элемент стека, не удаляя его (так что промежуточные вычисления могут быть проверены).

Сами операции помещения чисел в стек и их извлечения очень просты, но, в связи с включением в настоящую программу обнаружения ошибок и восстановления, они оказываются достаточно длинными. Поэтому лучше оформить их в виде отдельных функций, чем повторять соответствующий текст повсюду в программе. Кроме того, нужна отдельная функция для выборки из ввода следующей операции или операнда. Таким образом, структура программы имеет вид:

```
while( поступает операция или операнд, а не конец )
  if ( число )
    поместить его в стек
  else if ( операция )
    вынуть операнды из стека
    выполнить операцию
    поместить результат в стек
  else
    ошибка
```

Основной вопрос, который ещё не был обсуждён, заключается в том, где поместить стек, т.е. какие процедуры смогут обращаться к нему непосредственно. Одна из таких возможностей состоит в помещении стека в `main` и передачи самого стека и текущей позиции в стеке функциям, работающим со стеком. Но функции `main` нет необходимости иметь дело с переменными, управляющими стеком; ей естественно рассуждать в терминах помещения чисел в стек и извлечения их оттуда. В силу этого мы решили сделать стек и связанную с ним информацию внешними переменными, доступными функциям `push` (помещение в стек) и `pop` (извлечение из стека), но не `main`.

Перевод этой схемы в программу достаточно прост. Ведущая программа является по существу большим переключателем по типу операции или операнду; это, по-видимому, более характерное применение переключателя, чем то, которое было продемонстрировано в главе 3.

```
#define MAXOP   20      /* max size of operand, operator */
#define NUMBER '0'      /* signal that number found */
#define TOOBIG '9'      /* signal that string is too big */

main()
{       /* reverse polish desk calculator */
  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;
  }
}

#define MAXVAL 100      /* maximum depth of val stack */

int sp = 0;     /* stack pointer */
double val[MAXVAL];     /* value stack */

double push(f)  /* push f onto value stack */
double f;
{
  if (sp < MAXVAL)
```

```
    return (val[sp++] = f);
  else {
    printf("error: stack full\n");
    clear();
    return (0);
  }
}

double pop()
{       /* pop top value from steack */
  if (sp > 0)
    return (val[--sp]);
  else {
    printf("error: stack empty\n");
    clear();
    return (0);
  }
}

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

Команда «c» очищает стек с помощью функции `clear`, которая также используется в случае ошибки функциями `push` и `pop`. К функции `getop` мы очень скоро вернёмся.

Как уже говорилось в главе 1, переменная является внешней, если она определена вне тела какой бы то ни было функции. Поэтому стек и указатель стека, которые должны использоваться функциями `push`, `pop` и `clear`, определены вне этих трёх функций. Но сама функция `main` не ссылается ни к стеку, ни к указателю стека – их участие тщательно замаскировано. В силу этого часть программы, соответствующая операции =, использует конструкцию

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

для того, чтобы проанализировать верхний элемент стека, не изменяя его.

Отметим также, что так как операции + и $*$ коммутативны, порядок, в котором объединяются извлечённые операнды, несущественен, но в случае операций $-$ и / необходимо различать левый и правый операнды.

Упражнение 4–3

Приведённая основная схема допускает непосредственное расширение возможностей калькулятора. Включите операцию деления по модулю (%) и унарный минус. Включите команду «стереть», которая удаляет верхний элемент стека. Введите команды для работы с переменными. (Это просто, если имена переменных будут состоять из одной буквы из имеющихся двадцати шести букв.)

4.5 Правила, определяющие область действия

Функции и внешние переменные, входящие в состав **C**-программы, не обязаны компилироваться одновременно; программа на исходном языке может располагаться в нескольких файлах, и ранее скомпилированные процедуры могут загружаться из библиотек. Два вопроса представляют интерес:

- Как следует составлять описания, чтобы переменные правильно воспринимались во время компиляции?
- Как следует составлять описания, чтобы обеспечить правильную связь частей программы при загрузке?

4.5.1 Область действия

Областью действия имени является та часть программы, в которой это имя определено. Для автоматической переменной, описанной в начале функции, областью действия является та функция, в которой описано имя этой переменной, а переменные из разных функций, имеющие одинаковое имя, считаются не относящимися друг к другу. Это же справедливо и для аргументов функций.

Область действия внешней переменной простирается от точки, в которой она объявлена в исходном файле, до конца этого файла. Например, если `val`, `sp`, `push`, `pop` и `clear` определены в одном файле в порядке, указанном выше, а именно:

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

double push(f) {...}

double pop() {...}

clear() {...}
```

то переменные `val` и `sp` можно использовать в `push`, `pop` и `clear` прямо по имени; никакие дополнительные описания не нужны.

С другой стороны, если нужно сослаться на внешнюю переменную до её определения, или если такая переменная определена в файле, отличном от того, в котором она используется, то необходимо описание `extern`.

Важно различать описание внешней переменной и её определение. Описание указывает свойства переменной (её тип, размер и т.д.); определение же вызывает ещё и отведение памяти. Если вне какой бы то ни было функции появляются строчки

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

то они определяют внешние переменные `sp` и `val`, вызывают отведение памяти для них и служат в качестве описания для остальной части этого исходного файла. В то же время строчки

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

описывают в остальной части этого исходного файла переменную `sp` как `int`, а `val` как массив типа `double` (размер которого указан в другом месте), но не создают переменных и не отводят им места в памяти.

Во всех файлах, составляющих исходную программу, должно содержаться только одно определение внешней переменной; другие файлы могут содержать описания `extern` для доступа к ней.[1] Любая инициализация внешней переменной проводится только в определении. В определении должны указываться размеры массивов, а в описании `extern` этого можно не делать.

Хотя подобная организация приведённой выше программы и маловероятна, но `val` и `sp` могли бы быть определены и инициализированы в одном файле, а функция `push`, `pop` и `clear` определены в другом. В этом случае для связи были бы необходимы следующие определения и описания:

в файле 1:

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

double push(f) {...}

double pop() {...}

clear() {...}
```

в файле 2:

```
int sp = 0;             /* stack pointer */
double val[MAXVAL];     /* value stack */
```

так как описания `extern` *в файле 1* находятся выше и вне трёх указанных функций, они относятся ко всем ним; одного набора описаний достаточно для всего *файла 1*.

Для программ большого размера обсуждаемая позже в этой главе возможность включения файлов, `#include`, позволяет иметь во всей программе только одну копию описаний `extern` и вставлять её в каждый исходный файл во время его компиляции.

Обратимся теперь к функции `getop`, выбирающей из файла ввода следующую операцию или операнд. Основная задача проста: пропустить пробелы, знаки табуляции и новые строки. Если следующий символ отличен от цифры и десятичной точки, то возвратить его. В противном случае собрать строку цифр (она может включать десятичную точку) и возвратить `NUMBER` как сигнал о том, что выбрано число.

[1] Описание `extern` может иметься и в том файле, где находится определение.

Процедура существенно усложняется, если стремиться правильно обрабатывать ситуацию, когда вводимое число оказывается слишком длинным. Функция getop считывает цифры подряд (возможно с десятичной точкой) и запоминает их, пока последовательность не прерывается. Если при этом не происходит переполнения, то функция возвращает NUMBER и строку цифр. Если же число оказывается слишком длинным, то getop отбрасывает остальную часть строки из файла ввода, так что пользователь может просто перепечатать эту строку с места ошибки; функция возвращает TOOBIG как сигнал о переполнении.

```
getop(s, lim)   /* get next oprerator or operand */
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 == '.') {       /* collect fraction */
    if (i < lim)
      s[i] = c;
    for (i++; (c = getchar()) >= '0' && c <= '9'; i++)
      if (i < lim)
        s[i] = c;
  }
  if (i < lim) {        /* number is ok */
    ungetch(c);
    s[i] = '\0';
    return (NUMBER);
  } else {      /* it's too big; skip rest of line */
    while (c != '\n' && c != EOF)
      c = getchar();
    s[lim - 1] = '\0';
    return (TOOBIG);
  }
}
```

Что же представляют из себя функции getch и ungetch? Часто так бывает, что программа, считывающая входные данные, не может определить, что она прочла уже достаточно, пока она не прочтёт слишком много. Одним из примеров является выбор символов, составляющих число: пока не появится символ, отличный от цифры, число не закончено. Но при этом программа считывает один лишний символ, символ, для которого она ещё не подготовлена.

Эта проблема была бы решена, если бы было бы возможно «прочесть обратно» нежелательный символ. Тогда каждый раз, прочитав лишний символ, программа могла бы поместить его обратно в файл ввода таким образом, что остальная часть программы могла бы вести себя так, словно этот символ никогда не считывался. К счастью, такое неполучение символа легко имитировать, написав пару действующих совместно функций. Функция `getch` доставляет следующий символ ввода, подлежащий рассмотрению; функция `ungetch` помещает символ назад во ввод, так что при следующем обращении к `getch` он будет возвращён.

То, как эти функции совместно работают, весьма просто. Функция `ungetch` помещает возвращаемые назад символы в совместно используемый буфер, являющийся символьным массивом. Функция `getch` читает из этого буфера, если в нем что-либо имеется; если же буфер пуст, она обращается к `getchar`. При этом также нужна индексирующая переменная, которая будет фиксировать позицию текущего символа в буфере.

Так как буфер и его индекс совместно используются функциями `getch` и `ungetch` и должны сохранять свои значения в период между обращениями, они должны быть внешними для обеих функций. Таким образом, мы можем написать `getch`, `ungetch` и эти переменные как:

```
#define  BUFSIZE  100
char buf[BUFSIZE];      /* buffer for ungetch */
int bufp = 0;   /* next free position in buf */

getch()
{       /* get a (possibly pushed back) character */
  return ((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch(c)      /* push character back on input */
int c;
{
  if (bufp > BUFSIZE)
    printf("ungetch: too many characters\n");
  else
    buf[bufp++] = c;
}
```

Мы использовали для хранения возвращаемых символов массив, а не отдельный символ, потому что такая общность может пригодиться в дальнейшем.

Упражнение 4–4

Напишите функцию `ungets(s)`, которая будет возвращать во ввод целую строку. Должна ли `ungets` иметь дело с `buf` и `bufp` или она может просто использовать `ungetch`?

Упражнение 4–5

Предположите, что может возвращаться только один символ. Измените `getch` и `ungetch` соответствующим образом.

Упражнение 4–6

Наши функции `getch` и `ungetch` не обеспечивают обработку возвращённого символа `EOF` переносимым образом. Решите, каким свойством должны обладать эти функции, если возвращается `EOF`, и реализуйте ваши выводы.

4.6 Статические переменные

Статические переменные представляют собой третий класс памяти, в дополнении к автоматическим переменным и `extern`, с которыми мы уже встречались.

Статические переменные могут быть либо внутренними, либо внешними. Внутренние статические переменные точно так же, как и автоматические, являются локальными для некоторой функции, но, в отличие от автоматических, они остаются существовать, а не появляются и исчезают вместе с обращением к этой функции. Это означает, что внутренние статические переменные обеспечивают постоянное, недоступное извне хранение внутри функции. Символьные строки, появляющиеся внутри функции, как, например, аргументы `printf`, являются внутренними статическими.

Внешние статические переменные определены в остальной части того исходного файла, в котором они описаны, но не в каком-либо другом файле. Таким образом, они дают способ скрывать имена, подобные buf и bufp в комбинации `getch-ungetch`, которые в силу их совместного использования должны быть внешними, но все же не доступными для пользователей `getch` и `ungetch`, чтобы исключалась возможность конфликта. Если эти две функции и две переменные объединить в одном файле следующим образом

```
static char buf[BUFSIZE];       /* buffer for ungetch */
static int bufp = 0;    /* next free position in buf */

getch() {...}

ungetch() {...}
```

то никакая другая функция не будет в состоянии обратиться к buf и bufp; фактически, они не будут вступать в конфликт с такими же именами из других файлов той же самой программы.

Статическая память, как внутренняя, так и внешняя, специфицируется словом `static`, стоящим перед обычным описанием. Переменная является внешней, если она описана вне какой бы то ни было функции, и внутренней, если она описана внутри некоторой функции.

Нормально функции являются внешними объектами; их имена известны глобально. Возможно, однако, объявить функцию как `static`; тогда её имя становится неизвестным вне файла, в котором оно описано.

В языке **C** `static` отражает не только постоянство, но и степень того, что можно назвать «приватностью». Внутренние статические объекты определены только внутри одной функции; внешние статические объекты (переменные или функции) определены только внутри того исходного файла, где они появляются, и их имена не вступают в конфликт с такими же именами переменных и функций из других файлов.

Внешние статические переменные и функции предоставляют способ организовывать данные и работающие с ними внутренние процедуры таким образом, что другие процедуры и данные не могут прийти с ними в конфликт даже по недоразумению. Например, функции `getch` и `ungetch` образуют «модуль» для ввода и возвращения символов; buf и bufp должны быть статическими, чтобы они не были доступны извне. Точно так же функции `push`, `pop` и `clear` формируют модуль обработки стека; `val` и `sp` тоже должны быть внешними статическими.

4.7 Регистровые переменные

Четвёртый и последний класс памяти называется регистровым. Описание `register` указывает компилятору, что данная переменная будет часто использоваться. Когда это возможно, переменные, описанные как `register`, располагаются в машинных регистрах, что может привести к меньшим по размеру и более быстрым программам. Описание `register` выглядит как

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

и т.д.; часть `int` может быть опущена. Описание `register` можно использовать только для автоматических переменных и формальных параметров функций. В этом последнем случае описания выглядят следующим образом:

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

На практике возникают некоторые ограничения на регистровые переменные, отражающие реальные возможности имеющихся аппаратных средств. В регистры можно поместить только несколько переменных в каждой функции, причём только определённых типов. В случае превышения возможного числа или использования неразрешённых типов слово `register` игнорируется. Кроме того невозможно извлечь адрес регистровой переменной (этот вопрос обсуждается в главе 5). Эти специфические ограничения варьируются от машины к машине. Так, например, на PDP-11 эффективными являются только

первые три описания `register` в функции, а в качестве типов допускаются `int`, `char` или указатель.

4.8 Блочная структура

Язык **C** не является языком с блочной структурой в смысле PL/1 или алгола; в нем нельзя описывать одни функции внутри других.

Переменные же, с другой стороны, могут определяться по методу блочного структурирования. Описания переменных (включая инициализацию) могут следовать за левой фигурной скобкой, открывающей любой оператор, а не только за той, с которой начинается тело функции. Переменные, описанные таким образом, вытесняют любые переменные из внешних блоков, имеющие такие же имена, и остаются определёнными до соответствующей правой фигурной скобки. Например в

```
if (n > 0) {
  int i;        /* declare a new i */
  for (i = 0; i < n; i++)
    ...
}
```

Областью действия переменной i является «истинная» ветвь `if`; это i никак не связано ни с какими другими i в программе.

Блочная структура влияет и на область действия внешних переменных. Если даны описания

```
int x;

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

то появление x внутри функции f относится к внутренней переменной типа `double`, а вне f – к внешней целой переменной. Это же справедливо в отношении имён формальных параметров:

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

Внутри функции f имя x относится к формальному параметру, а не к внешней переменной.

4.9 Инициализация

Мы до сих пор уже много раз упоминали инициализацию, но всегда мимоходом, среди других вопросов. Теперь, после того как мы обсудили различные классы памяти, мы в этом разделе просуммируем некоторые правила, относящиеся к инициализации.

Если явная инициализация отсутствует, то внешним и статическим переменным присваивается значение нуль; автоматические и регистровые переменные имеют в этом случае неопределённые значения (мусор).

Простые переменные (не массивы или структуры) можно инициализировать при их описании, добавляя вслед за именем знак равенства и константное выражение:

```
int x = 1;
char squote = '\'';
long day = 60 * 24;     /* minutes in a day */
```

Для внешних и статических переменных инициализация выполняется только один раз, на этапе компиляции. Автоматические и регистровые переменные инициализируются каждый раз при входе в функцию или блок. В случае автоматических и регистровых переменных инициализатор не обязан быть константой: на самом деле он может быть любым значимым выражением, которое может включать определённые ранее величины и даже обращения к функциям. Например, инициализация в программе бинарного поиска из главы 3 могла бы быть записана в виде

```
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;
  ...
}
```

По своему результату, инициализации автоматических переменных являются сокращённой записью операторов присваивания. Какую форму предпочесть – в основном дело вкуса. Мы обычно используем явные присваивания, потому что инициализация в описаниях менее заметна.

Автоматические массивы не могут быть инициализированы. Внешние и статические массивы можно инициализировать, помещая вслед за описанием заключённый в фигурные скобки список начальных значений, разделённых запятыми. Например программа подсчёта символов из главы 1, которая начиналась с

```
main()
{       /* count digits, white space, others */
  int c, i, nwhite, nother;
  int ndigit[10];

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

Может быть переписана в виде

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

main()
{       /* count digits, white space, others */
  int c, i;
  ...
}
```

Эти инициализации фактически не нужны, так как все присваиваемые значения равны нулю, но хороший стиль – сделать их явными. Если количество начальных значений меньше, чем указанный размер массива, то остальные элементы заполняются нулями. Перечисление слишком большого числа начальных значений является ошибкой. К сожалению, не предусмотрена возможность указания, что некоторое начальное значение повторяется, и нельзя инициализировать элемент в середине массива без перечисления всех предыдущих.

Для символьных массивов существует специальный способ инициализации; вместо фигурных скобок и запятых можно использовать строку:

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

Это сокращение более длинной, но эквивалентной записи:

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

Если размер массива любого типа опущен, то компилятор определяет его длину, подсчитывая число начальных значений. В этом конкретном случае размер равен четырём (три символа плюс конечное \0).

4.10 Рекурсия

В языке **C** функции могут использоваться рекурсивно; это означает, что функция может прямо или косвенно обращаться к себе самой. Традиционным примером является печать числа в виде строки символов. Как мы уже ранее отмечали, цифры генерируются не в том порядке: цифры младших разрядов появляются раньше цифр из старших разрядов, но печататься они должны в обратном порядке.

Эту проблему можно решить двумя способами. Первый способ, которым мы воспользовались в главе 3 в функции `itoa`, заключается в запоминании цифр в некотором массиве по мере их поступления и последующем их печатании в обратном порядке. Первый вариант функции `printd` следует этой схеме.

```
printd(n)       /* print n in decimal */
int n;
{
  char s[10];
  int i;

  if (n < 0) {
    putchar('-');
    n = -n;
  }
  i = 0;
  do {
    s[i++] = n % 10 + '0';      /* get next char */
  } while ((n /= 10) > 0);      /* discard it */
  while (--i >= 0)
    putchar(s[i]);
}
```

Альтернативой этому способу является рекурсивное решение, когда при каждом вызове функция `printd` сначала снова обращается к себе, чтобы скопировать лидирующие цифры, а затем печатает последнюю цифру.

```
printd(n)       /* print n in decimal (recursive) */
int n;
{
  int i;

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

Когда функция вызывает себя рекурсивно, при каждом обращении образуется новый набор всех автоматических переменных, совершенно не зависящий от предыдущего набора. Таким образом, в `printd(123)` первая функция `printd` имеет n=123. Она передаёт 12 второй `printd`, а когда та возвращает управление ей, печатает 3. Точно так же вторая `printd` передаёт 1 третьей (которая эту единицу печатает), а затем печатает 2.

Рекурсия обычно не даёт никакой экономии памяти, поскольку приходится где-то создавать стек для обрабатываемых значений. Не приводит она и к созданию более быстрых программ. Но рекурсивные программы более компактны, и они зачастую становятся более лёгкими для понимания и написания. Рекурсия особенно удобна при работе с рекурсивно определяемыми структурами данных, например, с деревьями; хороший пример будет приведён в главе 6.

Упражнение 4–7

Приспособьте идеи, использованные в `printd` для рекурсивного написания `itoa`; т.е. преобразуйте целое в строку с помощью рекурсивной процедуры.

Упражнение 4–8

Напишите рекурсивный вариант функции `reverse(s)`, которая располагает в обратном порядке строку s.

4.11 Препроцессор языка C

В языке **C** предусмотрены определённые расширения языка с помощью простого макропрепроцессора. Одним из самых распространённых таких расширений, которое мы уже использовали, является конструкция `#define`; другим расширением является возможность включать во время компиляции содержимое других файлов.

4.11.1 Включение файлов

Для облегчения работы с наборами конструкций `#define` и описаний (среди прочих средств) в языке **C** предусмотрена возможность включения файлов. Любая строка вида

```
#include "filename"
```

заменяется содержимым файла с именем filename.[1] Часто одна или две строки такого вида появляются в начале каждого исходного файла, для того чтобы включить общие конструкции `#define` и описания `extern` для глобальных переменных. Допускается вложенность конструкций `#include`.

Конструкция `#include` является предпочтительным способом связи описаний в больших программах. Этот способ гарантирует, что все исходные файлы будут снабжены одинаковыми определениями и описаниями переменных, и, следовательно, исключает особенно неприятный сорт ошибок. Естественно, когда какой-то включаемый файл изменяется, все зависящие от него файлы должны быть перекомпилированы.

[1] Кавычки обязательны.

4.11.2 Макроподстановка

Определение вида

```
#define TES     1
```

приводит к макроподстановке самого простого вида – замене имени на строку символов. Имена в `#define` имеют ту же самую форму, что и идентификаторы в **C**; заменяющий текст совершенно произволен. Нормально заменяющим текстом является остальная часть строки; длинное определение можно продолжить, поместив \ в конец продолжаемой строки. «Область действия» имени, определённого в `#define`, простирается от точки определения до конца исходного файла. Имена могут быть переопределены, и определения могут использовать определения, сделанные ранее. Внутри заключённых в кавычки строк подстановки не производятся, так что если, например, `YES` – определённое имя, то в `printf("YES")` не будет сделано никакой подстановки.

Так как реализация `#define` является частью работы макропрепроцессора, а не собственно компилятора, имеется очень мало грамматических ограничений на то, что может быть определено. Так, например, любители алгола могут объявить

```
#define THEN
#define BEGIN {
#define END   };
```

и затем написать

```
if (i > 0) THEN
   BEGIN
           a = 1;
           b = 2;
   END
```

Имеется также возможность определения макроса с аргументами, так что заменяющий текст будет зависеть от вида обращения к макросу. Определим, например, макрос с именем `MAX` следующим образом:

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

когда строка

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

будет заменена строкой

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

Такая возможность обеспечивает «функцию максимума», которая расширяется в последовательный код, а не в обращение к функции. При правильном обращении с аргументами такой макрос будет работать с любыми типами данных; здесь нет необходимости в различных видах `MAX` для данных разных типов, как это было бы с функциями.

Конечно, если вы тщательно рассмотрите приведённое выше расширение `MAX`, вы заметите определённые недостатки. Выражения вычисляются дважды; это плохо, если они влекут за собой побочные эффекты, вызванные, например, обращениями к функциям или использованием операций увеличения. Нужно позаботиться о правильном использовании круглых скобок, чтобы гарантировать сохранение требуемого порядка вычислений.[1] Здесь возникают даже некоторые чисто лексические проблемы: между именем макрос и левой круглой скобкой, открывающей список её аргументов, не должно быть никаких пробелов.

Тем не менее аппарат макросов является весьма ценным. Один практический пример даёт описываемая в главе 7 стандартная библиотека ввода-вывода, в которой `getchar` и `putchar` определены как макросы (очевидно `putchar` должна иметь аргумент), что позволяет избежать затрат на обращение к функции при обработке каждого символа.

Другие возможности макропроцессора описаны в приложении А.

Упражнение 4–9

Определите макрос `SWAP(X,Y)`, который обменивает значениями два своих аргумента типа `int`. (В этом случае поможет блочная структура).

[1] Рассмотрите макрос `#define SQUARE(X) X*X` при обращении к нему, как `SQUARE(z+1)`.

5 Указатели и массивы

Указатель – это переменная, содержащая адрес другой переменной. Указатели очень широко используются в языке **C**. Это происходит отчасти потому, что иногда они дают единственную возможность выразить нужное действие, а отчасти потому, что они обычно ведут к более компактным и эффективным программам, чем те, которые могут быть получены другими способами.

Указатели обычно смешивают в одну кучу с операторами `goto`, характеризуя их как чудесный способ написания программ, которые невозможно понять. Это безусловно справедливо, если указатели используются беззаботно; очень просто ввести указатели, которые указывают на что-то совершенно неожиданное. Однако, при определённой дисциплине, использование указателей помогает достичь ясности и простоты. Именно этот аспект мы попытаемся здесь проиллюстрировать.

5.1 Указатели и адреса

Так как указатель содержит адрес объекта, это даёт возможность «косвенного» доступа к этому объекту через указатель. Предположим, что x – переменная, например, типа `int`, а px – указатель, созданный неким ещё не указанным способом. Унарная операция & выдаёт адрес объекта, так что оператор

```
px = &x;
```

присваивает адрес x переменной px; говорят, что px «указывает» на x. Операция & применима только к переменным и элементам массива, конструкции вида &(x-1) и &3 являются незаконными. Нельзя также получить адрес регистровой переменной.

Унарная операция $*$ рассматривает свой операнд как адрес конечной цели и обращается по этому адресу, чтобы извлечь содержимое. Следовательно, если y тоже имеет тип `int`, то

```
y = *px;
```

присваивает y содержимое того, на что указывает px. Так последовательность

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

присваивает y то же самое значение, что и оператор

```
y = x;
```

Переменные, участвующие во всем этом необходимо описать:

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

с описанием для x и y мы уже неоднократно встречались. Описание указателя

```
int *px;
```

является новым и должно рассматриваться как мнемоническое; оно говорит, что комбинация *px имеет тип int. Это означает, что если px появляется в контексте *px, то это эквивалентно переменной типа int. Фактически синтаксис описания переменной имитирует синтаксис выражений, в которых эта переменная может появляться. Это замечание полезно во всех случаях, связанных со сложными описаниями. Например,

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

говорит, что atof() и *dp имеют в выражениях значения типа double.

Вы должны также заметить, что из этого описания следует, что указатель может указывать только на определённый вид объектов.

Указатели могут входить в выражения. Например, если px указывает на целое x, то *px может появляться в любом контексте, где может встретиться x. Так оператор

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

присваивает y значение, на 1 большее значения x;

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

печатает текущее значение x;

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

получает в d квадратный корень из x, причём до передачи функции sqrt значение x преобразуется к типу double. (Смотри главу 2).

В выражениях вида

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

унарные операции * и & связаны со своим операндом более крепко, чем арифметические операции, так что такое выражение берет то значение, на которое указывает px, прибавляет 1 и присваивает результат переменной y. Мы вскоре вернёмся к тому, что может означать выражение

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

Ссылки на указатели могут появляться и в левой части присваиваний. Если px указывает на x, то

```
* px = 0
```

полагает x равным нулю, а

```
* px += 1
```

увеличивает его на единицу, как и выражение

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

Круглые скобки в последнем примере необходимы; если их опустить, то поскольку унарные операции, подобные * и ++, выполняются справа налево, это выражение увеличит px, а не ту переменную, на которую он указывает.

И наконец, так как указатели являются переменными, то с ними можно обращаться, как и с остальными переменными. Если py – другой указатель на переменную типа `int`, то

```
py = px
```

копирует содержимое px в py, в результате чего py указывает на то же, что и px.

5.2 Указатели и аргументы функций

Так как в **C** передача аргументов функциям осуществляется «по значению», вызванная процедура не имеет непосредственной возможности изменить переменную из вызывающей программы. Что же делать, если вам действительно надо изменить аргумент? Например, программа сортировки захотела бы поменять два нарушающих порядок элемента с помощью функции с именем `swap`. Для этого недостаточно написать

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

определив функцию `swap` при этом следующим образом:

```
swap(x, y)      /* wrong */
int x, y;
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

из-за вызова по значению `swap` не может воздействовать на аргументы a и b в вызывающей функции.

К счастью, все же имеется возможность получить желаемый эффект. Вызывающая программа передаёт указатели подлежащих изменению значений:

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

так как операция & выдаёт адрес переменной, то &a является указателем на a. В самой swap аргументы описываются как указатели и доступ к фактическим операндам осуществляется через них.

```
swap(px, py)    /* interchange *px and *py */
int *px, *py;
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

Указатели в качестве аргументов обычно используются в функциях, которые должны возвращать более одного значения. (Можно сказать, что swap возвращает два значения, новые значения её аргументов). В качестве примера рассмотрим функцию getint, которая осуществляет преобразование поступающих в свободном формате данных, разделяя поток символов на целые значения, по одному целому за одно обращение. Функция getint должна возвращать либо найденное значение, либо признак конца файла, если входные данные полностью исчерпаны. Эти значения должны возвращаться как отдельные объекты, какое бы значение ни использовалось для EOF, даже если это значение вводимого целого.

Одно из решений, основывающееся на описываемой в главе 7 функции ввода scanf, состоит в том, чтобы при выходе на конец файла getint возвращала EOF в качестве значения функции; любое другое возвращённое значение говорит о нахождении нормального целого. Численное же значение найденного целого возвращается через аргумент, который должен быть указателем целого. Эта организация разделяет статус конца файла и численные значения.

Следующий цикл заполняет массив целыми с помощью обращений к функции getint:

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

В результате каждого обращения v становится равным следующему целому значению, найденному во входных данных. Обратите внимание, что в качестве аргумента getint необходимо указать &v а не v. Использование просто v скорее всего приведёт к ошибке адресации, поскольку getint полагает, что она работает именно с указателем.

Сама getint является очевидной модификацией написанной нами ранее функции atoi:

```
getint(pn)      /* get next integer from input */
int *pn;
{
  int c, sign;

  while ((c = getch()) == ' ' || c == '\n' || c == '\t')
```

```
    ; /* skip white space */
  sign = 1;
  if (c == '+' || c == '-') {   /* record sign */
    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);
}
```

Выражение *pn используется всюду в `getint` как обычная переменная типа `int`. Мы также использовали функции `getch` и `ungetch` (описанные в главе 4), так что один лишний символ, который приходится считывать, может быть помещён обратно во ввод.

Упражнение 5–1

Напишите функцию `getfloat`, аналог `getint` для чисел с плавающей точкой. Какой тип должна возвращать `getfloat` в качестве значения функции?

5.3 Указатели и массивы

В языке **C** существует сильная взаимосвязь между указателями и массивами, настолько сильная, что указатели и массивы действительно следует рассматривать одновременно. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей. Вариант с указателями обычно оказывается более быстрым, но и несколько более трудным для непосредственного понимания, по крайней мере для начинающего. Описание

```
int a[10]
```

определяет массив размера 10, т.е. набор из 10 последовательных объектов, называемых a[0], a[1], . . . , a[9]. Запись a[i] соответствует элементу массива через i позиций от начала. Если pa – указатель целого, описанный как

```
int *pa
```

то присваивание

```
pa = &a[0]
```

приводит к тому, что pa указывает на нулевой элемент массива a; это означает, что pa содержит адрес элемента a[0]. Теперь присваивание

```
x = *pa
```

будет копировать содержимое a[0] в x.

Если pa указывает на некоторый определённый элемент массива a, то по определению pa+1 указывает на следующий элемент, и вообще pa−i указывает на элемент, стоящий на i позиций до элемента, указываемого pa, а pa+i на элемент, стоящий на i позиций после. Таким образом, если pa указывает на a[0], то ∗(pa+1) ссылается на содержимое a[1], pa+i – адрес a[i], а ∗(pa+i) содержимое a[i].

Эти замечания справедливы независимо от типа переменных в массиве a. Суть определения «добавления 1 к указателю», а также его распространения на всю арифметику указателей, состоит в том, что приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель. Таким образом, i в pa+i перед прибавлением умножается на размер объектов, на которые указывает pa.

Очевидно существует очень тесное соответствие между индексацией и арифметикой указателей. В действительности компилятор преобразует ссылку на массив в указатель на начало массива. В результате этого имя массива является указательным выражением. Отсюда вытекает несколько весьма полезных следствий. Так как имя массива является синонимом местоположения его нулевого элемента, то присваивание pa=&a[0] можно записать как

```
pa = a
```

Ещё более удивительным, по крайней мере на первый взгляд, кажется тот факт, что ссылку на a[i] можно записать в виде ∗(a+i). При анализировании выражения a[i] в языке **C** оно немедленно преобразуется к виду ∗(a+i); эти две формы совершенно эквивалентны. Если применить операцию & к обеим частям такого соотношения эквивалентности, то мы получим, что &a[i] и a+i тоже идентичны: a+i – адрес i-го элемента от начала a. С другой стороны, если pa является указателем, то в выражениях его можно использовать с индексом: pa[i] идентично ∗(pa+i). Короче, любое выражение, включающее массивы и индексы, может быть записано через указатели и смещения и наоборот, причём даже в одном и том же утверждении.

Имеется одно различие между именем массива и указателем, которое необходимо иметь в виду. Указатель является переменной, так что операции pa=a и pa++ имеют смысл. Но имя массива является константой, а не переменной: конструкции типа a=pa или a++, или p=&a будут незаконными.

Когда имя массива передаётся функции, то на самом деле ей передаётся местоположение начала этого массива. Внутри вызванной функции такой аргумент является точно такой же переменной, как и любая другая, так что имя массива в качестве аргумента действительно является указателем, т.е. переменной, содержащей адрес. Мы можем использовать это обстоятельство для написания нового варианта функции `strlen`, вычисляющей длину строки.

```
strlen(s)       /* return length of string s */
char *s;
{
  int n;
```

```
  for (n = 0; *s != '\0'; s++)
    n++;
  return (n);
}
```

Операция увеличения s совершенно законна, поскольку эта переменная является указателем; s++ никак не влияет на символьную строку в обратившейся к `strlen` функции, а только увеличивает локальную для функции `strlen` копию адреса. Описания формальных параметров в определении функции в виде

```
char s[];
char *s;
```

совершенно эквивалентны; какой вид описания следует предпочесть, определяется в значительной степени тем, какие выражения будут использованы при написании функции. Если функции передаётся имя массива, то в зависимости от того, что удобнее, можно полагать, что функция оперирует либо с массивом, либо с указателем, и действовать далее соответствующим образом. Можно даже использовать оба вида операций, если это кажется уместным и ясным.

Можно передать функции часть массива, если задать в качестве аргумента указатель начала подмассива. Например, если a – массив, то как

```
f(&a[2])
```

как и

```
f(a + 2)
```

передают функции f адрес элемента a[2], потому что и &a[2], и a+2 являются указательными выражениями, ссылающимися на третий элемент a. Внутри функции f описания аргументов могут присутствовать в виде:

```
f(arr)
int arr[];
{
  ...
}
```

или

```
f(arr)
int *arr;
{
  ...
}
```

Что касается функции f, то тот факт, что её аргумент в действительности ссылается к части большего массива, не имеет для неё никаких последствий.

5.4 Адресная арифметика

Если p является указателем, то каков бы ни был сорт объекта, на который он указывает, операция p++ увеличивает p так, что он указывает на следующий элемент набора этих объектов, а операция p +=i увеличивает p так, чтобы он указывал на элемент, отстоящий на i элементов от текущего элемента. Эти и аналогичные конструкции представляют собой самые простые и самые распространённые формы арифметики указателей или адресной арифметики.

Язык **C** последователен и постоянен в своём подходе к адресной арифметике; объединение в одно целое указателей, массивов и адресной арифметики является одной из наиболее сильных сторон языка. Давайте проиллюстрируем некоторые из соответствующих возможностей языка на примере элементарной (но полезной, несмотря на свою простоту) программы распределения памяти. Имеются две функции: функция `alloc(n)` возвращает в качестве своего значения указатель p, который указывает на первую из n последовательных символьных позиций, которые могут быть использованы вызывающей функцию `alloc` программой для хранения символов; функция `free(p)` освобождает приобретённую таким образом память, так что её в дальнейшем можно снова использовать. Программа является «элементарной», потому что обращения к `free` должны производиться в порядке, обратном тому, в котором производились обращения к `alloc`. Таким образом, управляемая функциями `alloc` и `free` память является стеком или списком, в котором последний вводимый элемент извлекается первым. Стандартная библиотека языка **C** содержит аналогичные функции, не имеющие таких ограничений, и, кроме того, в главе 8 мы приведём улучшенные варианты. Между тем, однако, для многих приложений нужна только тривиальная функция `alloc` для распределения небольших участков памяти неизвестных заранее размеров в непредсказуемые моменты времени.

Простейшая реализация состоит в том, чтобы функция раздавала отрезки большого символьного массива, которому мы присвоили имя `allocbuf`. Этот массив является собственностью функций `alloc` и `free`. Так как они работают с указателями, а не с индексами массива, никакой другой функции не нужно знать имя этого массива. Он может быть описан как внешний статический, т.е. он будет локальным по отношению к исходному файлу, содержащему `alloc` и `free`, и невидимым за его пределами. При практической реализации этот массив может даже не иметь имени; вместо этого он может быть получен в результате запроса к операционной системе на указатель некоторого неименованного блока памяти.

Другой необходимой информацией является то, какая часть массива `allocbuf` уже использована. Мы пользуемся указателем первого свободного элемента, названным `allocp`. Когда к функции `alloc` обращаются за выделением n символов, то она проверяет, достаточно ли осталось для этого места в `allocbuf`. Если достаточно, то `alloc` возвращает текущее значение `allocp` (т.е. начало свободного блока), затем увеличивает его на n, с тем чтобы он указывал на следующую свободную область. Функция `free(p)` просто полагает `allocp` равным p при условии, что p указывает на позицию внутри `allocbuf`.

```
#define NULL 0  /* pointer value for error report */
```

```
#define ALLOCSIZE 1000  /* size of available space */

static char allocbuf[ALLOCSIZE];        /* storage for alloc */
static char *allocp = allocbuf;         /* next free position */

char *alloc(n)  /* return pointer to n characters */
int n;
{
  if (allocp + n <= allocbuf + ALLOCSIZE) {
    allocp += n;
    return (allocp - n);        /* old p */
  } else        /* not enough room */
    return (NULL);
}

free(p) /* free storage pointed by p */
char *p;
{
  if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
    allocp = p;
}
```

Дадим некоторые пояснения. Вообще говоря, указатель может быть инициализирован точно так же, как и любая другая переменная, хотя обычно единственными осмысленными значениями являются NULL (это обсуждается ниже) или выражение, включающее адреса ранее определённых данных соответствующего типа. Описание

```
static char *allocp = allocbuf;
```

определяет `allocp` как указатель на символы и инициализирует его так, чтобы он указывал на `allocbuf`, т.е. на первую свободную позицию при начале работы программы. Так как имя массива является адресом его нулевого элемента, то это можно было бы записать в виде

```
static char *allocp = &allocbuf[0];
```

используйте ту запись, которая вам кажется более естественной. С помощью проверки

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

выясняется, осталось ли достаточно места, чтобы удовлетворить запрос на n символов. Если достаточно, то новое значение `allocp` не будет указывать дальше, чем на последнюю позицию `allocbuf`. Если запрос может быть удовлетворён, то `alloc` возвращает обычный указатель (обратите внимание на описание самой функции). Если же нет, то `alloc` должна вернуть некоторый признак, говорящий о том, что больше места не осталось. В языке **C** гарантируется, что ни один правильный указатель данных не может иметь значение нуль, так что возвращение нуля может служить в качестве сигнала о ненормальном событии, в данном случае об отсутствии места. Мы, однако, вместо нуля

пишем NULL, с тем чтобы более ясно показать, что это специальное значение указателя. Вообще говоря, целые не могут осмысленно присваиваться указателям, а нуль – это особый случай.

Проверки вида

```
if (allocp + n <= allocbuf + ALOOCSIZE)
```

и

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

демонстрируют несколько важных аспектов арифметики указателей. Во-первых, при определённых условиях указатели можно сравнивать. Если p и q указывают на элементы одного и того же массива, то такие отношения, как <, >= и т.д., работают надлежащим образом. Например,

```
p < q
```

истинно, если p указывает на более ранний элемент массива, чем q. Отношения == и != тоже работают. Любой указатель можно осмысленным образом сравнить на равенство или неравенство с NULL. Но ни за что нельзя ручаться, если вы используете сравнения при работе с указателями, указывающими на разные массивы. Если вам повезёт, то на всех машинах вы получите очевидную бессмыслицу. Если же нет, то ваша программа будет правильно работать на одной машине и давать непостижимые результаты на другой.

Во-вторых, как мы уже видели, указатель и целое можно складывать и вычитать. Конструкция

```
p + n
```

подразумевает n-ый объект за тем, на который p указывает в настоящий момент. Это справедливо независимо от того, на какой вид объектов p должен указывать; компилятор сам масштабирует n в соответствии с определяемым из описания p размером объектов, указываемых с помощью p. Например, на PDP-11 масштабирующий множитель равен 1 для char, 2 для int и short, 4 для long и float и 8 для double.

Вычитание указателей тоже возможно: если p и q указывают на элементы одного и того же массива, то p–q – количество элементов между p и q. Этот факт можно использовать для написания ещё одного варианта функции strlen:

```
strlen(s)       /* return length of string s */
char *s;
{
  char *p = s;

  while (*p != '\0')
    p++;
  return (p - s);
}
```

При описании указатель p в этой функции инициализирован посредством строки s, в результате чего он указывает на первый символ строки. В цикле `while` по очереди проверяется каждый символ до тех пор, пока не появится символ конца строки \0. Так как значение \0 равно нулю, а `while` только выясняет, имеет ли выражение в нем значение 0, то в данном случае явную проверку можно опустить. Такие циклы часто записывают в виде

```
while (*p)
  p++;
```

Так как p указывает на символы, то оператор p++ передвигает p каждый раз так, чтобы он указывал на следующий символ. В результате p−s даёт число просмотренных символов, т.е. длину строки. Арифметика указателей последовательна: если бы мы имели дело с переменными типа `float`, которые занимают больше памяти, чем переменные типа `char`, и если бы p был указателем на `float`, то оператор p++ передвинул бы p на следующее `float`. таким образом, мы могли бы написать другой вариант функции `alloc`, распределяющей память для `float`, вместо `char`, просто заменив всюду в `alloc` и `free` описатель `char` на `float`. Все действия с указателями автоматически учитывают размер объектов, на которые они указывают, так что больше ничего менять не надо.

За исключением упомянутых выше операций (сложение и вычитание указателя и целого, вычитание и сравнение двух указателей), вся остальная арифметика указателей является незаконной. Запрещено складывать два указателя, умножать, делить, сдвигать или маскировать их, а также прибавлять к ним переменные типа `float` или `double`.

5.5 Указатели символов и функции

Строчная константа, как, например,

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

является массивом символов. Компилятор завершает внутреннее представление такого массива символом \0, так что программы могут находить его конец. Таким образом, длина массива в памяти оказывается на единицу больше числа символов между двойными кавычками.

По-видимому чаще всего строчные константы появляются в качестве аргументов функций, как, например, в

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

когда символьная строка, подобная этой, появляется в программе, то доступ к ней осуществляется с помощью указателя символов; функция `printf` фактически получает указатель символьного массива.

Конечно, символьные массивы не обязаны быть только аргументами функций. Если описать message как

```
char *message;
```

то в результате оператора

```
message = "now is the time";
```

переменная message станет указателем на фактический массив символов. Это не копирование строки; здесь участвуют только указатели. В языке **C** не предусмотрены какие-либо операции для обработки всей строки символов как целого.

Мы проиллюстрируем другие аспекты указателей и массивов, разбирая две полезные функции из стандартной библиотеки ввода-вывода, которая будет рассмотрена в главе 7.

Первая функция – это `strcpy(s,t)`, которая копирует строку t в строку s. Аргументы написаны именно в этом порядке по аналогии с операцией присваивания, когда для того, чтобы присвоить t к s обычно пишут

```
s = t
```

сначала приведём версию с массивами:

```
strcpy(s, t)    /* copy t to s */
char s[], t[];
{
  int i;
  i = 0;
  while ((s[i] = t[i]) != '\0')
    i++;
}
```

Для сопоставления ниже даётся вариант `strcpy` с указателями.

```
strcpy(s, t)    /* copy t to s; pointer version 1 */
char *s, *t;
{
  while ((*s = *t) != '\0') {
    s++;
    t++;
  }
}
```

Так как аргументы передаются по значению, функция `strcpy` может использовать s и t так, как она пожелает. Здесь они с удобством полагаются указателями, которые передвигаются вдоль массивов, по одному символу за шаг, пока не будет скопирован в s завершающий в t символ \0.

На практике функция `strcpy` была бы записана не так, как мы показали выше. Вот вторая возможность:

```
strcpy(s, t)    /* copy t to s; pointer version 2 */
char *s, *t;
{
  while ((*s++ = *t++) != '\0');
}
```

Здесь увеличение s и t внесено в проверочную часть. Значением `*t++` является символ, на который указывал t до увеличения; постфиксная операция ++ не изменяет t, пока этот символ не будет извлечён. Точно так же этот символ помещается в старую позицию s, до того как s будет увеличено. Конечный результат заключается в том, что все символы, включая завершающий \0, копируются из t в s.

И как последнее сокращение мы опять отметим, что сравнение с \0 является излишним, так что функцию можно записать в виде

```
strcpy(s, t)    /* copy t to s; pointer version 3 */
char *s, *t;
{
  while (*s++ = *t++);
}
```

хотя с первого взгляда эта запись может показаться загадочной, она даёт значительное удобство. Этой идиомой следует овладеть уже хотя бы потому, что вы с ней будете часто встречаться в **C**-программах.

Вторая функция – `strcmp(s, t)`, которая сравнивает символьные строки s и t, возвращая отрицательное, нулевое или положительное значение в соответствии с тем, меньше, равно или больше лексикографически s, чем t. Возвращаемое значение получается в результате вычитания символов из первой позиции, в которой s и t не совпадают.

```
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]);
}
```

Вот версия `strcmp` с указателями:

```
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 ++ и −− могут быть как постфиксными, так и префиксными операциями, встречаются другие комбинации ∗ и ++ и −−, хотя и менее часто. Например

```
* ++p
```

увеличивает p до извлечения символа, на который указывает p, а

```
* --p
```

сначала уменьшает p.

Упражнение 5–2

Напишите вариант с указателями функции `strcat` из главы 2: `strcat(s,t)` копирует строку t в конец s.

Упражнение 5–3

Напишите макрос для `strcpy`.

Упражнение 5–4

Перепишите подходящие программы из предыдущих глав и упражнений, используя указатели вместо индексации массивов. Хорошие возможности для этого предоставляют функции `getline` (главы 1 и 4), `atoi`, `itoa` и их варианты (главы 2, 3 и 4), `reverse` (глава 3), `index` и `getop` (глава 4).

5.6 Указатели – не целые

Вы, возможно, обратили внимание в предыдущих **C**-программах на довольно непринуждённое отношение к копированию указателей. В общем это верно, что на большинстве машин указатель можно присвоить целому и передать его обратно, не изменив его; при этом не происходит никакого масштабирования или преобразования и ни один бит не теряется. К сожалению, это ведёт к вольному обращению с функциями, возвращающими указатели, которые затем просто передаются другим функциям, – необходимые описания указателей часто опускаются. Рассмотрим, например, функцию `strsave(s)`, которая копирует строку s в некоторое место для хранения, выделяемое посредством обращения к функции `alloc`, и возвращает указатель на это место. Правильно она должна быть записана так:

```
char *strsave(s)        /* save string s somewhere */
char *s;
{
  char *p, *alloc();

  if ((p = alloc(strlen(s) + 1)) != NULL)
    strcpy(p, s);
  return (p);
}
```

на практике существует сильное стремление опускать описания:

```
* strsave(s)
{       /* save string s somewhere */
  char *p;

  if ((p = alloc(strlen(s) + 1)) != NULL)
    strcpy(p, s);
  return (p);
}
```

Эта программа будет правильно работать на многих машинах, потому что по умолчанию функции и аргументы имеют тип `int`, а указатель и целое обычно можно безопасно пересылать туда и обратно. Однако такой стиль программирования в своём существе является рискованным, поскольку зависит от деталей реализации и архитектуры машины и может привести к неправильным результатам на конкретном используемом вами компиляторе. Разумнее всюду использовать полные описания.[1]

5.7 Многомерные массивы

В языке **C** предусмотрены прямоугольные многомерные массивы, хотя на практике существует тенденция к их значительно более редкому использованию по сравнению с массивами указателей. В этом разделе мы рассмотрим некоторые их свойства.

Рассмотрим задачу преобразования дня месяца в день года и наоборот. Например, 1-ое марта является 60-м днем невисокосного года и 61-м днем високосного года. Давайте введём две функции для выполнения этих преобразований: `day_of_year` преобразует месяц и день в день года, а `month_day` преобразует день года в месяц и день. Так как эта последняя функция возвращает два значения, то аргументы месяца и дня должны быть указателями. Так вызов:

```
month_day(1977, 60, &m, &d)
```

полагает m равным 3 и d равным 1 (1-ое марта).

Обе эти функции нуждаются в одной и той же информационной таблице, указывающей число дней в каждом месяце. Так как число дней в месяце в високосном и в невисокосном году отличается, то проще представить их в виде двух строк двумерного массива, чем пытаться прослеживать во время вычислений, что именно происходит в феврале. Вот этот массив и выполняющие эти преобразования функции:

```
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)
};
```

[1] Отладочная программа `lint` предупредит о таких конструкциях, если они по неосторожности все же появятся.

```
day_of_year(year, month, day)   /* set day of year */
int year, month, day;   /* from month & day */
{
  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)  /* set month,day */
int year, yearday, *pmonth, *pday;      /* from day of year */
{
  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` должен быть внешним как для `day_of_year`, так и для `month_day`, поскольку он используется обеими этими функциями.

Массив `day_tab` является первым двумерным массивом, с которым мы имеем дело. По определению в **C** двумерный массив по существу является одномерным массивом, каждый элемент которого является массивом. Поэтому индексы записываются как

```
day_tab[i][j]
```

а не

```
day_tab[i, j]
```

как в большинстве языков. В остальном с двумерными массивами можно в основном обращаться таким же образом, как в других языках. Элементы хранятся по строкам, т.е. при обращении к элементам в порядке их размещения в памяти быстрее всего изменяется самый правый индекс.

Массив инициализируется с помощью списка начальных значений, заключённых в фигурные скобки; каждая строка двумерного массива инициализируется соответствующим подсписком. Мы поместили в начало массива `day_tab` столбец из нулей для того, чтобы номера месяцев изменялись естественным образом от 1 до 12, а не от 0 до 11. Так как за экономию памяти у нас пока не награждают, такой способ проще, чем подгонка индексов.

Если двумерный массив передаётся функции, то описание соответствующего аргумента функции должно содержать количество столбцов; количество строк несущественно, поскольку, как и прежде, фактически передаётся указатель. В нашем конкретном случае это указатель объектов, являющихся массивами из 13 чисел типа `int`. Таким образом, если бы требовалось передать массив `day_tab` функции f, то описание в f имело бы вид:

```
f(day_tab)
int day_tab[2][13];
{
  ...
}
```

Так как количество строк является несущественным, то описание аргумента в f могло бы быть таким:

```
int day_tab[][13];
```

или таким

```
int (*day_tab)[13];
```

в котором говорится, что аргумент является указателем массива из 13 целых. Круглые скобки здесь необходимы, потому что квадратные скобки [] имеют более высокий уровень старшинства, чем $*$; как мы увидим в следующем разделе, без круглых скобок

```
int *day_tab[13];
```

является описанием массива из 13 указателей на целые.

5.8 Массивы указателей; указатели указателей

Так как указатели сами являются переменными, то вы вполне могли бы ожидать использования массива указателей. Это действительно так. Мы проиллюстрируем это написанием программы сортировки в алфавитном порядке набора текстовых строк, предельно упрощённого варианта утилиты `sort` операционной систем UNIX.

В главе 3 мы привели функцию сортировки по Шеллу, которая упорядочивала массив целых. Этот же алгоритм будет работать и здесь, хотя теперь мы будем иметь дело со строчками текста различной длины, которые, в отличие от целых, нельзя сравнивать или перемещать с помощью одной операции. Мы нуждаемся в таком представлении данных, которое бы позволяло удобно и эффективно обрабатывать строки текста переменной длины.

Здесь и возникают массивы указателей. Если подлежащие сортировке строки хранятся одна за другой в длинном символьном массиве (управляемом, например, функцией `alloc`), то к каждой строке можно обратиться с помощью указателя на её первый символ. Сами указатели можно хранить в массиве. Две строки можно сравнить, передав их указатели функции `strcmp`.

Если две расположенные в неправильном порядке строки должны быть переставлены, то фактически переставляются указатели в массиве указателей, а не сами тексты строк. Этим исключаются сразу две связанные проблемы: сложного управления памятью и больших дополнительных затрат на фактическую перестановку строк.

Процесс сортировки включает три шага:

- чтение всех строк ввода

- их сортировка
- вывод их в правильном порядке

Как обычно, лучше разделить программу на несколько функций в соответствии с естественным делением задачи и выделить ведущую функцию, управляющую работой всей программы. Давайте отложим на некоторое время рассмотрение шага сортировки и сосредоточимся на структуре данных и вводе-выводе. Функция, осуществляющая ввод, должна извлечь символы каждой строки, запомнить их и построить массив указателей строк. Она должна также подсчитать число строк во вводе, так как эта информация необходима при сортировке и выводе. Так как функция ввода в состоянии справиться только с конечным числом вводимых строк, в случае слишком большого их числа она может возвращать некоторое число, отличное от возможного числа строк, например -1. Функция осуществляющая вывод, должна печатать строки в том порядке, в каком они появляются в массиве указателей.

```
#define NULL 0
#define LINES 100       /* max lines to be sorted */

main()
{       /* sort input lines */
  char *lineptr[LINES]; /* pointers to text lines */
  int nlines;           /* number of input lines read */

  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)    /* read input lines */
char *lineptr[];                /* for sorting */
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';     /* zap newline */
        strcpy(p, line);
        lineptr[nlines++] = p;
      }
    return (nlines);
}
```

Символ новой строки в конце каждой строки удаляется, так что он никак не будет влиять на порядок, в котором сортируются строки.

```
writelines(lineptr, nlines)     /* write output lines */
char *lineptr[];
int nlines;
{
  int i;

  for (i = 0; i < nlines; i++)
    printf("%s\n", lineptr[i]);
}
```

Существенно новым в этой программе является описание

```
char *lineptr[LINES];
```

которое сообщает, что `lineptr` является массивом из `LINES` элементов, каждый из которых – указатель на переменные типа `char`. Это означает, что `lineptr[i]` – указатель на символы, а `*lineptr[i]` извлекает символ.

Так как сам `lineptr` является массивом, который передаётся функции `writelines`, с ним можно обращаться как с указателем точно таким же образом, как в наших более ранних примерах. Тогда последнюю функцию можно переписать в виде:

```
writelines(lineptr, nlines)     /* write output lines */
char *lineptr[];
int nlines;
{
  while (--nlines >= 0)
    printf("%s\n", *lineptr++);
}
```

здесь `*lineptr` сначала указывает на первую строку; каждое увеличение передвигает указатель на следующую строку, в то время как `nlines` убывает до нуля.

Справившись с вводом и выводом, мы можем перейти к сортировке. Программа сортировки по Шеллу из главы 3 требует очень небольших изменений: должны быть модифицированы описания, а операция сравнения выделена в отдельную функцию. Основной алгоритм остаётся тем же самым, и это даёт нам определённую уверенность, что он по-прежнему будет работать.

```
sort(v, n)      /* sort strings v[0] ... v[n-1] */
char *v[];      /* into increasing order */
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 каждый отдельный элемент массива v (имя формального параметра, соответствующего `lineptr`) является указателем на символы, то и `temp` должен быть указателем на символы, чтобы их было можно копировать друг в друга.

Мы написали эту программу по возможности более просто с тем, чтобы побыстрее получить работающую программу. Она могла бы работать быстрее, если, например, вводить строки непосредственно в массив, управляемый функцией `readlines`, а не копировать их в `line`, а затем в скрытое место с помощью функции `alloc`. Но мы считаем, что будет разумнее первоначальный вариант сделать более простым для понимания, а об «эффективности» позаботиться позднее. Все же, по-видимому, способ, позволяющий добиться заметного ускорения работы программы состоит не в исключении лишнего копирования вводимых строк. Более вероятно, что существенной разницы можно достичь за счёт замены сортировки по Шеллу на нечто лучшее, например, на метод быстрой сортировки.

В главе 1 мы отмечали, что поскольку в циклах `while` и `for` проверка осуществляется до того, как тело цикла выполнится хотя бы один раз, эти циклы оказываются удобными для обеспечения правильной работы программы при граничных значениях, в частности, когда ввода вообще нет. Очень полезно просмотреть все функции программы сортировки, разбираясь, что происходит, если вводимый текст отсутствует.

Упражнение 5–5

Перепишите функцию `readlines` таким образом, чтобы она помещала строки в массив, предоставляемый функцией `main`, а не в память, управляемую обращениями к функции `alloc`. Насколько быстрее стала программа?

5.9 Инициализация массивов указателей

Рассмотрим задачу написания функции `month_name(n)`, которая возвращает указатель на символьную строку, содержащую имя n-го месяца. Это идеальная задача для применения внутреннего статического массива. Функция `month_name` содержит локальный массив символьных строк и при обращении к ней возвращает указатель нужной строки. Тема настоящего раздела как инициализировать этот массив имён.

```
char *month_name(n)     /* return name of n-th month */
int n;
{
  static char *name[] = {
    "illegal month",
    "JANUARY",
    "FEBRUARY",
    "MARCH",
    "APRIL",
    "MAY",
    "JUN",
    "JULY",
    "AUGUST",
    "SEPTEMBER",
    "OCTOBER",
    "NOVEMBER",
    "DECEMBER"
  };
  return ((n < 1 || n > 12) ? name[0] : name[n]);
}
```

Описание массива указателей на символы `name` точно такое же, как аналогичное описание `lineptr` в примере с сортировкой. Инициализатором является просто список символьных строк; каждая строка присваивается соответствующей позиции в массиве. Более точно, символы i-ой строки помещаются в какое-то иное место, а её указатель хранится в `name[i]`. Поскольку размер массива `name` не указан, компилятор сам подсчитывает количество инициализаторов и соответственно устанавливает правильное число.

5.10 Указатели и многомерные массивы

Начинающие изучать язык **C** иногда становятся в тупик перед вопросом о различии между двумерным массивом и массивом указателей, таким как `name` в приведённом выше примере. Если имеются описания

```
int a[10][10];
int *b[10];
```

то a и b можно использовать сходным образом в том смысле, что как a[5][5], так и b[5][5] являются законными ссылками на отдельное число типа `int`. Но a – настоящий массив:

под него отводится 100 ячеек памяти и для нахождения любого указанного элемента проводятся обычные вычисления с прямоугольными индексами. Для b, однако, описание выделяет только 10 указателей; каждый указатель должен быть установлен так, чтобы он указывал на массив целых. Если предположить, что каждый из них указывает на массив из 10 элементов, то тогда где-то будет отведено 100 ячеек памяти плюс ещё десять ячеек для указателей. Таким образом, массив указателей использует несколько больший объем памяти и может требовать наличие явного шага инициализации. Но при этом возникают два преимущества: доступ к элементу осуществляется косвенно через указатель, а не посредством умножения и сложения, и строки массива могут иметь различные длины. Это означает, что каждый элемент b не должен обязательно указывать на вектор из 10 элементов; некоторые могут указывать на вектор из двух элементов, другие – из двадцати, а третьи могут вообще ни на что не указывать.

Хотя мы вели это обсуждение в терминах целых, несомненно, чаще всего массивы указателей используются так, как мы продемонстрировали на функции `month_name`, – для хранения символьных строк различной длины.

Упражнение 5–6

Перепишите функции `day_of_year` и `month_day`, используя вместо индексации указатели.

5.11 Командная строка аргументов

Системные средства, на которые опирается реализация языка **C**, позволяют передавать командную строку аргументов или параметров начинающей выполняться программе. Когда функция `main` вызывается к исполнению, она вызывается с двумя аргументами. Первый аргумент (условно называемый `argc`) указывает число аргументов в командной строке, с которыми происходит обращение к программе; второй аргумент (`argv`) является указателем на массив символьных строк, содержащих эти аргументы, по одному в строке. Работа с такими строками – это обычное использование многоуровневых указателей.

Самую простую иллюстрацию этой возможности и необходимых при этом описаний даёт программа `echo`, которая просто печатает в одну строку аргументы командной строки, разделяя их пробелами. Таким образом, если дана команда

```
echo HELLO, WORLD
```

то выходом будет

```
HELLO, WORLD
```

по соглашению `argv[0]` является именем, по которому вызывается программа, так что `argc` по меньшей мере равен 1. В приведённом выше примере `argc` равен 3, а `argv[0]`,

argv[1] и argv[2] равны соответственно "echo", "HELLO", и "WORLD". Первым фактическим аргументом является argv[1], а последним argv[argc-1]. Если argc равен 1, то за именем программы не следует никакой командной строки аргументов. Все это показано в echo:

```
main(argc, argv)        /* echo arguments; 1st version */
int argc;
char *argv[];
{
  int i;

  for (i = 1; i < argc; i++)
    printf("%s%c", argv[i], (i < argc - 1) ? ' ' : '\n');
}
```

Поскольку argv является указателем на массив указателей, то существует несколько способов написания этой программы, использующих работу с указателем, а не с индексацией массива. Мы продемонстрируем два варианта.

```
main(argc, argv)        /* echo arguments; 2nd version */
int argc;
char *argv[];
{
  while (--argc > 0)
    printf("%s%c", *++argv, (argc > 1) ? ' ' : '\n');
}
```

Так как argv является указателем на начало массива строк-аргументов, то, увеличив его на 1 (++argv), мы вынуждаем его указывать на подлинный аргумент argv[1], а не на argv[0]. Каждое последующее увеличение передвигает его на следующий аргумент; при этом *argv становится указателем на этот аргумент. Одновременно величина argc уменьшается; когда она обратится в нуль, все аргументы будут уже напечатаны.

Другой вариант:

```
main(argc, argv)        /* echo arguments; 3rd version */
int argc;
char *argv[];
{
  while (--argc > 0)
    printf((argc > 1) ? "%s" : "%s\n", *++argv);
}
```

Эта версия показывает, что аргумент формата функции printf может быть выражением, точно так же, как и любой другой. Такое использование встречается не очень часто, но его все же стоит запомнить.

Как второй пример, давайте внесём некоторые усовершенствования в программу отыскания заданной комбинации символов из главы 4. Если вы помните, мы поместили

искомую комбинацию глубоко внутрь программы, что очевидно является совершенно неудовлетворительным. Следуя утилите grep системы UNIX, давайте изменим программу так, чтобы эта комбинация указывалась в качестве первого аргумента строки.

```
#define MAXLINE 1000

main(argc, argv)        /* find pattern from first argument */
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 может быть развита основная модель, иллюстрирующая дальнейшее использование указателей. Предположим, что нам надо предусмотреть два необязательных аргумента. Один утверждает: «напечатать все строки за исключением тех, которые содержат данную комбинацию», второй гласит: «перед каждой выводимой строкой должен печататься её номер».

Общепринятым соглашением в **C**-программах является то, что аргумент, начинающийся со знака минус, вводит необязательный признак или параметр. Если мы, для того, чтобы сообщить об инверсии, выберем -x, а для указания о нумерации нужных строк выберем -n(«номер»), то команда

```
find -x -n the
```

при входных данных

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

Должна выдать

```
2:for all good men
```

Нужно, чтобы необязательные аргументы могли располагаться в произвольном порядке, и чтобы остальная часть программы не зависела от количества фактически присутствующих аргументов. В частности, вызов функции index не должен содержать ссылку на argv[2], когда присутствует один необязательный аргумент, и на argv[1], когда его нет. Более того, для пользователей удобно, чтобы необязательные аргументы можно было объединить в виде:

```
find -nx the
```

вот сама программа:

```
#define MAXLINE 1000

main(argc, argv)        /* find pattern from first argument */
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);
      }
    }
}
```

Аргумент `argv` увеличивается перед каждым необязательным аргументом, в то время как аргумент `argc` уменьшается. Если нет ошибок, то в конце цикла величина `argc` должна равняться 1, а `*argv` должно указывать на заданную комбинацию. Обратите внимание на то, что `*++argv` является указателем аргументной строки; `(*++argv)[0]` – её первый символ. Круглые скобки здесь необходимы, потому что без них выражение бы приняло совершенно отличный (и неправильный) вид `*++(argv[0])`. Другой правильной формой была бы `**++argv`.

Упражнение 5–7

Напишите программу `add`, вычисляющую обратное польское выражение из командной строки. Например,

```
add 2 3 4 + *
```

вычисляет `2*(3+4)`.

Упражнение 5–8

Модифицируйте программы `entab` и `detab` (указанные в качестве упражнений в главе 1) так, чтобы они получали список табуляционных остановок в качестве аргументов. Если аргументы отсутствуют, используйте стандартную установку табуляций.

Упражнение 5–9

Расширьте `entab` и `detab` таким образом, чтобы они воспринимали сокращённую нотацию

```
entab m +n
```

означающую табуляционные остановки через каждые n столбцов, начиная со столбца m. Выберите удобное (для пользователя) поведение функции по умолчанию.

Упражнение 5–10

Напишите программу для функции `tail`, печатающей последние n строк из своего файла ввода. Пусть по умолчанию n равно 10, но это число может быть изменено с помощью необязательного аргумента, так что

```
tail -n
```

печатает последние n строк. Программа должна действовать рационально, какими бы неразумными ни были бы ввод или значение n. Составьте программу так, чтобы она оптимальным образом использовала доступную память: строки должны храниться, как в функции `sort`, а не в двумерном массиве фиксированного размера.

5.12 Указатели на функции

В языке **C** сами функции не являются переменными, но имеется возможность определить указатель на функцию, который можно обрабатывать, передавать другим функциям, помещать в массивы и т.д. Мы проиллюстрируем это, проведя модификацию написанной ранее программы сортировки так, чтобы при задании необязательного аргумента -n она бы сортировала строки ввода численно, а не лексикографически.

Сортировка часто состоит из трёх частей – сравнения, которое определяет упорядочивание любой пары объектов, перестановки, изменяющей их порядок, и алгоритма сортировки, осуществляющего сравнения и перестановки до тех пор, пока объекты не расположатся в нужном порядке. Алгоритм сортировки не зависит от операций сравнения и перестановки, так что, передавая в него различные функции сравнения и перестановки, мы можем организовать сортировку по различным критериям. Именно такой подход используется в нашей новой программе сортировки.

Как и прежде, лексикографическое сравнение двух строк осуществляется функцией `strcmp`, а перестановка функцией `swap`; нам нужна ещё функция `numcmp`, сравнивающая две строки на основе численного значения и возвращающая условное указание того же вида, что и `strcmp`. Эти три функции описываются в `main` и указатели на них передаются в `sort`. В свою очередь функция `sort` обращается к этим функциям через их указатели. Мы урезали обработку ошибок в аргументах с тем, чтобы сосредоточиться на главных вопросах.

```
#define LINES 100       /* max number of lines to be sorted */

main(argc, argv)        /* sort input lines */
int argc;
char *argv[];
{
  char *lineptr[LINES]; /* pointers to text lines */
  int nlines;   /* number of input lines read */
  int strcmp(), numcmp();       /* comparsion functions */
  int swap();   /* exchange function */
  int numeric = 0;      /* 1 if numeric sort */

  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, nimcmp и swap – адреса функций; так как известно, что это функции, операция & здесь не нужна совершенно аналогично тому, как она не нужна и перед именем массива. Передача адресов функций организуется компилятором.

Второй шаг состоит в модификации sort:

```
sort(v, n, comp, exch)  /* sort strings v[0] ... v[n-1] */
char *v[];              /* into increasing order */
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]);
      }
}
```

Здесь следует обратить определённое внимание на описания. Описание

```
int (*comp) ()
```

говорит, что comp является указателем на функцию, которая возвращает значение типа int. Первые круглые скобки здесь необходимы; без них описание

```
int *comp()
```

говорило бы, что comp является функцией, возвращающей указатель на целые, что, конечно, совершенно другая вещь.

Использование comp в строке

```
if ((*comp) (v[j], v[j + gap]) <= 0)
```

полностью согласуется с описанием: comp – указатель на функцию, *comp – сама функция, а (*comp)(v[j], v[j+gap]) – обращение к ней. Круглые скобки необходимы для правильного объединения компонентов.

Мы уже приводили функцию strcmp, сравнивающую две строки по первому численному значению.

```
numcmp(s1, s2)  /* compare s1 and s2 numerically */
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);
}
```

Заключительный шаг состоит в добавлении функции `swap`, переставляющей два указателя. Это легко сделать, непосредственно используя то, что мы изложили ранее в этой главе.

```
swap(px, py)    /* interchange *px and *py */
char *px[], *py[];
{
  char *temp;

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

Имеется множество других необязательных аргументов, которые могут быть включены в программу сортировки: некоторые из них составляют интересные упражнения.

Упражнение 5–11

Модифицируйте `sort` таким образом, чтобы она работала с меткой -r, указывающей на сортировку в обратном (убывающем) порядке. Конечно, -r должна работать с -n.

Упражнение 5–12

Добавьте необязательный аргумент -f, объединяющий вместе прописные и строчные буквы, так чтобы различие регистров не учитывалось во время сортировки: данные из верхнего и нижнего регистров сортируются вместе, так что буква ‘A’ прописное и ‘a’ строчное оказываются соседними, а не разделёнными целым алфавитом.

Упражнение 5–13

Добавьте необязательный аргумент -d («словарное упорядочивание»), при наличии которого сравниваются только буквы, числа и пробелы. Позаботьтесь о том, чтобы эта функция работала и вместе с -f.

Упражнение 5–14

Добавьте возможность обработки полей, так чтобы можно было сортировать поля внутри строк. Каждое поле должно сортироваться в соответствии с независимым набором необязательных аргументов. (предметный указатель этой книги сортировался с помощью аргументов -df для категории указателя и с -n для номеров страниц).

6 Структуры

Структура – это набор из одной или более переменных, возможно различных типов, сгруппированных под одним именем для удобства обработки.[1]

Традиционным примером структуры является учётная карточка работающего: «служащий» описывается набором атрибутов таких, как фамилия, имя, отчество (ф.и.о.), адрес, код социального обеспечения, зарплата и т.д. Некоторые из этих атрибутов сами могут оказаться структурами: ф.и.о. имеет несколько компонент, как и адрес, и даже зарплата.

Структуры оказываются полезными при организации сложных данных особенно в больших программах, поскольку во многих ситуациях они позволяют сгруппировать связанные данные таким образом, что с ними можно обращаться, как с одним целым, а не как с отдельными объектами. В этой главе мы постараемся продемонстрировать то, как используются структуры. Программы, которые мы для этого будем использовать, больше, чем многие другие в этой книге, но все же достаточно умеренных размеров.

6.1 Основные сведения

Давайте снова обратимся к процедурам преобразования даты из главы 5. Дата состоит из нескольких частей таких, как день, месяц, и год, и, возможно, день года и имя месяца. Эти пять переменных можно объединить в одну структуру вида:

```
struct date {
  int day;
  int month;
  int year;
  int yearday;
  char mon_name[4];
};
```

Описание структуры, состоящее из заключённого в фигурные скобки списка описаний, начинается с ключевого слова `struct`. За словом `struct` может следовать необязательное имя, называемое ярлыком структуры (здесь это `date`). Такой ярлык именует структуры этого вида и может использоваться в дальнейшем как сокращённая запись подробного описания.

Элементы или переменные, упомянутые в структуре, называются членами. Ярлыки и члены структур могут иметь такие же имена, что и обычные переменные (т.е. не явля-

[1] В некоторых языках, самый известный из которых паскаль, структуры называются «записями».

ющиеся членами структур), поскольку их имена всегда можно различить по контексту. Конечно, обычно одинаковые имена присваивают только тесно связанным объектам.

Точно так же, как в случае любого другого базисного типа, за правой фигурной скобкой, закрывающей список членов, может следовать список переменных. Оператор

```
struct {...} x, y, z;
```

синтаксически аналогичен

```
int x, y, z;
```

в том смысле, что каждый из операторов описывает x, y и z в качестве переменных соответствующих типов и приводит к выделению для них памяти.

Описание структуры, за которым не следует списка переменных, не приводит к выделению какой-либо памяти; оно только определяет шаблон или форму структуры. Однако, если такое описание снабжено ярлыком, то этот ярлык может быть использован позднее при определении фактических экземпляров структур. Например, если дано приведённое выше описание `date`, то

```
struct date d;
```

определяет переменную d в качестве структуры типа `date`. Внешнюю или статическую структуру можно инициализировать, поместив вслед за её определением список инициализаторов для её компонент:

```
struct date d = { 4, 7, 1776, 186, "jul" };
```

Член определённой структуры может быть указан в выражении с помощью конструкции вида

```
имя структуры.член
```

Операция указания члена структуры « . » связывает имя структуры и имя члена. В качестве примера определим `leap` (признак високосности года) на основе даты, находящейся в структуре d

```
leap = (d.year % 4 == 0 && sd.year % 100 != 0 || d.year % 400 == 0);
```

или проверим имя месяца

```
if (strcmp(d.mon_name, "aug") == 0)
  ...
```

Или преобразуем первый символ имени месяца так, чтобы оно начиналось со строчной буквы

```
d.mon_name[0] = lower(d.mon_name[0]);
```

Структуры могут быть вложенными; учётная карточка служащего может фактически выглядеть так:

```
struct person {
  char name[namesize];
  char address[adrsize];
  long zipcode;                 /* почтовый индекс */
  long ss_number;               /* код соц. обеспечения */
  double salary;                /* зарплата */
  struct date birthdate;        /* дата рождения */
  struct date hiredate;         /* дата поступления на работу */
};
```

Структура `person` содержит две структуры типа `date`. Если мы определим `emp` как

```
struct person emp;
```

то

```
emp.birthdate.month
```

будет ссылаться на месяц рождения. Операция указания члена структуры « . » ассоциируется слева направо.

6.2 Структуры и функции

В языке **C** существует ряд ограничений на использование структур. Обязательные правила заключаются в том, что единственные операции, которые вы можете проводить со структурами, состоят в определении её адреса с помощью операции & и доступе к одному из её членов. Это влечёт за собой то, что структуры нельзя присваивать или копировать как целое, и что они не могут быть переданы функциям или возвращены ими.[1] На указатели структур эти ограничения однако не накладываются, так что структуры и функции все же могут с удобством работать совместно. И наконец, автоматические структуры, как и автоматические массивы, не могут быть инициализированы; инициализация возможна только в случае внешних или статических структур.

Давайте разберём некоторые из этих вопросов, переписав с этой целью функции преобразования даты из предыдущей главы так, чтобы они использовали структуры. Так как правила запрещают непосредственную передачу структуры функции, то мы должны либо передавать отдельно компоненты, либо передать указатель всей структуры. Первая возможность демонстрируется на примере функции `day_of_year`, как мы её написали в главе 5:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

другой способ состоит в передаче указателя. Если мы опишем `hiredate` как

```
struct date hiredate;
```

и перепишем `day_of_year` нужным образом, мы сможем тогда написать

[1] В последующих версиях эти ограничения сняты.

```
hiredate yearday = day_of_year(&hiredate);
```

передавая указатель на `hiredate` функции `day_of_year`. Функция должна быть модифицирована, потому что её аргумент теперь является указателем, а не списком переменных.

```
day_of_year(pd) /* set day of year from month, day */
struct date *pd;
{
  int i, day, leap;

  day = pd->day;
  leap = (d.year % 4 == 0 && sd.year % 100 != 0 || d.year % 400 == 0);
  for (i = 1; i < pd->month; i++)
    day += day_tab[leap][i];
  return (day);
}
```

Описание

```
struct date *pd;
```

говорит, что `pd` является указателем структуры типа `date`. Запись, показанная на примере

```
pd->year
```

является новой. Если p – указатель на структуру, то `p->член структуры` – обращается к конкретному члену.[1]

Так как `pd` указывает на структуру, то к члену `year` можно обратиться и следующим образом

```
(*pd).year
```

но указатели структур используются настолько часто, что запись `->` оказывается удобным сокращением. Круглые скобки в `(*pd).year` необходимы, потому что операция указания члена структуры старше, чем $*$. Обе операции, `->` и « . », ассоциируются слева направо, так что конструкции слева и справа эквивалентны

```
p->q->memb                  (p->q)->memb
emp.birthdate.month         (emp.birthdate).month
```

Для полноты ниже приводится другая функция, `month_day`, переписанная с использованием структур.

[1] Операция -> – это знак минус, за которым следует знак больше.

```
month_day(pd)   /* set month and day from day of year */
struct date *pd;
{
  int i, leap;

  leap = (d.year % 4 == 0 && sd.year % 100 != 0 || d.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 работы со структурами -> и « . » наряду со () для списка аргументов и [] для индексов находятся на самом верху иерархии старшинства операций и, следовательно, связываются очень крепко. Если, например, имеется описание

```
struct {
  int x;
  int *y;
} *p;
```

то выражение

```
++ p->x
```

увеличивает x, а не p, так как оно эквивалентно выражению ++(p->x). Для изменения порядка выполнения операций можно использовать круглые скобки: (++p)->x увеличивает p до доступа к x, а (p++)->x увеличивает p после.[1]

Совершенно аналогично *p->y извлекает то, на что указывает y; *p->y++ увеличивает y после обработки того, на что он указывает (точно так же, как и *s++); (*p->y)++ увеличивает то, на что указывает y; *p++->y увеличивает p после выборки того, на что указывает y.

6.3 Массивы структур

Структуры особенно подходят для управления массивами связанных переменных. Рассмотрим, например, программу подсчёта числа вхождений каждого ключевого слова языка **C**. Нам нужен массив символьных строк для хранения имён и массив целых для подсчёта. Одна из возможностей состоит в использовании двух параллельных массивов keyword и keycount:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Но сам факт, что массивы параллельны, указывает на возможность другой организации. Каждое ключевое слово здесь по существу является парой:

[1] Круглые скобки в последнем случае необязательны. Почему?

```
char *keyword;
int keycount;
```

и, следовательно, имеется массив пар. Описание структуры

```
struct key {
  char *keyword;
  int keycount;
} keytab[NKEYS];
```

определяет массив keytab структур такого типа и отводит для них память. Каждый элемент массива является структурой. Это можно было бы записать и так:

```
struct key {
  char *keyword;
  int keycount;
};
struct key keytab[NKEYS];
```

Так как структура keytab фактически содержит постоянный набор имён, то легче всего инициализировать её один раз и для всех членов при определении. Инициализация структур вполне аналогична предыдущим инициализациям – за определением следует заключённый в фигурные скобки список инициализаторов:

```
struct key {
  char *keyword;
  int keycount;
} keytab[] = {
               "break", 0,
               "case", 0,
               "char", 0,
               "continue", 0,
               "default", 0,
               /* ... */
               "unsigned", 0,
               "while", 0
             };
```

Инициализаторы перечисляются парами соответственно членам структуры. Было бы более точно заключать в фигурные скобки инициализаторы для каждой «строки» или структуры следующим образом:

```
{ "break", 0 },
{ "case", 0 },
. . .
```

Но когда инициализаторы являются простыми переменными или символьными строками и все они присутствуют, то во внутренних фигурных скобках нет необходимости. Как

обычно, компилятор сам вычислит число элементов массива keytab, если инициализаторы присутствуют, а скобки [] оставлены пустыми.

Программа подсчёта ключевых слов начинается с определения массива keytab. Ведущая программа читает свой файл ввода, последовательно обращаясь к функции getword, которая извлекает из ввода по одному слову за обращение. Каждое слово ищется в массиве keytab с помощью варианта функции бинарного поиска, написанной нами в главе 3. (Конечно, чтобы эта функция работала, список ключевых слов должен быть расположен в порядке возрастания).

```
#define MAXWORD 20

main()
{       /* count "C" keywords */
  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)    /* find word in 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);
}
```

Мы вскоре приведём функцию getword; пока достаточно сказать, что она возвращает LETTER каждый раз, как она находит слово, и копирует это слово в свой первый аргумент.

Величина NKEYS – это количество ключевых слов в массиве keytab. Хотя мы можем сосчитать это число вручную, гораздо легче и надёжнее поручить это машине, особенно в том случае, если список ключевых слов подвержен изменениям. Одной из возможностей было бы закончить список инициализаторов указанием на нуль и затем пройти в цикле сквозь массив keytab, пока не найдётся конец.

Но, поскольку размер этого массива полностью определён к моменту компиляции, здесь имеется более простая возможность. Число элементов просто есть

```
sizeof(keytab) / sizeof(struct key)
```

дело в том, что в языке **C** предусмотрена унарная операция sizeof, выполняемая во время компиляции, которая позволяет вычислить размер любого объекта. Выражение

```
sizeof(object)
```

выдаёт целое, равное размеру указанного объекта.[1] Объект может быть фактической переменной, массивом и структурой, или именем основного типа, как int или double, или именем производного типа, как структура. В нашем случае число ключевых слов равно размеру массива, делённому на размер одного элемента массива. Это вычисление используется в утверждении #define для установления значения NKEYS:

```
#define NKEYS (sizeof(keytab)/sizeof(struct key))
```

Теперь перейдём к функции getword. Мы фактически написали более общий вариант функции getword, чем необходимо для этой программы, но он не на много более сложен. Функция getword возвращает следующее «слово» из ввода, где словом считается либо строка букв и цифр, начинающихся с буквы, либо отдельный символ. Тип объекта возвращается в качестве значения функции; это – LETTER, если найдено слово, EOF для конца файла и сам символ, если он не буквенный.

```
getword(w, lim) /* get next word from input */
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());
```

[1] Размер определяется в неспецифицированных единицах, называемых «байтами», которые имеют тот же размер, что и переменные типа char.

```
    if (t != LETTER && t != DIGIT) {
      ungetch(c);
      break;
    }
  }
  *(w - 1) = '\0';
  return (LETTER);
}
```

Функция getword использует функции getch и ungetch, которые мы написали в главе 4: когда набор алфавитных символов прерывается, функция getword получает один лишний символ. В результате вызова ungetch этот символ помещается назад во ввод для следующего обращения.

Функция getword обращается к функции type для определения типа каждого отдельного символа из файла ввода. Вот вариант, справедливый только для алфавита ASCII.

```
type(c) /* return type of ascii character */
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, лишь бы они не вступали в конфликт с символами, отличными от буквенно-цифровых, и с EOF; очевидно возможен следующий выбор

```
#define LETTER 'A'
#define DIGIT  '0'
```

функция getword могла бы работать быстрее, если бы обращения к функции type были заменены обращениями к соответствующему массиву type[]. В стандартной библиотеке языка **C** предусмотрены макросы ISALPHA и ISDIGIT, действующие необходимым образом.

Упражнение 6–1

Сделайте такую модификацию функции getword и оцените, как изменится скорость работы программы.

Упражнение 6–2

Напишите вариант функции type, не зависящий от конкретного набора символов.

Упражнение 6–3

Напишите вариант программы подсчёта ключевых слов, который бы не учитывал появления этих слов в заключённых в кавычки строках.

6.4 Указатели на структуры

Чтобы проиллюстрировать некоторые соображения, связанные с использованием указателей и массивов структур, давайте снова составим программу подсчёта ключевых строк, используя на этот раз указатели, а не индексы массивов.

Внешнее описание массива `keytab` не нужно изменять, но функции `main` и `binary` требуют модификации.

```
main()
{       /* count "C" keywords; pointer version */
  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)        /* find word */
char *word      /* in 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 имеется несколько моментов, которые стоит отметить. Во-первых, описание функции `binary` должно указывать, что она возвращает указатель на структуру типа `key`, а не на целое; это объявляется как в функции `main`, так и в `binary`. Если функция `binary` находит слово, то она возвращает указатель на него; если же нет, она возвращает `NULL`.

Во-вторых, все обращения к элементам массива `keytab` осуществляются через указатели. Это влечёт за собой одно существенное изменение в функции `binary`: средний элемент больше нельзя вычислять просто по формуле

```
mid = (low + high) / 2;
```

потому что сложение двух указателей не даёт какого-нибудь полезного результата (даже после деления на 2) и в действительности является незаконным. Эту формулу надо заменить на

```
mid = low + (high - low) / 2;
```

в результате которой `mid` становится указателем на элемент, расположенный посередине между `low` и `high`.

Вам также следует разобраться в инициализации `low` и `high`. Указатель можно инициализировать адресом ранее определённого объекта; именно как мы здесь и поступили.

В функции `main` мы написали

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Если p является указателем структуры, то любая арифметика с p учитывает фактический размер данной структуры, так что p++ увеличивает p на нужную величину, в результате чего p указывает на следующий элемент массива структур. Но не считайте, что размер структуры равен сумме размеров её членов, – из-за требований выравнивания для различных объектов в структуре могут возникать «дыры».

И, наконец, несколько второстепенный вопрос о форме записи программы. Если возвращаемая функцией величина имеет тип, как, например, в

```
struct key *binary(word, tab, n)
```

То может оказаться, что имя функции трудно выделить среди текста. В связи с этим иногда используется другой стиль записи:

```
struct key *
binary(word, tab, n)
```

Это главным образом дело вкуса; выберите ту форму, которая вам нравится, и придерживайтесь её.

6.5 Структуры, ссылающиеся на себя

Предположим, что нам надо справиться с более общей задачей, состоящей в подсчёте числа появлений всех слов в некотором файле ввода. Так как список слов заранее не известен, мы не можем их упорядочить удобным образом и использовать бинарный поиск. Мы даже не можем осуществлять последовательный просмотр при поступлении каждого слова, с тем чтобы установить, не встречалось ли оно ранее; такая программа будет работать вечно. (Более точно, ожидаемое время работы растёт как квадрат числа вводимых слов). Как же нам организовать программу, чтобы справиться со списком произвольных слов?

Одно из решений состоит в том, чтобы все время хранить массив поступающих до сих пор слов в упорядоченном виде, помещая каждое слово в нужное место по мере их поступления. Однако это не следует делать, перемещая слова в линейном массиве, – это также потребует слишком много времени. Вместо этого мы используем структуру данных, называемую двоичным деревом.

Каждому новому слову соответствует один «узел» дерева; каждый узел содержит:

- указатель текста слова;
- счётчик числа появлений;
- указатель узла левого потомка;
- указатель узла правого потомка.

Никакой узел не может иметь более двух детей; возможно отсутствие детей или наличие только одного потомка.

Узлы создаются таким образом, что левое поддерево каждого узла содержит только те слова, которые меньше слова в этом узле, а правое поддерево только те слова, которые больше. Чтобы определить, находится ли новое слово уже в дереве, начинают с корня и сравнивают новое слово со словом, хранящимся в этом узле. Если слова совпадают, то вопрос решается утвердительно. Если новое слово меньше слова в дереве, то переходят к рассмотрению левого потомка; в противном случае исследуется правый потомок. Если в нужном направлении потомок отсутствует, то значит новое слово не находится в дереве и место этого недостающего потомка как раз и является местом, куда следует поместить новое слово. Поскольку поиск из любого узла приводит к поиску одного из его потомков, то сам процесс поиска по существу является рекурсивным. В соответствии с этим наиболее естественно использовать рекурсивные процедуры ввода и вывода.

Возвращаясь назад к описанию узла, ясно, что это будет структура с четырьмя компонентами:

```
struct tnode {  /* the basic node */
  char *word;   /* points to the text */
  int count;    /* number of occurrences */
  struct tnode *left;   /* left child */
  struct tnode *right;  /* right child */
};
```

Это «рекурсивное» описание узла может показаться рискованным, но на самом деле оно вполне корректно. Структура не имеет права содержать ссылку на саму себя, но

```
struct tnode *left;
```

описывает `left` как указатель на узел, а не как сам узел.

Текст самой программы оказывается удивительно маленьким, если, конечно, иметь в распоряжении набор написанных нами ранее процедур, обеспечивающих нужные действия. Мы имеем в виду функцию `getword` для извлечения каждого слова из файла ввода и функцию `alloc` для выделения места для хранения слов.

Ведущая программа просто считывает слова с помощью функции `getword` и помещает их в дерево, используя функцию `tree`.

```
#define MAXWORD 20
main()
{       /* word freguency count */
  struct tnode *root, *tree();
  char word[MAXWORD];
  int t;

  root = NULL;
  while ((t = getword(word, MAXWORD)) != EOF)
    if (type(t) == LETTER)
      root = tree(root, word);

  treeprint(root);
}
```

Функция `tree` сама по себе проста. Слово передаётся функцией `main` к верхнему уровню (корню) дерева. На каждом этапе это слово сравнивается со словом, уже хранящимся в этом узле, и с помощью рекурсивного обращения к `tree` просачивается вниз либо к левому, либо к правому поддереву. В конце концов это слово либо совпадает с каким-то словом, уже находящимся в дереве (в этом случае счётчик увеличивается на единицу), либо программа натолкнётся на нулевой указатель, свидетельствующий о необходимости создания и добавления к дереву нового узла. В случае создания нового узла функция `tree` возвращает указатель этого узла, который помещается в родительский узел.

```
struct tnode *tree(p, w)        /* install w at or below p */
struct tnode *p;
char *w;
{
  struct tnode *talloc();
  char *strsave();
  int cond;

  if (p == NULL) {      /* a new word has arrived */
```

```
    p == talloc();      /* make a new node */
    p->word = strsave(w);
    p->count = 1;
    p->left = p->right = NULL;
  } else if ((cond = strcmp(w, p->word)) == 0)  /* repeated word */
    p->count++;
  else if (cond < 0)    /* lower goes into left subtree */
    p->left = tree(p->left, w);
  else  /* greater into right subtree */
    p->right = tree(p->right, w);

  return (p);
}
```

Память для нового узла выделяется функцией `talloc`, являющейся адаптацией для данного случая функции `alloc`, написанной нами ранее. Она возвращает указатель свободного пространства, пригодного для хранения нового узла дерева. (Мы вскоре обсудим это подробнее). Новое слово копируется функцией `strsave` в скрытое место, счётчик инициализируется единицей, и указатели обоих потомков полагаются равными нулю. Эта часть программы выполняется только при добавлении нового узла к ребру дерева. Мы здесь опустили проверку на ошибки возвращаемых функций `strsave` и `talloc` значений (что неразумно для практически работающей программы).

Функция `treeprint` печатает дерево, начиная с левого поддерева; в каждом узле сначала печатается левое поддерево (все слова, которые младше этого слова), затем само слово, а затем правое поддерево (все слова, которые старше). Если вы неуверенно оперируете с рекурсией, нарисуйте дерево сами и напечатайте его с помощью функции `treeprint`; это одна из наиболее ясных рекурсивных процедур, которую можно найти.

```
treeprint(p)    /* print tree p recursively */
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 поступают не в случайном порядке, то время работы программы может расти слишком быстро. В худшем случае, когда поступающие слова уже упорядочены, настоящая программа осуществляет дорогостоящую имитацию линейного поиска. Существуют различные обобщения двоичного дерева, особенно 2-3 деревья и AVL деревья, которые не ведут себя так «в худших случаях», но мы не будем здесь на них останавливаться.

Прежде чем расстаться с этим примером, уместно сделать небольшое отступление в связи с вопросом о распределении памяти. Ясно, что в программе желательно иметь

только один распределитель памяти, даже если ему приходится размещать различные виды объектов. Но если мы хотим использовать один распределитель памяти для обработки запросов на выделение памяти для указателей на переменные типа `char` и для указателей на `struct tnode`, то при этом возникают два вопроса. Первый: как выполнить то существующее на большинстве реальных машин ограничение, что объекты определённых типов должны удовлетворять требованиям выравнивания. Например, часто целые должны размещаться в чётных адресах? Второй: как организовать описания, чтобы справиться с тем, что функция `alloc` должна возвращать различные виды указателей?

Вообще говоря, требования выравнивания легко выполнить за счёт выделения некоторого лишнего пространства, просто обеспечив то, чтобы распределитель памяти всегда возвращал указатель, удовлетворяющий всем ограничениям выравнивания. Например, на PDP-11 достаточно, чтобы функция `alloc` всегда возвращала чётный указатель, поскольку в чётный адрес можно поместить любой тип объекта. Единственный расход при этом лишний символ при запросе на нечётную длину. Аналогичные действия предпринимаются на других машинах. Таким образом, реализация `alloc` может не оказаться переносимой, но её использование будет переносимым. Функция `alloc` из главы 5 не предусматривает никакого определённого выравнивания; в главе 8 мы продемонстрируем, как правильно выполнить эту задачу.

Вопрос описания типа функции `alloc` является мучительным для любого языка, который серьёзно относится к проверке типов. Лучший способ в языке **C** – объявить, что `alloc` возвращает указатель на переменную типа `char`, а затем явно преобразовать этот указатель к желаемому типу с помощью операции перевода типов. Таким образом, если описать p в виде

```
char *p;
```

то

```
(struct tnode *) p
```

преобразует его в выражениях в указатель на структуру типа `tnode`. Следовательно, функцию `talloc` можно записать в виде:

```
struct tnode *talloc()
{
  char *alloc();
  return ((struct tnode *) alloc(sizeof(struct tnode)));
}
```

это более чем достаточно для работающих в настоящее время компиляторов, но это и самый безопасный путь с учётом будущего.

Упражнение 6–4

Напишите программу, которая читает **C**-программу и печатает в алфавитном порядке каждую группу имён переменных, которые совпадают в первых семи символах, но отличаются где-то дальше. (Сделайте так, чтобы семь было параметром).

Упражнение 6–5

Напишите программу выдачи перекрёстных ссылок, т.е. программу, которая печатает список всех слов документа и для каждого из этих слов печатает список номеров строк, в которые это слово входит.

Упражнение 6–6

Напишите программу, которая печатает слова из своего файла ввода, расположенные в порядке убывания частоты их появления. Перед каждым словом напечатайте число его появлений.

6.6 Поиск в таблице

Для иллюстрации дальнейших аспектов использования структур в этом разделе мы напишем программу, представляющую собой содержимое пакета поиска в таблице. Эта программа является типичным представителем подпрограмм управления символьными таблицами макропроцессора или компилятора. Рассмотрим, например, оператор `#define` языка **C**. Когда встречается строка вида

```
#define YES 1
```

то имя YES и заменяющий текст 1 помещаются в таблицу. Позднее, когда имя YES появляется в операторе вида

```
inword = YES;
```

Оно должно быть замещено на 1.

Имеются две основные процедуры, которые управляют именами и заменяющими их текстами. Функция `instal(s,t)` записывает имя s и заменяющий текст t в таблицу; здесь s и t просто символьные строки. Функция `lookup(s)` ищет имя s в таблице и возвращает либо указатель того места, где это имя найдено, либо `NULL`, если этого имени в таблице не оказалось.

При этом используется поиск по алгоритму хеширования. Поступающее имя преобразуется в маленькое положительное число, которое затем используется для индексации массива указателей. Элемент массива указывает на начало цепочных блоков, описывающих имена, которые имеют это значение хеширования. Если никакие имена при хешировании не получают этого значения, то элементом массива будет `NULL`.

Блоком цепи является структура, содержащая указатели на соответствующее имя, на заменяющий текст и на следующий блок в цепи. Нулевой указатель следующего блока служит признаком конца данной цепи.

```
struct nlist {  /* basic table entry */
  char *name;
  char *def;
  struct nlist *next;   /* next entry in chain */
};
```

Массив указателей это просто

```
#define HASHSIZE 100
static struct nlist *hashtab[HASHSIZE]
```

Значение функции хеширования, используемой обеими функциями `lookup` и `instal`, получается просто как остаток от деления суммы символьных значений строки на размер массива. (Это не самый лучший возможный алгоритм, но его достоинство состоит в исключительной простоте).

```
hash(s) /* form hash value for string */
char *s;
{
  int hashval;

  for (hashval = 0; *s != '\0';)
    hashval += *s++;

  return (hashval % HASHSIZE);
}
```

В результате процесса хеширования выдаётся начальный индекс в массиве `hashtab`; если данная строка может быть где-то найдена, то именно в цепи блоков, начало которой указано там. Поиск осуществляется функцией `lookup`. Если функция `lookup` находит, что данный элемент уже присутствует, то она возвращает указатель на него; если нет, то она возвращает `NULL`.

```
struct nlist *lookup(s) /* look for s in hashtab */
char *s;
{
  struct nlist *np;

  for (np = hashtab[hash(s)]; np != NULL; np = np->next)
    if (strcmp(s, np->name) == 0)
      return (np);      /* found it */

  return (NULL);        /* not found */
}
```

Функция `instal` использует функцию `lookup` для определения, не присутствует ли уже вводимое в данный момент имя; если это так, то новое определение должно вытеснить старое. В противном случае создаётся совершенно новый элемент. Если по какой-либо причине для нового элемента больше нет места, то функция `instal` возвращает `NULL`.

```
struct nlist *install(name, def)        /* put (name, def) */
char *name, *def;
{
```

```
    struct nlist *np, *lookup();
    char *strsave(), *alloc();
    int hashval;

    if ((np = lookup(name)) == NULL) {    /* not found */
      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        /* already there */
      free(np->def);      /* free previous definition */

    if ((np->def = strsave(def)) == NULL)
      return (NULL);

    return (np);
  }
```

Функция `strsave` просто копирует строку, указанную в качестве аргумента, в место хранения, полученное в результате обращения к функции `alloc`. Мы уже привели эту функцию в главе 5. Так как обращение к функции `alloc` и `free` могут происходить в любом порядке и в связи с проблемой выравнивания, простой вариант функции `alloc` из главы 5 нам больше не подходит; смотрите главы 7 и 8.

Упражнение 6–7

Напишите процедуру, которая будет удалять имя и определение из таблицы, управляемой функциями `lookup` и `instal`.

Упражнение 6–8

Разработайте простую, основанную на функциях этого раздела, версию процессора для обработки конструкций `#define`, пригодную для использования с **C**-программами. Вам могут также оказаться полезными функции `getchar` и `ungetch`.

6.7 Поля

Когда вопрос экономии памяти становится очень существенным, то может оказаться необходимым помещать в одно машинное слово несколько различных объектов; одно из

особенно распространённых употреблений – набор однобитовых признаков в применениях, подобных символьным таблицам компилятора. Внешне обусловленные форматы данных, такие как интерфейсы аппаратных средств также зачастую предполагают возможность получения слова по частям.

Представьте себе фрагмент компилятора, который работает с символьной таблицей. С каждым идентификатором программы связана определённая информация, например, является он или нет ключевым словом, является ли он или нет внешним и/или статическим и т.д. Самый компактный способ закодировать такую информацию – поместить набор однобитовых признаков в отдельную переменную типа `char` или `int`.

Обычный способ, которым это делается, состоит в определении набора «масок», отвечающих соответствующим битовым позициям, как в

```
#define KEYWORD  01
#define EXTERNAL 02
#define STATIC   04
```

(числа должны быть степенями двойки). Тогда обработка битов сведётся к «жонглированию битами» с помощью операций сдвига, маскирования и дополнения, описанных нами в главе 2.

Некоторые часто встречающиеся идиомы:

```
flags |= EXTERNAL | STATIC;
```

включает биты EXTERNAL и STATIC в flags, в то время как

```
flags &= ~(EXTERNAL | STATIC);
```

их выключает, а

```
if ((flags & (EXTERNAL | STATIC)) == 0)
  ...
```

истинно, если оба бита выключены.

Хотя этими идиомами легко овладеть, язык **C** в качестве альтернативы предлагает возможность определения и обработки полей внутри слова непосредственно, а не посредством побитовых логических операций. Поле – это набор смежных битов внутри одной переменной типа `int`. Синтаксис определения и обработки полей основывается на структурах. Например, символьную таблицу конструкций `#define`, приведённую выше, можно бы было заменить определением трёх полей:

```
struct {
  unsigned is_keyword:1;
  unsigned is_extern:1;
  unsigned is_static:1;
} flags;
```

Здесь определяется переменная с именем `flags`, которая содержит три однобитовых поля. Следующее за двоеточием число задаёт ширину поля в битах. Поля описаны как `unsigned`, чтобы подчеркнуть, что они действительно будут величинами без знака.

На отдельные поля можно ссылаться, как `flags.is_static`, `flags.is_extern`, `flags.is_keyword` и т.д., то есть точно так же, как на другие члены структуры. Поля ведут себя подобно небольшим целым без знака и могут участвовать в арифметических выражениях точно так же, как и другие целые. Таким образом, предыдущие примеры более естественно переписать так:

```
flags.is_extern = flags.is_static = 1;
```

для включения битов;

```
flags.is_extern = flags.is_static = 0;
```

для выключения битов;

```
if (flags.is_extern == 0 && flags.is_static == 0)
  ...
```

для их проверки.

Поле не может перекрывать границу `int`; если указанная ширина такова, что это должно случиться, то поле выравнивается по границе следующего `int`. Полям можно не присваивать имена; неименованные поля (только двоеточие и ширина) используются для заполнения свободного места. Чтобы вынудить выравнивание на границу следующего `int`, можно использовать специальную ширину 0.

При работе с полями имеется ряд моментов, на которые следует обратить внимание. По-видимому наиболее существенным является то, что отражая природу различных аппаратных средств, распределение полей на некоторых машинах осуществляется слева направо, а на некоторых справа налево. Это означает, что хотя поля очень полезны для работы с внутренне определёнными структурами данных, при разделении внешне определяемых данных следует тщательно рассматривать вопрос о том, какой конец поступает первым.

Другие ограничения, которые следует иметь в виду: поля не имеют знака; они могут храниться только в переменных типа `int` (или, что эквивалентно, типа `unsigned`); они не являются массивами; они не имеют адресов, так что к ним не применима операция &.

6.8 Объединения

Объединения – это переменная, которая в различные моменты времени может содержать объекты разных типов и размеров, причём компилятор берет на себя отслеживание размера и требований выравнивания. Объединения представляют возможность работать с различными видами данных в одной области памяти, не вводя в программу никакой машинно-зависимой информации.

В качестве примера, снова из символьной таблицы компилятора, предположим, что константы могут быть типа `int`, `float` или быть указателями на символы. Значение каждой конкретной константы должно храниться в переменной соответствующего типа, но все же для управления таблицей самым удобным было бы, если это значение занимало бы один и тот же объем памяти и хранилось в том же самом месте независимо от его типа. Это и является назначением объединения – выделить отдельную переменную, в которой можно законно хранить любую одну из переменных нескольких типов. Как и в случае полей, синтаксис основывается на структурах.

```
union u_tag {
  int ival;
  float fval;
  char *pval;
} uval;
```

Переменная `uval` будет иметь достаточно большой размер,чтобы хранить наибольший из трёх типов, независимо от машины, на которой осуществляется компиляция, – программа не будет зависеть от характеристик аппаратных средств. Любой из этих трёх типов может быть присвоен `uval` и затем использован в выражениях, пока такое использование совместимо: извлекаемый тип должен совпадать с последним помещённым типом. Дело программиста – следить за тем, какой тип хранится в объединении в данный момент; если что-либо хранится как один тип, а извлекается как другой, то результаты будут зависеть от используемой машины.

Синтаксически доступ к членам объединения осуществляется следующим образом:

```
имя объединения.член
```

или

```
указатель объединения->член
```

то есть точно так же, как и в случае структур. Если для отслеживания типа, хранимого в данный момент в `uval`, используется переменная `utype`, то можно встретить такой участок программы:

```
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);
```

Объединения могут появляться внутри структур и массивов и наоборот. Запись для обращения к члену объединения в структуре (или наоборот) совершенно идентична той, которая используется во вложенных структурах. Например, в массиве структур, определённым следующим образом

```
struct {
  char *name;
  int flags;
  int utype;
  union {
    int ival;
    float fval;
    char *pval;
  } uval;
} symtab[nsym];
```

на переменную `ival` можно сослаться как

```
symtab[i].uval.ival
```

а на первый символ строки `pval` как

```
* symtab[i].uval.pval
```

В сущности объединение является структурой, в которой все члены имеют нулевое смещение. Сама структура достаточно велика, чтобы хранить «самый широкий» член, и выравнивание пригодно для всех типов, входящих в объединение. Как и в случае структур, единственными операциями, которые в настоящее время можно проводить с объединениями, являются доступ к члену и извлечение адреса; объединения не могут быть присвоены, переданы функциям или возвращены ими. Указатели объединений можно использовать в точно такой же манере, как и указатели структур.

Программа распределения памяти, приводимая в главе 8, показывает, как можно использовать объединение, чтобы сделать некоторую переменную выровненной по определённому виду границы памяти.

6.9 Определение типа

В языке **C** предусмотрена возможность, называемая `typedef` для введения новых имён для типов данных. Например, описание

```
typedef int length;
```

делает имя `length` синонимом для `int`. «Тип» `length` может быть использован в описаниях, переводов типов и т.д. Точно таким же образом, как и тип `int`:

```
length len, maxlen;
length *lengths[];
```

Аналогично описанию

```
typedef char *string;
```

делает `string` синонимом для `char*`, то есть для указателя на символы, что затем можно использовать в описаниях вида

```
string p, lineptr[lines], alloc();
```

Обратите внимание, что объявляемый в конструкции typedef тип появляется в позиции имени переменной, а не сразу за словом typedef. Синтаксически конструкция typedef подобна описаниям класса памяти extern, static и т.д. Мы также использовали прописные буквы, чтобы яснее выделить имена.

В качестве более сложного примера мы используем конструкцию typedef для описания узлов дерева, рассмотренных ранее в этой главе:

```
typedef struct tnode {  /* the basic node */
  char *word;           /* points to the text */
  int count;            /* number of occurrences */
  struct tnode *left;   /* left child */
  struct tnode *right;  /* right child */
} treenode, *treeptr;
```

В результате получаем два новых ключевых слова: treenode (структура) и treeptr (указатель на структуру). Тогда функцию talloc можно записать в виде

```
treeptr talloc()
{
  char *alloc();
  return ((treeptr) alloc(sizeof(treenode)));
}
```

Необходимо подчеркнуть, что описание typedef не приводит к созданию нового в каком-либо смысле типа; оно только добавляет новое имя для некоторого существующего типа. При этом не возникает и никакой новой семантики: описанные таким способом переменные обладают точно теми же свойствами, что и переменные, описанные явным образом. По существу конструкция typedef сходна с #define за исключением того, что она интерпретируется компилятором и потому может осуществлять подстановки текста, которые выходят за пределы возможностей макропроцессора языка **C**. Например,

```
typedef int (*pfi) ();
```

создаёт тип pfi для «указателя функции, возвращающей значение типа int», который затем можно было бы использовать в программе сортировки из главы 5 в контексте вида

```
pfi strcmp, numcmp, swap;
```

Имеются две основные причины применения описаний typedef. Первая причина связана с параметризацией программы, чтобы облегчить решение проблемы переносимости. Если для типов данных, которые могут быть машинно-зависимыми, использовать описание typedef, то при переносе программы на другую машину придётся изменить только эти описания. Одна из типичных ситуаций состоит в использовании определяемых с помощью typedef имён для различных целых величин и в последующем подходящем выборе типов short, int и long для каждой имеющейся машины. Второе назначение typedef состоит в обеспечении лучшей документации для программы – тип с именем treeptr

может оказаться более удобным для восприятия, чем тип, который описан только как указатель сложной структуры. И наконец, всегда существует вероятность, что в будущем компилятор или некоторая другая программа, такая как `lint`, сможет использовать содержащуюся в описаниях `typedef` информацию для проведения некоторой дополнительной проверки программы.

7 Ввод и вывод

Средства ввода-вывода не являются составной частью языка **C**, так что мы не выделяли их в нашем предыдущем изложении. Однако реальные программы взаимодействуют со своей окружающей средой гораздо более сложным образом, чем мы видели до сих пор. В этой главе будет описана «стандартная библиотека ввода-вывода», то есть набор функций, разработанных для обеспечения стандартной системы ввода-вывода для **C**–программ. Эти функции предназначены для удобства программного интерфейса, и все же отражают только те операции, которые могут быть обеспечены на большинстве современных операционных систем. Процедуры достаточно эффективны для того, чтобы пользователи редко чувствовали необходимость обойти их «ради эффективности», как бы ни была важна конкретная задача. И, наконец, эти процедуры задуманы быть «переносимыми» в том смысле, что они должны существовать в совместимом виде на любой системе, где имеется язык **C**, и что программы, которые ограничивают свои взаимодействия с системой возможностями, предоставляемыми стандартной библиотекой, можно будет переносить с одной системы на другую по существу без изменений.

Мы здесь не будем пытаться описать всю библиотеку ввода-вывода; мы более заинтересованы в том, чтобы продемонстрировать сущность написания **C**-программ, которые взаимодействуют со своей операционной средой.

7.1 Обращение к стандартной библиотеке

Каждый исходный файл, который обращается к функции из стандартной библиотеки, должен вблизи начала содержать строку

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

в файле stdio.h определяются некоторые макросы и переменные, используемые библиотекой ввода-вывода. Использование угловых скобок вместо обычных двойных кавычек – указание компилятору искать этот файл в справочнике, содержащем заголовки стандартной информации.[1]

Кроме того, при загрузке программы может оказаться необходимым указать библиотеку явно; на системе PDP-11 UNIX, например, команда компиляции программы имела бы вид:

```
cc   исходные файлы и т.д. -ls
```

где `-ls` указывает на загрузку из стандартной библиотеки.

[1] На системе UNIX обычно `/usr/include`.

7.2 Стандартный ввод и вывод – функции getchar и putchar

Самый простой механизм ввода заключается в чтении по одному символу за раз из «стандартного ввода», обычно с терминала пользователя, с помощью функции `getchar`. Функция `getchar()` при каждом к ней обращении возвращает следующий вводимый символ. В большинстве сред, которые поддерживают язык **C**, терминал может быть заменён некоторым файлом с помощью обозначения « < » : если некоторая программа `prog` использует функцию `getchar` то командная строка

```
prog < infile
```

приведёт к тому, что `prog` будет читать из файла `infile`, а не с терминала. Переключение ввода делается таким образом, что сама программа `prog` не замечает изменения; в частности строка « `< infile` » не включается в командную строку аргументов в `argv`. Переключение ввода оказывается незаметным и в том случае, когда вывод поступает из другой программы посредством поточного (pipe) механизма; командная строка

```
otherprog | prog
```

прогоняет две программы, `otherprog` и `prog`, и организует так, что стандартным вводом для `prog` служит стандартный вывод `otherprog`.

Функция `getchar` возвращает значение `EOF`, когда она попадает на конец файла, какой бы ввод она при этом не считывала. Стандартная библиотека полагает символическую константу `EOF` равной -1 (посредством `#define` в файле stdio.h), но проверки следует писать в терминах `EOF`, а не -1, чтобы избежать зависимости от конкретного значения.

Вывод можно осуществлять с помощью функции `putchar(c)`, помещающей символ «c» в «стандартный вывод», который по умолчанию является терминалом. Вывод можно направить в некоторый файл с помощью обозначения >: если `prog` использует `putchar`, то командная строка

```
prog > outfile
```

приведёт к записи стандартного вывода в файл `outfile`, а не на терминал. На системе UNIX можно также использовать поточный механизм. Строка

```
prog | anotherprog
```

помещает стандартный вывод `prog` в стандартный ввод `anotherprog`. И опять `prog` не будет осведомлена об изменении направления.

Вывод, осуществляемый функцией `printf`, также поступает в стандартный вывод, и обращения к `putchar` и `printf` могут перемежаться.

Поразительное количество программ читает только из одного входного потока и пишет только в один выходной поток; для таких программ ввод и вывод с помощью функций `getchar`, `putchar` и `printf` может оказаться вполне адекватным и для начала определённо достаточным. Это особенно справедливо тогда, когда имеется возможность указания файлов для ввода и вывода и поточный механизм для связи вывода одной программы с вводом другой. Рассмотрим, например, программу `lower`, которая преобразует прописные буквы из своего ввода в строчные:

```
#include <stdio.h>

main()
{       /* convert input to lower case */
  int c;

  while ((c = getchar()) != EOF)
    putchar(isupper(c) ? tolower(c) : c);
}
```

«Функции» `isupper` и `tolower` на самом деле являются макросами, определёнными в stdio.h. Макрос `isupper` проверяет, является ли его аргумент буквой из верхнего регистра, и возвращает ненулевое значение, если это так, и нуль в противном случае. Макрос `tolower` преобразует букву из верхнего регистра в ту же букву нижнего регистра. Независимо от того, как эти функции реализованы на конкретной машине, их внешнее поведение совершенно одинаково, так что использующие их программы избавлены от знания символьного набора.

Если требуется преобразовать несколько файлов, то можно собрать эти файлы с помощью программы, подобной утилите `cat` системы UNIX,

```
cat file1 file2 ... | lower > output
```

и избежать тем самым вопроса о том, как обратиться к этим файлам из программы.[1]

Кроме того отметим, что в стандартной библиотеке ввода-вывода «функции» `getchar` и `putchar` на самом деле могут быть макросами. Это позволяет избежать накладных расходов на обращение к функции для обработки каждого символа. В главе 8 мы продемонстрируем, как это делается.

7.3 Форматный вывод – функция printf

Две функции: `printf` для вывода и `scanf` для ввода (следующий раздел) позволяют преобразовывать численные величины в символьное представление и обратно. Они также позволяют генерировать и интерпретировать форматные строки. Мы уже всюду в предыдущих главах неформально использовали функцию `printf`; здесь приводится более полное и точное описание. Функция

```
printf(control, arg1, arg2, ...)
```

преобразует, определяет формат и печатает свои аргументы в стандартный вывод под управлением строки `control`. Управляющая строка содержит два типа объектов: обычные символы, которые просто копируются в выходной поток, и спецификации преобразований, каждая из которых вызывает преобразование и печать очередного аргумента `printf`.

Каждая спецификация преобразования начинается с символа % и заканчивается символом преобразования. Между % и символом преобразования могут находиться:

[1] Программа `cat` приводится позже в этой главе.

- знак минус, который указывает о выравнивании преобразованного аргумента по левому краю его поля.
- Строка цифр, задающая минимальную ширину поля. Преобразованное число будет напечатано в поле по крайней мере этой ширины, а если необходимо, то и в более широком. Если преобразованный аргумент имеет меньше символов, чем указанная ширина поля, то он будет дополнен слева (или справа, если было указано выравнивание по левому краю) заполняющими символами до этой ширины. Заполняющим символом обычно является пробел, а если ширина поля указывается с лидирующим нулём, то этим символом будет нуль.[1]
- Точка, которая отделяет ширину поля от следующей строки цифр.
- Строка цифр (точность), которая указывает максимальное число символов строки, которые должны быть напечатаны, или число печатаемых справа от десятичной точки цифр для переменных типа `float` или `double`.
- Модификатор длины `l`, который указывает, что соответствующий элемент данных имеет тип `long`, а не `int`.

Ниже приводятся символы преобразования и их смысл:

d – аргумент преобразуется к десятичному виду.

o – Аргумент преобразуется в беззнаковую восьмеричную форму (без лидирующего нуля).

x – Аргумент преобразуется в беззнаковую шестнадцатеричную форму (без лидирующих `0x`).

u – Аргумент преобразуется в беззнаковую десятичную форму.

c – Аргумент рассматривается как отдельный символ.

s – Аргумент является строкой: символы строки печатаются до тех пор, пока не будет достигнут нулевой символ или не будет напечатано количество символов, указанное в спецификации точности.

e – Аргумент, рассматриваемый как переменная типа `float` или `double`, преобразуется в десятичную форму в виде [-]m.nnnnnne±xx, где длина строки из n определяется указанной точностью. Точность по умолчанию равна 6.

f – Аргумент, рассматриваемый как переменная типа `float` или `double`, преобразуется в десятичную форму в виде [-]mmm.nnnnn, где длина строки из n определяется указанной точностью. Точность по умолчанию равна 6. Отметим, что эта точность не определяет количество печатаемых в формате f значащих цифр.

[1] Лидирующий нуль в данном случае не означает восьмеричной ширины поля.

g – Используется или формат %e или %f, какой короче; незначащие нули не печатаются.

% – Если идущий за % символ не является символом преобразования, то печатается сам этот символ; следовательно,символ % можно напечатать, указав %%.

Большинство из форматных преобразований очевидно и было проиллюстрировано в предыдущих главах. Единственным исключением является то, как точность взаимодействует со строками. Следующая таблица демонстрирует влияние задания различных спецификаций на печать "HELLO, WORLD" (12 символов). Мы поместили двоеточия вокруг каждого поля для того, чтобы вы могли видеть его протяжённость.

```
:%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 свой первый аргумент для определения числа последующих аргументов и их типов. Если количество аргументов окажется недостаточным или они будут иметь несоответственные типы, то возникнет путаница и вы получите бессмысленные результаты.

Упражнение 7–1

Напишите программу, которая будет печатать разумным образом произвольный ввод. Как минимум она должна печатать неграфические символы в восьмеричном или шестнадцатеричном виде (в соответствии с принятыми у вас обычаями) и складывать длинные строки.

7.4 Форматный ввод – функция scanf

Осуществляющая ввод функция `scanf` является аналогом `printf` и позволяет проводить в обратном направлении многие из тех же самых преобразований. Функция

```
scanf(control, arg1, arg2, ...)
```

читает символы из стандартного ввода, интерпретирует их в соответствии с форматом, указанном в аргументе `control`, и помещает результаты в остальные аргументы. Управляющий аргумент описывается ниже; другие аргументы, каждый из которых должен быть указателем, определяют, куда следует поместить соответствующим образом преобразованный ввод.

Управляющая строка обычно содержит спецификации преобразования, которые используются для непосредственной интерпретации входных последовательностей. Управляющая строка может содержать:

- пробелы, табуляции или символы новой строки («символы пустых промежутков»), которые игнорируются.
- Обычные символы (не %), которые предполагаются совпадающими со следующими отличными от символов пустых промежутков символами входного потока.
- Спецификации преобразования, состоящие из символа %, необязательного символа подавления присваивания *, необязательного числа, задающего максимальную ширину поля и символа преобразования.

Спецификация преобразования управляет преобразованием следующего поля ввода. Нормально результат помещается в переменную, которая указывается соответствующим аргументом. Если, однако, с помощью символа *указано подавление присваивания, то это поле ввода просто пропускается и никакого присваивания не производится. Поле ввода определяется как строка символов, которые отличны от символов простых промежутков; оно продолжается либо до следующего символа пустого промежутка, либо пока не будет исчерпана ширина поля, если она указана. Отсюда следует, что при поиске нужного ей ввода, функция `scanf` будет пересекать границы строк, поскольку символ новой строки входит в число пустых промежутков.

Символ преобразования определяет интерпретацию поля ввода; согласно требованиям основанной на вызове по значению семантики языка **C** соответствующий аргумент должен быть указателем. Допускаются следующие символы преобразования:

d – на вводе ожидается десятичное целое; соответствующий аргумент должен быть указателем на целое.

o – На вводе ожидается восьмеричное целое (с лидирующим нулём или без него); соответствующий аргумент должен быть указателем на целое.

x – На вводе ожидается шестнадцатеричное целое (с лидирующими `0x` или без них); соответствующий аргумент должен быть указателем на целое.

h – На вводе ожидается целое типа `short`; соответствующий аргумент должен быть указателем на целое типа `short`.

c – Ожидается отдельный символ; соответствующий аргумент должен быть указателем на символы; следующий вводимый символ помещается в указанное место. Обычный пропуск символов пустых промежутков в этом случае подавляется; для чтения следующего символа, который не является символом пустого промежутка, пользуйтесь спецификацией преобразования `%1s`.

s – Ожидается символьная строка; соответствующий аргумент должен быть указателем символов, который указывает на массив символов, который достаточно велик для принятия строки и добавляемого в конце символа \0.

f – Ожидается число с плавающей точкой; соответствующий аргумент должен быть указателем на переменную типа `float`.

e – Символ преобразования e является синонимом для f. Формат ввода переменной типа `float` включает необязательный знак, строку цифр, возможно содержащую десятичную точку и необязательное поле экспоненты, состоящее из буквы e, за которой следует целое, возможно имеющее знак.

Перед символами преобразования d, o и x может стоять `l`, которая означает, что в списке аргументов должен находиться указатель на переменную типа `long`, а не типа `int`. Аналогично, буква `l` может стоять перед символами преобразования e или f, говоря о том, что в списке аргументов должен находиться указатель на переменную типа `double`, а не типа `float`.

Например, обращение

```
int i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name);
```

со строкой на вводе

```
25     54.32e-1     thompson
```

приводит к присваиванию i значения 25, x – значения 5.432 и `name` – строки "thompson", надлежащим образом законченной символом \0. Эти три поля ввода можно разделить столькими пробелами, табуляциями и символами новых строк, сколько вы пожелаете. Обращение

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %2s", &i, &x, name);
```

с вводом

```
56789     0123     45a72
```

присвоит i значение 56, x – значение 789.0, пропустит 0123 и поместит в `name` строку "45". При следующем обращении к любой процедуре ввода рассмотрение начнётся с буквы a. В этих двух примерах `name` является указателем и, следовательно, перед ним не нужно помещать знак &.

В качестве другого примера перепишем теперь элементарный калькулятор из главы 4, используя для преобразования ввода функцию `scanf`:

```
#include  <stdio.h>
main()
{       /* rudimentary desk calculator */
  double sum, v;
  sum = 0;
  while (scanf("%lf", &v) != EOF)
    printf("\t%.2f\n", sum += v);
}
```

выполнение функции scanf заканчивается либо тогда, когда она исчерпывает свою управляющую строку, либо когда некоторый элемент ввода не совпадает с управляющей спецификацией. В качестве своего значения она возвращает число правильно совпадающих и присвоенных элементов ввода. Это число может быть использовано для определения количества найденных элементов ввода. При выходе на конец файла возвращается EOF. Подчеркнём, что это значение отлично от 0, так как 0 возвращается когда следующий вводимый символ не удовлетворяет первой спецификации в управляющей строке. При следующем обращении к scanf поиск возобновляется непосредственно за последним введённым символом.

Заключительное предостережение: аргументы функции scanf должны быть указателями. Несомненно наиболее распространённая ошибка состоит в написании

```
scanf("%d", n);
```

вместо

```
scanf("%d", &n);
```

7.5 Форматное преобразование в памяти

От функции scanf и printf происходят функции sscanf и sprintf, которые осуществляют аналогичные преобразования, но оперируют со строкой, а не с файлом. Обращения к этим функциям имеют вид:

```
sprintf(string, control, arg1, arg2,...)
sscanf(string, control, arg1, arg2,...)
```

Как и раньше, функция sprintf преобразует свои аргументы arg1, arg2 и т.д. в соответствии с форматом, указанным в control, но помещает результаты в string, а не в стандартный вывод. Конечно, строка string должна быть достаточно велика, чтобы принять результат. Например, если name – это символьный массив, а n – целое, то

```
sprintf(name, "temp%d", n);
```

создаёт в name строку вида tempNNN, где NNN – значение n.

Функция sscanf выполняет обратные преобразования – она просматривает строку string в соответствии с форматом в аргументе control и помещает результирующие значения в аргументы arg1, arg2 и т.д. Эти аргументы должны быть указателями. В результате обращения

```
sscanf(name, "temp%d", &n);
```

переменная n получает значение строки цифр, следующих за temp в name.

Упражнение 7–2

Перепишите настольный калькулятор из главы 4, используя для ввода и преобразования чисел scanf и/или sscanf.

7.6 Доступ к файлам

Все до сих пор написанные программы читали из стандартного ввода и писали в стандартный вывод, относительно которых мы предполагали, что они магическим образом предоставлены программе местной операционной системой.

Следующим шагом в вопросе ввода-вывода является написание программы, работающей с файлом, который не связан заранее с программой. Одной из программ, которая явно демонстрирует потребность в таких операциях, является `cat`, которая объединяет набор из нескольких именованных файлов в стандартный вывод. Программа `cat` используется для вывода файлов на терминал и в качестве универсального сборщика ввода для программ, которые не имеют возможности обращаться к файлам по имени. Например, команда

```
cat x.c y.c
```

печатает содержимое файлов x.c и y.c в стандартный вывод.

Вопрос состоит в том, как организовать чтение из именованных файлов, т.е., как связать внешние имена, которыми мыслит пользователь, с фактически читающими данные операторами.

Эти правила просты. Прежде чем можно считывать из некоторого файла или записывать в него, этот файл должен быть открыт с помощью функции `fopen` из стандартной библиотеки. Функция `fopen` берет внешнее имя (подобное x.c или y.c), проводит некоторые обслуживающие действия и переговоры с операционной системой (детали которых не должны нас касаться) и возвращает внутреннее имя, которое должно использоваться при последующих чтениях из файла или записях в него.

Это внутреннее имя, называемое «указателем файла», фактически является указателем структуры, которая содержит информацию о файле, такую как место размещения буфера, текущая позиция символа в буфере, происходит ли чтение из файла или запись в него и тому подобное. Пользователи не обязаны знать эти детали, потому что среди определений для стандартного ввода-вывода, получаемых из файла stdio.h, содержится определение структуры с именем `FILE`. Единственное необходимое для указателя файла описание демонстрируется примером:

```
FILE *fopen(), *fp;
```

Здесь говорится, что `fp` является указателем на `FILE` и `fopen` возвращает указатель на `FILE`. Обратите внимание, что `FILE` является именем типа, подобным `int`, а не ярлыку структуры; это реализовано с помощью `typedef`.[1]

Фактическое обращение к функции `fopen` в программе имеет вид:

```
fp = fopen(name, mode);
```

Первым аргументом функции `fopen` является «имя» файла, которое задаётся в виде символьной строки. Второй аргумент `mode` («режим») также является символьной строкой, которая указывает, как этот файл будет использоваться. Допустимыми режимами являются: чтение (”r”), запись (”w”) и добавление (”a”).

[1] Подробности того, как все это работает на системе UNIX, приведены в главе 8.

Если вы откроете файл, который ещё не существует, для записи или добавления, то такой файл будет создан (если это возможно). Открытие существующего файла на запись приводит к отбрасыванию его старого содержимого. Попытка чтения несуществующего файла является ошибкой. Ошибки могут быть обусловлены и другими причинами (например, попыткой чтения из файла, не имея на то разрешения). При наличии какой-либо ошибки функция возвращает нулевое значение указателя `NULL` (которое для удобства также определяется в файле stdio.h).

Другой необходимой вещью является способ чтения или записи, если файл уже открыт. Здесь имеется несколько возможностей, из которых `getc` и `putc` являются простейшими. Функция `getc` возвращает следующий символ из файла; ей необходим указатель файла, чтобы знать, из какого файла читать. Таким образом,

```
c = getc(fp);
```

помещает в «c» следующий символ из файла, указанного посредством fp, и `EOF`, если достигнут конец файла.

Функция `putc`, являющаяся обратной к функции `getc`,

```
putc(c, fp)
```

помещает символ «c» в файл fp и возвращает «c». Подобно функциям `getchar` и `putchar`, `getc` и `putc` могут быть макросами, а не функциями.

При запуске программы автоматически открываются три файла, которые снабжены определёнными указателями файлов. Этими файлами являются стандартный ввод, стандартный вывод и стандартный вывод ошибок; соответствующие указатели файлов называются `stdin`, `stdout` и `stderr`. Обычно все эти указатели связаны с терминалом, но `stdin` и `stdout` могут быть перенаправлены на файлы или в поток (pipe), как описывалось в разделе 7.2.

Функции `getchar` и `putchar` могут быть определены в терминах `getc`, `putc`, `stdin` и `stdout` следующим образом:

```
#define getchar()   getc(stdin)
#define putchar(c)  putc(c,stdout)
```

При работе с файлами для форматного ввода и вывода можно использовать функции `fscanf` и `fprintf`. Они идентичны функциям `scanf` и `printf`, за исключением того, что первым аргументом является указатель файла, определяющий тот файл, который будет читаться или куда будет вестись запись; управляющая строка будет вторым аргументом.

Покончив с предварительными замечаниями, мы теперь в состоянии написать программу `cat` для конкатенации файлов. Используемая здесь основная схема оказывается удобной во многих программах: если имеются аргументы в командной строке, то они обрабатываются последовательно. Если такие аргументы отсутствуют, то обрабатывается стандартный ввод. Это позволяет использовать программу как самостоятельно, так и как часть большей задачи.

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

```
main(argc, argv)        /* cat: concatenate files */
int argc;
char *argv[];
{
  FILE *fp, *fopen();
  if (argc == 1)        /* no args; copy standard input */
    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)    /* copy file fp to standard output */
FILE *fp;
{
  int c;
  while ((c = getc(fp)) != EOF)
    putc(c, stdout);
}
```

Указатели файлов `stdin` и `stdout` заранее определены в библиотеке ввода-вывода как стандартный ввод и стандартный вывод; они могут быть использованы в любом месте, где можно использовать объект типа `FILE*`; они однако являются константами, а не переменными, так что не пытайтесь им что-либо присваивать.

Функция `fclose` является обратной по отношению к `fopen`; она разрывает связь между указателем файла и внешним именем, установленную функцией `fopen`, и высвобождает указатель файла для другого файла.большинство операционных систем имеют некоторые ограничения на число одновременно открытых файлов, которыми может распоряжаться программа. Поэтому, то как мы поступили в `cat`, освободив не нужные нам более объекты, является хорошей идеей. Имеется и другая причина для применения функции `fclose` к выходному файлу – она вызывает выдачу информации из буфера, в котором `putc` собирает вывод.[1]

7.7 Обработка ошибок – stderr и exit

Обработка ошибок в `cat` неидеальна. Неудобство заключается в том, что если один из файлов по некоторой причине оказывается недоступным, диагностическое сообщение об

[1] При нормальном завершении работы программы функция `fclose` вызывается автоматически для каждого открытого файла.

этом печатается в конце объединённого вывода. Это приемлемо, если вывод поступает на терминал, но не годится, если вывод поступает в некоторый файл или через поточный (pipeline) механизм в другую программу.

Чтобы лучше обрабатывать такую ситуацию, к программе точно таким же образом, как `stdin` и `stdout`, присоединяется второй выходной файл, называемый `stderr`. Если это вообще возможно, вывод, записанный в поток `stderr`, появляется на терминале пользователя, даже если стандартный вывод направляется в другое место.

Давайте переделаем программу `cat` таким образом, чтобы сообщения об ошибках писались в стандартный поток ошибок.

```
#include <stdio.h>

main(argc, argv)        /* cat: concatenate files */
int argc;
char *argv[];
{
  FILE *fp, *fopen();
  if (argc == 1)        /* no args; copy standard input */
    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);
      }
  exit(0);
}
```

Программа сообщает об ошибках двумя способами. Диагностическое сообщение, выдаваемое функцией `fprintf`, поступает в `stderr` и, таким образом, оказывается на терминале пользователя, а не исчезает в потоке (pipeline) или в выходном файле.

Программа также использует функцию `exit` из стандартной библиотеки, обращение к которой вызывает завершение выполнения программы. Аргумент функции `exit` доступен любой программе, обращающейся к данной функции, так что успешное или неудачное завершение данной программы может быть проверено другой программой, использующей эту в качестве подзадачи. По соглашению величина 0 в качестве возвращаемого значения свидетельствует о том, что все в порядке, а различные ненулевые значения являются признаками нормальных ситуаций.

Функция `exit` вызывает функцию `fclose` для каждого открытого выходного файла, с тем чтобы вывести всю помещённую в буферы выходную информацию, а затем вызывает функцию `_exit`. Функция `_exit` приводит к немедленному завершению без очистки каких-либо буферов; конечно, при желании к этой функции можно обратиться непосредственно.

7.8 Ввод и вывод строк

Стандартная библиотека содержит функцию `fgetc`, совершенно аналогичную функции `getline`, которую мы использовали на всем протяжении книги. В результате обращения

```
fgets(line, maxline, fp)
```

следующая строка ввода (включая символ новой строки) считывается из файла `fp` в символьный массив `line`; самое большое `maxline-1` символ будет прочитан. Результирующая строка заканчивается символом \0. Нормально функция `fgetc` возвращает `line`; в конце файла она возвращает `NULL`. (Наша функция `getline` возвращает длину строки, а при выходе на конец файла – нуль).

Предназначенная для вывода функция `fputc` записывает строку (которая не обязана содержать символ новой строки) в файл:

```
fputs(line, fp)
```

Чтобы показать, что в функциях типа `fgetc` и `fputc` нет ничего таинственного, мы приводим их ниже, скопированными непосредственно из стандартной библиотеки ввода-вывода:

```
#include <stdio.h>

char *fgets(s, n, iop)  /* get at most n chars from iop */
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)   /* put string s on file iop */
register char *s;
register FILE *iop;
{
  register int c;
```

```
    while (c = *s++)
      putc(c, iop);
}
```

Упражнение 7–3

Напишите программу сравнения двух файлов, которая будет печатать первую строку и позицию символа, где они различаются.

Упражнение 7–4

Переделайте программу поиска заданной комбинации символов из главы 5 таким образом, чтобы в качестве ввода использовался набор именованных файлов или, если никакие файлы не указаны как аргументы, стандартный ввод. Следует ли печатать имя файла при нахождении подходящей строки?

Упражнение 7–5

Напишите программу печати набора файлов, которая начинает каждый новый файл с новой страницы и печатает для каждого файла заголовок и счётчик текущих страниц.

7.9 Несколько разнообразных функций

Стандартная библиотека предоставляет множество разнообразных функций, некоторые из которых оказываются особенно полезными. Мы уже упоминали функции для работы со строками: `strlen`, `strcpy`, `strcat` и `strcmp`. Вот некоторые другие.

7.9.1 Проверка вида символов и преобразования

Некоторые макросы выполняют проверку символов и преобразования:

`isalpha(c)` не 0, если «c» алфавитный символ, 0 – если нет.

`isupper(c)` Не 0, если «c» буква верхнего регистра, 0 – если нет.

`islower(c)` Не 0, если «c» буква нижнего регистра, 0 – если нет.

`isdigit(c)` Не 0, если «c» цифра, 0 – если нет.

`isspace(c)` Не 0, если «c» пробел, табуляция или новая строка, 0 – если нет.

`toupper(c)` Преобразует «c» в букву верхнего регистра.

`tolower(c)` Преобразует «c» в букву нижнего регистра.

7.9.2 Функция ungetc

Стандартная библиотека содержит довольно ограниченную версию функции `ungetch`, написанной нами в главе 4; она называется `ungetc`. В результате обращения

```
ungetc(c, fp)
```

символ «c» возвращается в файл `fp`. Позволяется возвращать в каждый файл только один символ. Функция `ungetc` может быть использована в любой из функций ввода и с макросами типа `scanf`, `getc` или `getchar`.

7.9.3 Обращение к системе

Функция `system(s)` выполняет команду, содержащуюся в символьной строке s, и затем возобновляет выполнение текущей программы. Содержимое s сильно зависит от используемой операционной системы. В качестве тривиального примера, укажем, что на системе UNIX строка

```
system("date");
```

приводит к выполнению программы `date`, которая печатает дату и время дня.

7.9.4 Управление памятью

Функция `calloc` весьма сходна с функцией `alloc`, использованной нами в предыдущих главах. В результате обращения

```
calloc(n, sizeof(object))
```

возвращается либо указатель пространства, достаточного для размещения N объектов указанного размера, либо `NULL`, если запрос не может быть удовлетворён. Отводимая память инициализируется нулевыми значениями.

Указатель обладает нужным для рассматриваемых объектов выравниванием, но ему следует приписывать соответствующий тип, как в примере:

```
char *calloc();
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

Функция `cfree(p)` освобождает пространство, на которое указывает p, причём указатель p первоначально должен быть получен в результате обращения к `calloc`. Здесь нет никаких ограничений на порядок освобождения пространства, но будет неприятнейшей ошибкой освободить что-нибудь, что не было получено обращением к `calloc`.

Реализация программы распределения памяти, подобной `calloc`, в которой размещённые блоки могут освобождаться в произвольном порядке, продемонстрирована в главе 8.

8 Интерфейс системы UNIX

Материал этой главы относится к интерфейсу между **C**-программами и операционной системой UNIX. Так как большинство пользователей языка **C** работают на системе UNIX, эта глава окажется полезной для большинства читателей. Даже если вы используете **C**-компилятор на другой машине, изучение приводимых здесь примеров должно помочь вам глубже проникнуть в методы программирования на языке **C**.

Эта глава делится на три основные части: ввод-вывод, система файлов и распределение памяти. Первые две части предполагают небольшое знакомство с внешними характеристиками системы UNIX.

В главе 7 мы имели дело с системным интерфейсом, который одинаков для всего многообразия операционных систем. На каждой конкретной системе функции стандартной библиотеки должны быть написаны в терминах ввода-вывода, доступных на данной машине. В следующих нескольких разделах мы опишем основную систему связанных с вводом и выводом точек входа операционной системы UNIX и проиллюстрируем, как с их помощью могут быть реализованы различные части стандартной библиотеки.

8.1 Дескрипторы файлов

В операционной системе UNIX весь ввод и вывод осуществляется посредством чтения файлов или их записи, потому что все периферийные устройства, включая даже терминал пользователя, являются файлами определённой файловой системы. Это означает, что один однородный интерфейс управляет всеми связями между программой и периферийными устройствами.

В наиболее общем случае перед чтением из файла или записью в файл необходимо сообщить системе о вашем намерении; этот процесс называется «открытием» файла. Система выясняет, имеете ли вы право поступать таким образом (существует ли этот файл? имеется ли у вас разрешение на обращение к нему?), и если все в порядке, возвращает в программу небольшое положительное целое число, называемое дескриптором файла. Всякий раз, когда этот файл используется для ввода или вывода, для идентификации файла употребляется дескриптор файла, а не его имя.[1] Вся информация об открытом файле содержится в системе; программа пользователя обращается к файлу только через дескриптор файла.

Для удобства выполнения обычных операций ввода и вывода с помощью терминала пользователя существуют специальные соглашения. Когда интерпретатор команд (SHELL) прогоняет программу, он открывает три файла, называемые стандартным вво-

[1] Здесь существует примерная аналогия с использованием `READ(5,...)` и `WRITE(6,...)` в фортране.

дом, стандартным выводом и стандартным выводом ошибок, которые имеют соответственно числа 0, 1 и 2 в качестве дескрипторов этих файлов. В нормальном состоянии все они связаны с терминалом, так что если программа читает с дескриптором файла 0 и пишет с дескрипторами файлов 1 и 2, то она может осуществлять ввод и вывод с помощью терминала, не заботясь об открытии соответствующих файлов.

Пользователь программы может перенаправлять ввод и вывод на файлы, используя операции командного интерпретатора SHELL < , > и |.

```
prog < infile > outfile
```

В этом случае интерпретатор команд SHELL изменит присваивание по умолчанию дескрипторов файлов 0 и 1 с терминала на указанные файлы. Нормально дескриптор файла 2 остаётся связанным с терминалом, так что сообщения об ошибках могут поступать туда. Подобные замечания справедливы и тогда, когда ввод и вывод связан с каналом. Следует отметить, что во всех случаях прикрепления файлов изменяются интерпретатором SHELL, а не программой. Сама программа, пока она использует файл 0 для ввода и файлы 1 и 2 для вывода, не знает ни откуда приходит её ввод, ни куда поступает её выдача.

8.2 Низкоуровневый ввод-вывод – операторы read и write

Самый низкий уровень ввода-вывода в системе UNIX не предусматривает ни какой-либо буферизации, ни какого-либо другого сервиса; он по существу является непосредственным входом в операционную систему. Весь ввод и вывод осуществляется двумя функциями: `read` и `write`. Первым аргументом обеих функций является дескриптор файла. Вторым аргументом является буфер в вашей программе, откуда или куда должны поступать данные. Третий аргумент – это число подлежащих пересылке байтов. Обращения к этим функциям имеют вид:

```
n_read = read(fd, buf, n);
n_written = write(fd, buf, n);
```

При каждом обращении возвращается счётчик байтов, указывающий фактическое число переданных байтов. При чтении возвращённое число байтов может оказаться меньше, чем запрошенное число. Возвращённое нулевое число байтов означает конец файла, а -1 указывает на наличие какой-либо ошибки. При записи возвращённое значение равно числу фактически записанных байтов; несовпадение этого числа с числом байтов, которое предполагалось записать, обычно свидетельствует об ошибке.

Количество байтов, подлежащих чтению или записи, может быть совершенно произвольным. Двумя самыми распространёнными величинами являются 1, которая означает передачу одного символа за обращение (т.е. без использования буфера), и 512, которая соответствует физическому размеру блока на многих периферийных устройствах. Этот последний размер будет наиболее эффективным, но даже ввод или вывод по одному символу за обращение не будет необыкновенно дорогим.

Объединив все эти факты, мы написали простую программу для копирования ввода на вывод, эквивалентную программе копировки файлов, написанной в главе 1. На системе UNIX эта программа будет копировать что угодно куда угодно, потому что ввод и вывод могут быть перенаправлены на любой файл или устройство.

```
#define BUFSIZE 512     /* best size for PDP-11 UNIX */

main()
{       /* copy input to output */
  char buf[BUFSIZE];
  int n;
  while ((n = read(0, buf, BUFSIZE)) > 0)
    write(1, buf, n);
}
```

Если размер файла не будет кратен BUFSIZE, то при некотором обращении к `read` будет возвращено меньшее число байтов, которые затем записываются с помощью `write`; при следующем после этого обращении к `read` будет возвращён нуль.

Поучительно разобраться, как можно использовать функции `read` и `write` для построения процедур более высокого уровня, таких как `getchar`, `putchar` и т.д. Вот, например, вариант функции `getchar`, осуществляющий ввод без использования буфера.

```
#define CMASK 0377      /* for making char's > 0 */

getchar()
{       /* unbuffered single character input */
  char c;
  return ((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

Переменная c должна быть описана как `char`, потому что функция `read` принимает указатель на символы. Возвращаемый символ должен быть маскирован числом 0377 для гарантии его положительности; в противном случае знаковый разряд может сделать его значение отрицательным.[1]

Второй вариант функции `getchar` осуществляет ввод большими порциями, а выдаёт символы по одному за обращение.

```
#define CMASK 0377      /* for making char's > 0 */
#define BUFSIZE 512

getchar()
{       /* buffered version */
  static char buf[BUFSIZE];
  static char *bufp = buf;
  static int n = 0;
```

[1]Константа 0377 подходит для ЭВМ PDP-11, но не обязательно для других машин.

```
  if (n == 0) {
    /*buffer is empty * /n = read(0, buf, BUFSIZE);
    bufp = buf;
  }
  return ((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

8.3 Открытие, создание, закрытие и расцепление (unlink)

Кроме случая, когда по умолчанию определены стандартные файлы ввода, вывода и ошибок, вы должны явно открывать файлы, чтобы затем читать из них или писать в них. Для этой цели существуют две точки входа: `open` и `creat`.

Функция `open` весьма сходна с функцией `fopen`, рассмотренной в главе 7, за исключением того, что вместо возвращения указателя файла она возвращает дескриптор файла, который является просто целым типа `int`.

```
int fd;
fd = open(name, rwmode);
```

Как и в случае `fopen`, аргумент `name` является символьной строкой, соответствующей внешнему имени файла. Однако аргумент, определяющий режим доступа, отличен: `rwmode` равно: 0 для чтения, 1 – для записи, 2 – для чтения и записи. Если происходит какая-то ошибка, функция `open` возвращает -1; в противном случае она возвращает действительный дескриптор файла.

Попытка открыть файл, который не существует, является ошибкой. Точка входа `creat` предоставляет возможность создания новых файлов или перезаписи старых. В результате обращения

```
fd = creat(name, mode);
```

возвращает дескриптор файла, если оказалось возможным создать файл с именем `name`, и -1 в противном случае. Если файл с таким именем уже существует, `creat` усечёт его до нулевой длины; создание файла, который уже существует, не является ошибкой.

Если файл является совершенно новым, то `creat` создаёт его с определённым режимом защиты, специфицируемым аргументом `mode`. В системе файлов на UNIX с файлом связываются девять битов защиты информации, которые управляют разрешением на чтение, запись и выполнение для владельца файла, для группы владельцев и для всех остальных пользователей. Таким образом, трёхзначное восьмеричное число наиболее удобно для спецификации разрешений. Например, число 0755 свидетельствует о разрешении на чтение, запись и выполнение для владельца и о разрешении на чтение и выполнение для группы и всех остальных.

Для иллюстрации ниже приводится программа копирования одного файла в другой, являющаяся упрощённым вариантом утилиты cp системы UNIX.[1].

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644      /* rw for owner, r for group,others */

main(argc, argv)        /* cp: copy f1 to 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)   /* print error message and die */
char *s1, s2;
{
  printf(s1, s2);
  printf("\n");
  exit(1);
}
```

Существует ограничение (обычно 15 – 25) на количество файлов, которые программа может иметь открытыми одновременно. В соответствии с этим любая программа, собирающаяся работать со многими файлами, должна быть подготовлена к повторному использованию дескрипторов файлов. Процедура `close` прерывает связь между дескриптором файла и открытым файлом и освобождает дескриптор файла для использования с некоторым другим файлом. Завершение выполнения программы через `exit` или в результате возврата из ведущей программы приводит к закрытию всех открытых файлов.

Функция расцепления `unlink(filename)` удаляет из системы файлов файл с именем `filename`.[2]

[1]Основное упрощение заключается в том, что наш вариант копирует только один файл и что второй аргумент не должен быть справочником

[2]Из данного справочного файла. Файл может быть сцеплен с другим справочником, возможно, под другим именем – примечание переводчика.

Упражнение 8–1

Перепишите программу `cat` из главы 7, используя функции `read`, `write`, `open` и `close` вместо их эквивалентов из стандартной библиотеки. Проведите эксперименты для определения относительной скорости работы этих двух вариантов.

8.4 Произвольный доступ – seek и lseek

Нормально при работе с файлами ввод и вывод осуществляется последовательно: при каждом обращении к функциям `read` и `write` чтение или запись начинаются с позиции, непосредственно следующей за предыдущей обработанной. Но при необходимости файл может читаться или записываться в любом произвольном порядке. Обращение к системе с помощью функции `lseek` позволяет передвигаться по файлу, не производя фактического чтения или записи. В результате обращения

```
lseek(fd, offset, origin);
```

текущая позиция в файле с дескриптором `fd` передвигается на позицию `offset` (смещение), которая отсчитывается от места, указываемого аргументом `origin` (начало отсчёта). Последующее чтение или запись будут теперь начинаться с этой позиции. Аргумент `offset` имеет тип `long`; `fd` и `origin` имеют тип `int`. Аргумент `origin` может принимать значения 0, 1 или 2, указывая на то, что величина `offset` должна отсчитываться соответственно от начала файла, от текущей позиции или от конца файла. Например, чтобы дополнить файл, следует перед записью найти его конец:

```
lseek(fd, 0l, 2);
```

чтобы вернуться к началу («перемотать обратно»), можно написать:

```
lseek(fd, 0l, 0);
```

обратите внимание на аргумент `0l`; его можно было бы записать и в виде `(long)0`.

Функция `lseek` позволяет обращаться с файлами примерно так же, как с большими массивами, правда ценой более медленного доступа. Следующая простая функция, например, считывает любое количество байтов, начиная с произвольного места в файле.

```
get(fd, pos, buf, n)    /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
  lseek(fd, pos, 0);    /* get to pos */
  return (read(fd, buf, n));
}
```

В более ранних редакциях, чем редакция 7 системы UNIX, основная точка входа в систему ввода-вывода называется `seek`. Функция `seek` идентична функции `lseek`, за исключением того, что аргумент `offset` имеет тип `int`, а не `long`. В соответствии с этим, поскольку на PDP-11 целые имеют только 16 битов, аргумент `offset`, указываемый функции `seek`, ограничен величиной 65535; по этой причине аргумент `origin` может иметь значения 3, 4, 5, которые заставляют функцию `seek` умножить заданное значение `offset` на 512 (количество байтов в одном физическом блоке) и затем интерпретировать `origin`, как если это 0, 1 или 2 соответственно. Следовательно, чтобы достичь произвольного места в большом файле, нужно два обращения к `seek`: сначала одно, которое выделяет нужный блок, а затем второе, где `origin` имеет значение 1 и которое осуществляет передвижение на желаемый байт внутри блока.

Упражнение 8–2

Очевидно, что `seek` может быть написана в терминалах `lseek` и наоборот. Напишите каждую функцию через другую.

8.5 Пример – реализация функций fopen и getc

Давайте теперь на примере реализации функций `fopen` и `getc` из стандартной библиотеки подпрограмм продемонстрируем, как некоторые из описанных элементов объединяются вместе.

Напомним, что в стандартной библиотеке файлы описываются посредством указателей файлов, а не дескрипторов. Указатель файла является указателем на структуру, которая содержит несколько элементов информации о файле: указатель буфера, чтобы файл мог читаться большими порциями; счётчик числа символов, оставшихся в буфере; указатель следующей позиции символа в буфере; некоторые признаки, указывающие режим чтения или записи и т.д.; дескриптор файла.

Описывающая файл структура данных содержится в файле stdio.h, который должен включаться (посредством `#include`) в любой исходный файл, в котором используются функции из стандартной библиотеки. Он также включается функциями этой библиотеки. В приводимой ниже выдержке из файла stdio.h имена, предназначаемые только для использования функциями библиотеки, начинаются с подчёркивания, с тем чтобы уменьшить вероятность совпадения с именами в программе пользователя.

```
#define _BUFSIZE 512
#define _NFILE   20     /* files that can be handled */

typedef struct _iobuf {
  char *_ptr;   /* next character position */
  int _cnt;     /* number of characters left */
  char *_base;  /* location of buffer */
  int _flag;    /* mode of file access */
  int _fd;      /* file descriptor */
```

```
} file;
extern file _iob[_NFILE];

#define   stdin         (&_iob[0])
#define   stdout        (&_iob[1])
#define   stderr        (&_iob[2])

#define   _READ   01    /* file open for reading */
#define   _WRITE  02    /* file open for writing */
#define   _UNBUF  04    /* file is unbuffered */
#define   _BIGBUF 010   /* big buffer allocated */
#define   _EOF 020      /* EOF has occurred on this file */
#define   _ERR 040      /* error has occurred on this file */

#define   NULL 0
#define   EOF  (-1)

#define   getc(p) (--(p)->_cnt >= 0 \
                        ? *(p)->_ptr++ & 0377 : _filebuf(p))
#define   getchar()     getc(stdin)

#define   putc(x,p) (--(p)->_cnt >= 0 \
                        ? *(p)->_ptr++ = (x) : _flushbuf((x),p))
#define   putchar(x)    putc(x,stdout)
```

В нормальном состоянии макрос `getc` просто уменьшает счётчик, передвигает указатель и возвращает символ.[1] Если однако счётчик становится отрицательным, то `getc` вызывает функцию `_filebuf`, которая снова заполняет буфер, реинициализирует содержимое структуры и возвращает символ. Функция может предоставлять переносимый интерфейс и в то же время содержать непереносимые конструкции: `getc` маскирует символ числом 0377, которое подавляет знаковое расширение, осуществляемое на PDP-11, и тем самым гарантирует положительность всех символов.

Хотя мы не собираемся обсуждать какие-либо детали, мы все же включили сюда определение макроса `putc`, для того чтобы показать, что она работает в основном точно также, как и `getc`, обращаясь при заполнении буфера к функции `_flushbuf`.

Теперь может быть написана функция `fopen`. Большая часть программы функции `fopen` связана с открыванием файла и расположением его в нужном месте, а также с установлением битов признаков таким образом, чтобы они указывали нужное состояние. Функция `fopen` не выделяет какой-либо буферной памяти; это делается функцией `_filebuf` при первом чтении из файла.

```
#include <stdio.h>
#define  PMODE  0644    /* r/w for owner;r for others */
```

[1] Если определение `#define` слишком длинное, то оно продолжается с помощью обратной косой черты.

```
file *fopen(name, mode) /* open file,return file 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;    /* found free slot */

  if (fp >= _iob + _NFILE)      /* no free slots */
    return (NULL);

  if (*mode == 'w')     /* access file */
    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) /* couldn't access name */
    return (NULL);

  fp->_fd = fd;
  fp->_cnt = 0;
  fp->_base = NULL;
  fp->_flag &= (_READ | _WRITE);
  fp->_flag |= (*mode == 'r') ? _READ : _WRITE;
  return (fp);
}
```

Функция `_filebuf` несколько более сложная. Основная трудность заключается в том, что `_filebuf` стремится разрешить доступ к файлу и в том случае, когда может не оказаться достаточно места в памяти для буферизации ввода или вывода. Если пространство для нового буфера может быть получено обращением к функции `calloc`, то все отлично; если же нет, то `_filebuf` осуществляет небуферизованный ввод–вывод, используя отдельный символ, помещённый в локальном массиве.

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

```
_fillbuf(fp)    /* allocate and fill input buffer */
register file *fp;
{
  static char smallbuf(_NFILE); /* for unbuffered 1/0 */
  char *calloc();

  if ((fp->_flag & _READ) == 0 || (fp->_flag & (EOF | _ERR)) != 0)
    return (EOF);

  while (fp->_base == NULL)     /* find buffer space */
    if (fp->_flag & _UNBUF)     /* unbuffered */
      fp->_base = &smallbuf[fp->_fd];
    else if ((fp->_base = calloc(_BUFSIZE, 1)) == NULL)
      fp->_flag = _UNBUF;       /* can't get big buf */
    else
      fp->_flag = _BIGBUF;      /* got big one */

  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);  /* make char positive */
}
```

При первом обращении к `getc` для конкретного файла счётчик оказывается равным нулю, что приводит к обращению к `_filebuf`. Если функция `_filebuf` найдёт, что этот файл не открыт для чтения, она немедленно возвращает `EOF`. В противном случае она пытается выделить большой буфер, а если ей это не удаётся, то буфер из одного символа. При этом она заносит в `_flag` соответствующую информацию о буферизации.

Раз буфер уже создан, функция `_filebuf` просто вызывает функцию `read` для его заполнения, устанавливает счётчик и указатели и возвращает символ из начала буфера.

Единственный оставшийся невыясненным вопрос состоит в том, как все начинается. Массив `_iob` должен быть определён и инициализирован для `stdin`, `stdout` и `stderr`:

```
file _iob[nfile] = {
  (NULL, 0, _READ, 0),                  /* stdin */
  (NULL, 0, NULL, 1),                   /* stdout */
  (NULL, 0, NULL, _WRITE | _UNBUF, 2)   /* stderr */
};
```

Из инициализации части `_flag` этого массива структур видно, что файл `stdin` предназначен для чтения, файл `stdout` – для записи и файл `stderr` – для записи без использования буфера.

Упражнение 8–3

Перепишите функции `fopen` и `_filebuf`, используя поля вместо явных побитовых операций.

Упражнение 8–4

Разработайте и напишите функции `_flushbuf` и `fclose`.

Упражнение 8–5

Стандартная библиотека содержит функцию

```
fseek(fp, offset, origin)
```

которая идентична функции `lseek`, исключая то, что `fp` является указателем файла, а не дескриптором файла. Напишите `fseek`. Убедитесь, что ваша `fseek` правильно согласуется с буферизацией, сделанной для других функций библиотеки.

8.6 Пример – распечатка справочников

Иногда требуется другой вид взаимодействия с системой файлов – определение информации о файле, а не того, что в нем содержится. Примером может служить команда ls («список справочника») системы UNIX. По этой команде распечатываются имена файлов из справочника и, необязательно, другая информация, такая как размеры, разрешения и т.д.

Поскольку, по крайней мере, на системе UNIX справочник является просто файлом, то в такой команде, как ls нет ничего особенного; она читает файл и выделяет нужные части из находящейся там информации. Однако формат информации определяется системой, так что ls должна знать, в каком виде все представляется в системе.

Мы это частично проиллюстрируем при написании программы `fsize`. Программа `fsize` представляет собой специальную форму ls, которая печатает размеры всех файлов, указанных в списке её аргументов. Если один из файлов является справочником, то для обработки этого справочника программа `fsize` обращается сама к себе рекурсивно. Если же аргументы вообще отсутствуют, то обрабатывается текущий справочник.

Для начала дадим краткий обзор структуры системы файлов. Справочник – это файл, который содержит список имён файлов и некоторое указание о том, где они размещаются. Фактически это указание является индексом для другой таблицы, которую называют

«i – узловой таблицей» (inode). Для файла i-узел – это то, где содержится вся информация о файле, за исключением его имени. Запись в справочнике состоит только из двух элементов: номера i-узла и имени файла. Точная спецификация поступает при включении файла sys/dir.h, который содержит

```
#define DIRSIZ 14       /* max length of file name */
struct direct {         /* structure of directory entry */
  ino_t & _ino;         /* inode number */
  char &_name[DIRSIZ];  /* file name */
};
```

«Тип» `ino_t` – это определяемый посредством `typedef` тип, который описывает индекс i-узловой таблицы. На PDP-11 UNIX этим типом оказывается `unsigned`, но это не тот сорт информации, который помещают внутрь программы: на разных системах этот тип может быть различным. Поэтому и следует использовать `typedef`. Полный набор «системных» типов находится в файле sys/types.h.

Функция `stat` берет имя файла и возвращает всю содержащуюся в i-ом узле информацию об этом файле (или -1, если имеется ошибка). Таким образом, в результате

```
struct stat stbuf;
char *name;
stat(name, &stbuf);
```

структура `stbuf` наполняется информацией из i-го узла о файле с именем `name`. Структура, описывающая возвращаемую функцией `stat` информацию, находится в файле sys/stat.h и выглядит следующим образом:

```
struct stat {   /* structure returned by stat */
  dev_t st_dev;         /* device of inode */
  ino_t st_ino;         /* inode number */
  short st_mode         /* mode bits */
  short st_nlink;       /* number of links to file */
  short st_uid;         /* owner's user id */
  short st_gid;         /* owner's group id */
  dev_t st_rdev;        /* for special files */
  off_t st_size;        /* file size in characters */
  time_t st_atime;      /* time last accessed */
  time_t st_mtime;      /* time last modified */
  time_t st_ctime;      /* time originally created */
}
```

Большая часть этой информации объясняется в комментариях. Элемент `st_mode` содержит набор флагов, описывающих файл; для удобства определения флагов также находятся в файле sys/stat.h.

```
#define S_IFMT    0160000       /* type of file */
#define S_IFDIR   0040000       /* directory */
#define S_IFCHR   0020000       /* character special */
```

```
#define S_IFBLK   0060000       /* block special */
#define S_IFREG   0100000       /* regular */
#define S_ISUID   04000         /* set user id on execution */
#define S_ISGID   02000         /* set group id on execution */
#define S_ISVTX   01000         /* save swapped text after use */
#define S_IREAD   0400          /* read permission */
#define S_IWRITE  0200          /* write permission */
#define S_IEXEC   0100          /* execute permission */
```

Теперь мы в состоянии написать программу `fsize`. Если полученный от функции `stat` режим указывает, что файл не является справочником, то его размер уже под рукой и может быть напечатан непосредственно. Если же он оказывается справочником, то мы должны обрабатывать этот справочник отдельно для каждого файла; так как справочник может в свою очередь содержать подсправочники, этот процесс обработки является рекурсивным.

Как обычно, ведущая программа главным образом имеет дело с командной строкой аргументов; она передаёт каждый аргумент функции `fsize` в большой буфер.

```
#include <stdio.h.>
#include <sys/types.h>  /* typedefs */
#include <sys/dir.h>    /* directory entry structure */
#include <sys/stat.h>   /* structure returned by stat */
#define BUFSIZE  256

main(argc, argv)        /* fsize:print file sizes */
char *argv[];
{
  char buf[BUFSIZE];

  if (argc == 1) {      /* default:current directory */
    strcpy(buf, ".");
    fsize(buf);
  } else
    while (--argc > 0) {
      strcpy(buf, *++argv);
      fsize(buf);
    }
}
```

Функция `fsize` печатает размер файла. Если однако файл оказывается справочником, то `fsize` сначала вызывает функцию `directory` для обработки всех указанных в нем файлов. Обратите внимание на использование имён флагов `S_IFMT` и `S_IFDIR` из файла stat.h.

```
fsize(name)     /* print size for name */
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 является самой сложной. Однако значительная её часть связана с созданием для обрабатываемого в данный момент файла его полного имени, по которому можно восстановить путь в дереве.

```
directory(name) /* fsize for all files in name */
char *name;
{
  struct direct dirbuf;
  char *nbp, *nep;
  int i, fd;
  nbp = name + strlen(name);
  *nbp++ = '/'; /* add slash to directory name */
  if (nbp + DIRSIZ + 2 >= name + BUFSIZE)       /* name too long */
    return;
  if ((fd = open(name, 0)) == -1)
    return;

  while (read(fd, (char *) &dirbuf, sizeof(dirbuf)) > 0) {
    if (dirbuf.d_ino == 0)      /* slot not in use */
      continue;
    if (strcmp(dirbuf.d_name, ".") == 0
        || strcmp(dirbuf.d_name, "..") == 0)
      continue; /* skip self and parent */
    for (i = 0, nep = nbp; i < DIRSIZ; i++)
      *nep++ = dirbuf.d_name[i];
    *nep++ = '\0';
    fsize(name);
  }
  close(fd);
  *--nbp = '\0';        /* restore name */
}
```

Если некоторая дыра в справочнике в настоящее время не используется (потому что файл был удалён), то в соответствующее i-узловое число равно нулю, и эта позиция пропускается. Каждый справочник также содержит запись в самом себе, называемую « . », и

о своём родителе, « .. »; они, очевидно, также должны быть пропущены, а то программа будет работать весьма и весьма долго.

Хотя программа `fsize` довольно специализированна, она все же демонстрирует пару важных идей. Во-первых, многие программы не являются «системными программами»; они только используют информацию, форма или содержание которой определяется операционной системой. Во-вторых, для таких программ существенно, что представление этой информации входит только в стандартные «заголовочные файлы», такие как stat.h и dir.h, и что программы включают эти файлы, а не помещают фактические описания внутрь самих программ.

8.7 Пример – распределитель памяти

В главе 5 мы написали бесхитростный вариант функции `alloc`. Вариант, который мы напишем теперь, не содержит ограничений: обращения к функциям `alloc` и `free` могут перемежаться в любом порядке; когда это необходимо, функция `alloc` обращается к операционной системе за дополнительной памятью. Кроме того, что эти процедуры полезны сами по себе, они также иллюстрируют некоторые соображения, связанные с написанием машинно-зависимых программ относительно машинно-независимым образом, и показывают практическое применение структур, объединений и конструкций `typedef`.

Вместо того, чтобы выделять память из скомпилированного внутри массива фиксированного размера, функция `alloc` будет по мере необходимости обращаться за памятью к операционной системе. Поскольку различные события в программе могут требовать асинхронного выделения памяти, то память, управляемая `alloc`, не может быть непрерывной. В силу этого свободная память хранится в виде цепочки свободных блоков. Каждый блок включает размер, указатель следующего блока и саму свободную память. Блоки упорядочиваются в порядке возрастания адресов памяти, причём последний блок (с наибольшим адресом) указывает на первый, так что цепочка фактически оказывается кольцом.

При поступлении запроса список свободных блоков просматривается до тех пор, пока не будет найден достаточно большой блок. Если этот блок имеет в точности требуемый размер, то он отцепляется от списка и передаётся пользователю. Если же этот блок слишком велик, то он разделяется, нужное количество передаётся пользователю, а остаток возвращается в свободный список. Если достаточно большого блока найти не удаётся, то операционной системой выделяется новый блок, который включается в список свободных блоков; затем поиск возобновляется.

Освобождение памяти также влечёт за собой просмотр свободного списка в поиске подходящего места для введения освобождённого блока. Если этот освободившийся блок с какой-либо стороны примыкает к блоку из списка свободных блоков, то они объединяются в один блок большего размера, так что память не становится слишком раздробленной. Обнаружить смежные блоки просто, потому что свободный список содержится в порядке возрастания адресов.

Одна из проблем, о которой мы упоминали в главе 5, заключается в обеспечении того, чтобы возвращаемая функцией `alloc` память была выровнена подходящим образом для

тех объектов, которые будут в ней храниться. Хотя машины и различаются, для каждой машины существует тип, требующий наибольших ограничений по размещению памяти, если данные самого ограничительного типа можно поместить в некоторый определённый адрес, то это же возможно и для всех остальных типов. Например, на IBM 360/370, HONEYWELL 6000 и многих других машинах любой объект может храниться в границах, соответствующим переменным типа `double`; на PDP-11 будут достаточны переменные типа `int`.

Свободный блок содержит указатель следующего блока в цепочке, запись о размере блока и само свободное пространство; управляющая информация в начале называется заголовком. Для упрощения выравнивания все блоки кратны размеру заголовка, а сам заголовок выровнен надлежащим образом. Это достигается с помощью объединения, которое содержит желаемую структуру заголовка и образец наиболее ограничительного по выравниванию типа:

```
typedef int align;      /* forces alignment on pdp-11 */
union header {  /* free block header */
  struct {
    union header *ptr;  /* next free block */
    unsigned size;      /* size of this free block */
  } s;
  align x;      /* force alignment of blocks */
};
typedef union header header;
```

Функция `alloc` округляет требуемый размер в символах до нужного числа единиц размера заголовка; фактический блок, который будет выделен, содержит на одну единицу больше, предназначаемую для самого заголовка, и это и есть значение, которое записывается в поле `size` заголовка. Указатель, возвращаемый функцией `alloc`, указывает на свободное пространство, а не на сам заголовок.

```
static header base;             /* empty list to get started */
static header *allocp = NULL;   /* last allocated block */

char *alloc(nbytes)     /* general-purpose storage allocator */
unsigned nbytes;
{
  header *morecore();
  register header *p, *g;
  register int nunits;

  nunits = 1 + (nbytes + sizeof(header) - 1) / sizeof(header);
  if ((g = allocp) == NULL) {   /* no free list yet */
    base.s.ptr = allocp = g = &base;
    base.s.size = 0;
  }
```

```
  for (p = g->s.ptr;; g = p, p = p->s.ptr) {
    if (p->s.size >= nunits) {  /* big enough */
      if (p->s.size == nunits)  /* exactly */
        g->s.ptr = p->s.ptr;
      else {    /* allocate tail end */
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      allocp = g;
      return ((char *) (p + 1));
    }
    if (p == allocp)    /* wrapped around free list */
      if ((p = morecore(nunits)) == NULL)
        return (NULL);  /* none left */
  }
}
```

Переменная `base` используется для начала работы. Если `allocp` имеет значение `NULL`, как в случае первого обращения к `alloc`, то создаётся вырожденный свободный список: он состоит из свободного блока размера нуль и указателя на самого себя. В любом случае затем исследуется свободный список. Поиск свободного блока подходящего размера начинается с того места (`allocp`), где был найден последний блок; такая стратегия помогает сохранить однородность диска. Если найден слишком большой блок, то пользователю предлагается его хвостовая часть; это приводит к тому, что в заголовке исходного блока нужно изменить только его размер. Во всех случаях возвращаемый пользователю указатель указывает на действительно свободную область, лежащую на единицу дальше заголовка. Обратите внимание на то, что функция `alloc` перед возвращением `p` преобразует его в указатель на символы.

Функция `morecore` получает память от операционной системы. Детали того, как это осуществляется, меняются, конечно, от системы к системе. На системе UNIX точка входа `sbrk(n)` возвращает указатель на n дополнительных байтов памяти.[1] Так как запрос к системе на выделение памяти является сравнительно дорогой операцией, мы не хотим делать это при каждом обращении к функции `alloc`. Поэтому функция `morecore` округляет затребованное число единиц до большего значения; этот больший блок будет затем разделён так, как необходимо. Масштабирующая величина является параметром, который может быть подобран в соответствии с необходимостью.

```
#define NALLOC 128      /* #units to allocate at once */

static header *morecore(nu)     /* ask system for memory */
unsigned nu;
{
  char *sbrk();
```

[1] Указатель удовлетворяет всем ограничениям на выравнивание.

```
  register char *cp;
  register header *up;
  register int rnu;

  rnu = NALLOC * ((nu + NALLOC - 1) / NALLOC);
  cp = sbrk(rnu * sizeof(header));
  if ((int) cp == -1)   /* no space at all */
    return (NULL);

  up = (header *) cp;
  up->s.size = rnu;
  free((char *) (up + 1));
  return (allocp);
}
```

Если больше не осталось свободного пространства, то функция `sbrk` возвращает -1, хотя `NULL` был бы лучшим выбором. Для надёжности сравнения -1 должна быть преобразована к типу `int`. Снова приходится многократно использовать явные преобразования (перевод) типов, чтобы обеспечить определённую независимость функций от деталей представления указателей на различных машинах.

И последнее – сама функция `free`. Начиная с `allocp`, она просто просматривает свободный список в поиске места для введения свободного блока. Это место находится либо между двумя существующими блоками, либо в одном из концов списка. В любом случае, если освободившийся блок примыкает к одному из соседних, смежные блоки объединяются. Следить нужно только затем, чтобы указатели указывали на то, что нужно, и чтобы размеры были установлены правильно.

```
free(ap)        /* put blocke ap in free list */
char *ap;
{
  register header *p, *g;

  p = (header *) ap - 1;        /* point to header */
  for (g = allocp; !(p > g && p > g->s.ptr); g = g->s.ptr)
    if (g >= g->s.ptr && (p > g || p < g->s.ptr))
      break;    /* at one end or other */

  if (p + p->s.size == g->s.ptr) {      /* join to upper nbr */
    p->s.size += g->s.ptr->s.size;
    p->s.ptr = g->s.ptr->s.ptr;
  } else
    p->s.ptr = g->s.ptr;

  if (g + g->s.size == p) {     /* join to lower nbr */
    g->s.size += p->s.size;
    g->s.ptr = p->s.ptr;
  } else
```

```
        g->s.ptr = p;

      allocp = g;
    }
```

Хотя распределение памяти по своей сути зависит от используемой машины, приведённая выше программа показывает, как эту зависимость можно регулировать и ограничить весьма небольшой частью программы. Использование `typedef` и `union` позволяет справиться с выравниванием (при условии, что функция `sbrk` обеспечивает подходящий указатель). Переводы типов организуют выполнение явного преобразования типов и даже справляются с неудачно разработанным системным интерфейсом. И хотя рассмотренные здесь подробности связаны с распределением памяти, общий подход равным образом применим и к другим ситуациям.

Упражнение 8–6

Функция из стандартной библиотеки `calloc(n,size)` возвращает указатель на n объектов размера `size`, причём соответствующая память инициализируется на нуль. Напишите программу для `calloc`, используя функцию `alloc` либо в качестве образца, либо как функцию, к которой происходит обращение.

Упражнение 8–7

Функция `alloc` принимает затребованный размер, не проверяя его правдоподобности; функция `free` полагает, что тот блок, который она должна освободить, содержит правильное значение в поле размера. Усовершенствуйте эти процедуры, затратив больше усилий на проверку ошибок.

Упражнение 8–8

Напишите функцию `bfree(p,n)`, которая включает произвольный блок p из n символов в список свободных блоков, управляемый функциями `alloc` и `free`. С помощью функции `bfree` пользователь может в любое время добавлять в свободный список статический или внешний массив.

9 Приложение A: Справочное руководство по языку C

9.1 Введение

Это руководство описывает язык **C** для компьютеров DEC PDP-11, HONEYWELL 6000, IBM SYSTEM/370 И INTERDATA 8/32. Там, где есть расхождения, мы сосредотачиваемся на версии для PDP-11, стремясь в то же время указать детали, которые зависят от реализации. За малым исключением, эти расхождения непосредственно обусловлены основными свойствами используемого аппаратного оборудования; различные компиляторы обычно вполне совместимы.

9.2 Лексические соглашения

Имеется шесть классов лексем: идентификаторы, ключевые слова, константы, строки, операции и другие разделители. Пробелы, табуляции, новые строки и комментарии (совместно, «пустые промежутки»), как описано ниже, игнорируются, за исключением тех случаев, когда они служат разделителями лексем. Необходим какой-то пустой промежуток для разделения идентификаторов, ключевых слов и констант, которые в противном случае сольются.

Если сделан разбор входного потока на лексемы вплоть до данного символа, то в качестве следующей лексемы берется самая длинная строка символов, которая еще может представлять собой лексему.

9.2.1 Комментарии

Комментарий открывается символами `/*` и заканчивается символами `*/`. Комментарии не вкладываются друг в друга.

9.2.2 Идентификаторы (имена)

Идентификатор – это последовательность букв и цифр; первый символ должен быть буквой. Подчеркивание «_» считается буквой. Буквы нижнего и верхнего регистров различаются. Значащими являются не более, чем первые восемь символов, хотя можно использовать и больше. На внешние идентификаторы, которые используются различными ассемблерами и загрузчиками, накладываются более жесткие ограничения:

DEC PDP–11	7 символов,	2 регистра
HONEYWELL 6000	6 символов,	1 регистр
IBM 360/370	7 символов,	1 регистр
INTERDATA 8/32	8 символов,	2 регистра

9.2.3 Ключевые слова

Следующие идентификаторы зарезервированы для использования в качестве ключевых слов и не могут использоваться иным образом:

```
int       extern    if        switch
char      register  else      case
float     static    while     default
double    auto      for       goto
long      struct    do        return
short     union     break     sizeof
unsigned  typedef   continue  entry
```

Ключевое слово `entry` в настоящее время не используется каким-либо компилятором; оно зарезервировано для использования в будущем. В некоторых реализациях резервируется также слова `fortran` и `asm`.

9.2.4 Константы

Имеется несколько видов констант, которые перечислены ниже. В пункте 9.3 резюмируются характеристики аппаратных средств, которые влияют на размеры.

Целые константы

Целая константа, состоящая из последовательности цифр, считается восьмеричной, если она начинается с 0 (цифра нуль), и десятичной в противном случае. Цифры 8 и 9 имеют восьмеричные значения 010 и 011 соответственно. Последовательность цифр, которой предшествуют символы `0x` или `0X`, рассматривается как шестнадцатеричное целое. Шестнадцатеричные цифры включают буквы от a (или A) до f (или F) со значениями от 10 до 15. Десятичная константа, величина которой превышает наибольшее машинное целое со знаком, считается длинной; восьмеричная или шестнадцатеричная константа, которое превышает наибольшее машинное целое без знака, также считается длинной.

Явные длинные константы

Десятичная, восьмеричная или шестнадцатеричная константа, за которой непосредственно следует `l` (или `L`), является длинной константой. Как обсуждается ниже, на некоторых машинах целые и длинные значения могут рассматриваться как идентичные.

Символьные константы

Символьная константа – это символ, заключенный в одиночные кавычки, как, например, 'x'. Значением символьной константы является численное значение этого символа в машинном представлении набора символов.

Некоторые неграфические символы, одиночная кавычка ' и обратная косая черта \ могут быть представлены в соответствии со следующей таблицей условных последовательностей:

новая строка	NL(LF)	\n
горизонтальная табуляция	HT	\t
символ возврата на одну позицию	BS	\b
возврат каретки	CR	\r
переход на новую страницу	FF	\f
обратная косая черта	\	\\
одиночная кавычка	'	\'
комбинация битов	ddd	\ddd

Условная последовательность \ddd состоит из обратной косой черты, за которой следуют 1,2 или 3 восьмеричных цифры, которые рассматриваются как задающие значение желаемого символа. Специальным случаем этой конструкции является последовательность \0 (за нулем не следует цифра), которая определяет символ NULL. Если следующий за обратной косой чертой символ не совпадает с одним из указанных, то обратная косая черта игнорируется.

Плавающие константы

Плавающая константа состоит из целой части, десятичной точки, дробной части, буквы e (или E) и целой экспоненты с необязательным знаком. Как целая, так и дробная часть являются последовательностью цифр. Либо целая, либо дробная часть (но не обе) может отсутствовать; либо десятичная точка, либо экспонента (но не то и другое одновременно) может отсутствовать. Каждая плавающая константа считается имеющей двойную точность.

9.2.5 Строки

Строка – это последовательность символов, заключенная в двойные кавычки, как, например, "...". Строка имеет тип «массив массивов» и класс памяти `static` (см. пункт 4 ниже). Строка инициализирована указанными в ней символами. Все строки, даже идентично записанные, считаются различными. Компилятор помещает в конец каждой строки нулевой байт \0, с тем чтобы просматривающая строку программа могла определить ее конец. Перед стоящим внутри строки символом двойной кавычки " должен быть поставлен символ обратной косой черты \ ; кроме того, могут использоваться те же условные последовательности, что и в символьных константах. И последнее, обратная косая черта \ , за которой непосредственно следует символ новой строки, игнорируется.

9.3 Характеристики аппаратных средств

Следующая ниже таблица суммирует некоторые свойства аппаратного оборудования, которые меняются от машины к машине. Хотя они и влияют на переносимость программ, на практике они представляют маленькую проблему, чем это может казаться заранее.

	DEC PDP–11	HONEYWELL 6000	IBM 370	INTERDATA 8/32
	ASCII	ASCII	EBCDIC	ASCII
`char`	8-BITS	9-BITS	8-BITS	8-BITS
`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
`range`	-38/+38	-38/+38	-76/+76	-76/+76

9.4 Синтаксическая нотация

В используемой в этом руководстве синтаксической нотации синтаксические категории выделяются рублёным шрифтом, а литерные слова и символы `равноширинным шрифтом`. Альтернативные категории перечисляются на отдельных строчках. Необязательный символ, терминальный или нетерминальный, указывается *наклонным шрифтом*, так что {*выражение*} указывает на необязательное выражение, заключенное в фигурных скобках. Синтаксис суммируется в пункте 9.19.

9.5 Что в имени тебе моем?

Язык **C** основывает интерпретацию идентификатора на двух признаках идентификатора: его классе памяти и его типе. Класс памяти определяет место и время хранения памяти, связанной с идентификатором; тип определяет смысл величин, находящихся в памяти, определенной под идентификатором.

Имеются четыре класса памяти: автоматическая, статическая, внешняя и регистровая. Автоматические переменные являются локальными для каждого вызова блока и исчезают при выходе из этого блока. Статические переменные являются локальными, но сохраняют свои значения для следующего входа в блок даже после того, как управление передается за пределы блока. Внешние переменные существуют и сохраняют свои значения в течение выполнения всей программы и могут использоваться для связи между функциями, в том числе и между независимо скомпилированными функциями. Регистровые переменные хранятся (если это возможно) в быстрых регистрах машины; подобно автоматическим переменным они являются локальными для каждого блока и исчезают при выходе из этого блока.

В языке **C** предусмотрено несколько основных типов объектов:

- объекты, написанные как символы (`char`), достаточно велики, чтобы хранить любой член из соответствующего данной реализации внутреннего набора символов, и если действительный символ из этого набора символов хранится в символьной переменной, то ее значение эквивалентно целому коду этого символа. В символьных переменных можно хранить и другие величины, но реализация будет машинно-зависимой.

- Можно использовать до трех размеров целых, описываемых как `short int`, `int` и `long int`. Длинные целые занимают не меньше памяти, чем короткие, но в конкретной реализации может оказаться, что либо короткие целые, либо длинные целые, либо те и другие будут эквивалентны простым целым. «Простые» целые имеют естественный размер, предусматриваемый архитектурой используемой машины; другие размеры вводятся для удовлетворения специальных потребностей.

- Целые без знака, описываемые как `unsigned`, подчиняются законам арифметики по модулю 2^n, где n – число битов в их представлении.[1]

- Плавающие одинарной точности (`float`) и плавающие двойной точности (`double`).[2]

Поскольку объекты упомянутых выше типов могут быть разумно интерпретированы как числа, эти типы будут называться арифметическими. Типы `char` и `int` всех размеров совместно будут называться целочисленными. Типы `float` и `double` совместно будут называться плавающими типами.

Кроме основных арифметических типов существует концептуально бесконечный класс производных типов, которые образуются из основных типов следующим образом:

– массивы объектов большинства типов;

– функции, которые возвращают объекты заданного типа;

– указатели на объекты данного типа;

– структуры, содержащие последовательность объектов различных типов;

– объединения, способные содержать один из нескольких объектов различных типов.

Вообще говоря, эти методы построения объектов могут применяться рекурсивно.

9.6 Объекты и L-значения

Объект является доступным обработке участком памяти; L-значение – это выражение, ссылающееся на объект. Очевидным примером выражения L-значения является идентификатор. Существуют операции, результатом которых являются L-значения; если, например, `E` – выражение указанного типа, то `*E` является выражением L-значения, ссылающимся на объект `E`. Название «L-значение» происходит от выражения присваивания

[1] На PDP-11 длинные величины без знака не предусмотрены.

[2] В некоторых реализациях могут быть синонимами.

`E1=E2`, в котором левая часть должна быть выражением L-значения. При последующем обсуждении каждой операции будет указываться, ожидает ли она операндов L-значения и выдает ли она L-значение.

9.7 Преобразования

Ряд операций может в зависимости от своих операндов вызывать преобразование значение операнда из одного типа в другой. В этом разделе объясняются результаты, которые следует ожидать от таких преобразований. В п.9.7.6 подводятся итоги преобразований, требуемые большинством обычных операций; эти сведения дополняются необходимым образом при обсуждении каждой операции.

9.7.1 Символы и целые

Символ или короткое целое можно использовать всюду, где можно использовать целое. Во всех случаях значение преобразуется к целому. Преобразование более короткого целого к более длинному всегда сопровождается знаковым расширением; целые являются величинами со знаком. Осуществляется или нет знаковое расширение для символов, зависит от используемой машины, но гарантируется, что член стандартного набора символов неотрицателен. Из всех машин, рассматриваемых в этом руководстве, только PDP-11 осуществляет знаковое расширение. Область значений символьных переменных на PDP-11 меняется от -128 до 127; символы из набора ASCII имеют положительные значения. Символьная константа, заданная с помощью восьмеричной условной последовательности, подвергается знаковому расширению и может оказаться отрицательной; например, '\377' имеет значение -1.

Когда более длинное целое преобразуется в более короткое или в `char`, оно обрезается слева; лишние биты просто отбрасываются.

9.7.2 Типы float и double

Вся плавающая арифметика в **C** выполняется с двойной точностью каждый раз, когда объект типа `float` появляется в выражении, он удлиняется до `double` посредством добавления нулей в его дробную часть. Когда объект типа `double` должен быть преобразован к типу `float`, например, при присваивании, перед усечением `double` округляется до длины `float`.

9.7.3 Плавающие и целочисленные величины

Преобразование плавающих значений к целочисленному типу имеет тенденцию быть до некоторой степени машинно-зависимым; в частности направление усечения отрицательных чисел меняется от машине к машине. Результат не определен, если значение не помещается в предоставляемое пространство.

Преобразование целочисленных значений в плавающие выполняется без осложнений. Может произойти некоторая потеря точности, если для результата не содержится достаточного количества битов.

9.7.4 Указатели и целые

Целое или длинное целое может быть прибавлено к указателю или вычтено из него; в этом случае первая величина преобразуется так, как указывается в разделе описания операции сложения.

Два указателя на объекты одинакового типа могут быть вычтены; в этом случае результат преобразуется к целому, как указывается в разделе описания операции вычитания.

9.7.5 Целое без знака

Всякий раз, когда целое без знака объединяется с простым целым, простое целое преобразуется в целое без знака и результат оказывается целым без знака. Значением является наименьшее целое без знака, соответствующее целому со знаком (по модулю 2^n где n – размер слова). В двоичном дополнительном представлении это преобразование является чисто умозрительным и не изменяет фактическую комбинацию битов.

Когда целое без знака преобразуется к типу `long`, значение результата совпадает со значением целого без знака. Таким образом, это преобразование сводится к добавлению нулей слева.

9.7.6 Арифметические преобразования

Подавляющее большинство операций вызывает преобразование и определяет типы результата аналогичным образом. Приводимая ниже схема в дальнейшем будет называться «обычными арифметическими преобразованиями».

– Сначала любые операнды типа `char` или `short` преобразуются в `int`, а любые операнды типа `float` преобразуются в `double`.

– Затем, если какой-либо операнд имеет тип `double`, то другой преобразуется к типу `double`, и это будет типом результата.

– В противном случае, если какой-либо операнд имеет тип `long`, то другой операнд преобразуется к типу `long`, и это и будет типом результата.

– В противном случае, если какой-либо операнд имеет тип `unsigned`, то другой операнд преобразуется к типу `unsigned`, и это будет типом результата.

– В противном случае оба операнда будут иметь тип `int`, и это будет типом результата.

9.8 Выражения

Старшинство операций в выражениях совпадает с порядком следования основных подразделов настоящего раздела, начиная с самого высокого уровня старшинства. Так, например, выражениями, указываемыми в качестве операндов операции + (п.9.8.4), являются выражения, определенные в п.9.8.1. Внутри каждого подраздела операции имеет одинаковое старшинство. В каждом подразделе для описываемых там операций указывается их ассоциативность слева или справа. Старшинство и ассоциативность всех операций в выражениях резюмируются в грамматической сводке в п.9.11.

В противном случае порядок вычислений выражений не определен. В частности, компилятор считает себя в праве вычислять подвыражения в том порядке, который он находит наиболее эффективным, даже если эти подвыражения приводят к побочным эффектам. Порядок, в котором происходят побочные эффекты, не специфицируется. Выражения, включающие коммутативные и ассоциативные операции (*, +, &, !, ^), могут быть переупорядочены произвольным образом даже при наличии круглых скобок; чтобы вынудить определенный порядок вычислений, в этом случае необходимо использовать явные промежуточные переменные.

При вычислении выражений обработка переполнения и проверка при делении являются машинно-зависимыми. Все существующие реализации языка **C** игнорируют переполнение целых; обработка ситуаций при делении на 0 и при всех особых случаях с плавающими числами меняется от машины к машине и обычно выполняется с помощью библиотечной функции.

9.8.1 Первичные выражения

Первичные выражения, включающие « . », ->, индексацию и обращения к функциям, группируются слева направо.

Первичное выражение:
    идентификатор
    константа
    строка
    (выражение)
    первичное-выражение[выражение]
    первичное-выражение(*список-выражений*)
    первичное-L-значение . идентификатор
    первичное-выражение -> идентификатор

список-выражений:
    выражение
    список-выражений, выражение

Идентификатор является первичным выражением при условии, что он описан подходящим образом, как это обсуждается ниже. Тип идентификатора определяется его описанием. Если, однако, типом идентификатора является «массив T», то значением

выражения, состоящего из этого идентификатора, является указатель на первый объект в этом массиве, а типом выражения будет «указатель на T». Более того, идентификатор массива не является выражением L-значения. Подобным образом идентификатор, который описан как «функция, возвращающая T», за исключением того случая, когда он используется в позиции имени функции при обращении, преобразуется в «указатель на функцию, которая возвращает T».

Константа является первичным выражением. В зависимости от ее формы типом константы может быть `int`, `long` или `double`.

Строка является первичным выражением. Исходным ее типом является «массив символов»; но следуя тем же самым правилам, которые приведены выше для идентификаторов, он модифицируется в «указатель на символы», и результатом является указатель на первый символ строки.[1]

Выражение в круглых скобках является первичным выражением, тип и значение которого идентичны типу и значению этого выражения без скобок. Наличие круглых скобок не влияет на то, является ли выражение L-значением или нет.

Первичное выражение, за которым следует выражение в квадратных скобках, является первичным выражением. Интуитивно ясно, что это выражение с индексом. Обычно первичное выражение имеет тип «указатель на T», индексное выражение имеет тип `int`, а типом результата является «T». Выражение `E1[E2]` по определению идентично выражению `*((E1) + (E2))`. Все, что необходимо для понимания этой записи, содержится в этом разделе; вопросы, связанные с понятием идентификаторов и операций `*` и `+` рассматриваются в п.п.9.8.1, 9.8.2 и 9.8.4 соответственно; выводы суммируются ниже в п.9.15.3.

Обращение к функции является первичным выражением, за которым следует заключенный в круглые скобки возможно пустой список выражений, разделенных запятыми, которые и представляют собой фактические аргументы функции. Первичное выражение должно быть типа «функция, возвращающая T», а результат обращения к функции имеет тип «T». Как указывается ниже, ранее не встречавшийся идентификатор, за которым непосредственно следует левая круглая скобка, считается описанным по контексту, как представляющий функцию, возвращающую целое; следовательно чаще всего встречающийся случай функции, возвращающей целое значение, не нуждается в описании.

Перед обращением любые фактические аргументы типа `float` преобразуются к типу `double`, любые аргументы типа `char` или `short` преобразуются к типу `int`, и, как обычно, имена массивов преобразуются в указатели. Никакие другие преобразования не выполняются автоматически; в частности, не сравнивает типы фактических аргументов с типами формальных аргументов. Если преобразование необходимо, используйте явный перевод типа (cast); см. п.п.9.8.2, 9.9.7.

При подготовке к вызову функции делается копия каждого фактического параметра; таким образом, все передачи аргументов в языке **C** осуществляются строго по значению. Функция может изменять значения своих формальных параметров, но эти изменения не влияют на значения фактических параметров. С другой стороны имеется возможность передавать указатель при таком условии, что функция может изменять значение объек-

[1] Имеется исключение в некоторых инициализаторах; см. п.9.9.6.

та, на который этот указатель указывает. Порядок вычисления аргументов в языке не определен; обратите внимание на то, что различные компиляторы вычисляют по разному.

Допускаются рекурсивные обращения к любой функции.

Первичное выражение, за которым следует точка и идентификатор, является выражением. Первое выражение должно быть L-значением, именующим структуру или объединение, а идентификатор должен быть именем члена структуры или объединения. Результатом является L-значение, ссылающееся на поименованный член структуры или объединения.

Первичное выражение, за которым следует стрелка (составленная из знаков − и >) и идентификатор, является выражением. Первое выражение должно быть указателем на структуру или объединение, а идентификатор должен именовать член этой структуры или объединения. Результатом является L-значение, ссылающееся на поименованный член структуры или объединения, на который указывает указательное выражение.

Следовательно, выражение `E1->MOS` является тем же самым, что и выражение `(*E1).MOS`. Структуры и объединения рассматриваются в п.9.9.5. Приведенные здесь правила использования структур и объединений не навязываются строго, для того чтобы иметь возможность обойти механизм типов. См. п.9.15.1.

9.8.2 Унарные операции

Выражение с унарными операциями группируется справа налево.

Унарное-выражение:
- * выражение
- & L-значение
- − выражение
- ! Выражение
- ~ выражение
- ++ L-значение
- −− L-значение
- L-значение ++
- L-значение −−
- (имя-типа) выражение
- sizeof (выражение)
- sizeof (имя-типа)

Унарная операция * означает косвенную адресацию: выражение должно быть указателем, а результатом является L-значение, ссылающееся на тот объект, на который указывает выражение. Если типом выражения является «указатель на T», то типом результата будет «T».

Результатом унарной операции & является указатель на объект, к которому ссылается L-значение. Если L-значение имеет тип «T», то типом результата будет «указатель на T».

Результатом унарной операции − (минус) является ее операнд, взятый с противоположным знаком. Для величины типа `unsigned` результат получается вычитанием ее значения из 2^n, где n – число битов в `int`. Унарной операции + (плюс) не существует.

Результатом операции логического отрицания ! является 1, если значение ее операнда равно 0, и 0, если значение ее операнда отлично от нуля. Результат имеет тип `int`. Эта операция применима к любому арифметическому типу или указателям.

Операция ~ дает обратный код, или дополнение до единицы, своего операнда. Выполняются обычные арифметические преобразования. Операнд должен быть целочисленного типа.

Объект, на который ссылается операнд L-значения префиксной операции ++, увеличивается. Значением является новое значение операнда, но это не L-значение. Выражение `++x` эквивалентно `x+=1`. Информацию о преобразованиях смотри в разборе операции сложения (п.9.8.4) и операции присваивания (п.9.8.14).

Префиксная операция `--` аналогична префиксной операции ++, но приводит к уменьшению своего операнда L-значения.

При применении постфиксной операции ++ к L-значению результатом является значение объекта, на который ссылается L-значение. После того, как результат принят к сведению, объект увеличивается точно таким же образом, как и в случае префиксной операции ++. Результат имеет тот же тип, что и выражение L-значения.

При применении постфиксной операции `--` к L-значению результатом является значение объекта, на который ссылается L-значение. После того, как результат принят к сведению, объект уменьшается точно таким же образом, как и в случае префиксной операции `--`. Результат имеет тот же тип, что и выражение L-значения.

Заключенное в круглые скобки имя типа данных,стоящее перед выражением, вызывает преобразование значения этого выражения к указанному типу. Эта конструкция называется перевод (cast). Имена типов описываются в п.9.9.7.

Операция `sizeof` выдает размер своего операнда в байтах.[1] При применении к массиву результатом является полное число байтов в массиве. Размер определяется из описаний объектов в выражении. Это выражение семантически является целой константой и может быть использовано в любом месте, где требуется константа. Основное применение эта операция находит при связях с процедурами, подобным распределителям памяти, и в системах ввода-вывода.

Операция `sizeof` может быть также применена и к заключенному в круглые скобки имени типа. В этом случае она выдает размер в байтах объекта указанного типа.

Конструкция `sizeof(тип)` рассматривается как целое, так что выражение `sizeof(тип)-2` эквивалентно выражению `(sizeof(тип))-2`.

9.8.3 Мультипликативные операции

Мультипликативные операции *, /, и % группируются слева направо. Выполняются обычные арифметические преобразования.

Мультипликативное-выражение:
    выражение * выражение
    выражение / выражение

[1] Понятие байт в языке не определено, разве только как значение операции `sizeof`. Однако во всех существующих реализациях байтом является пространство, необходимое для хранения объекта типа `char`.

выражение % выражение

Бинарная операция * означает умножение. Операция * ассоциативна, и выражения с несколькими умножениями на одном и том же уровне могут быть перегруппированы компилятором.

Бинарная операция / означает деление. При делении положительных целых осуществляется усечение по направлению к нулю, но если один из операндов отрицателен, то форма усечения зависит от используемой машины. На всех машинах, охватываемых настоящим руководством, остаток имеет тот же знак, что и делимое. Всегда справедливо, что `(a/b)*b+a%b` равно a (если b не равно 0).

Бинарная операция % выдает остаток от деления первого выражения на второе. Выполняются обычные арифметические преобразования. Операнды не должны быть типа `float`.

9.8.4 Аддитивные операции

Аддитивные операции + и − группируются слева направо. выполняются обычные арифметические преобразования. Для каждой операции имеются некоторые дополнительные возможности, связанные с типами операндов.

Аддитивное-выражение:
выражение + выражение
выражение − выражение

Результатом операции + является сумма операндов. Можно складывать указатель на объект в массиве и значение любого целочисленного типа. Во всех случаях последнее преобразуется в адресное смещение посредством умножения его на длину объекта, на который указывает этот указатель. Результатом является указатель того же самого типа, что и исходный указатель, который указывает на другой объект в том же массиве, смещенный соответствующим образом относительно первоначального объекта. Таким образом, если p является указателем объекта в массиве, то выражение p+1 является указателем на следующий объект в этом массиве.

Никакие другие комбинации типов для указателей не разрешаются.

Операция + ассоциативна, и выражение с несколькими сложениями на том же самом уровне могут быть переупорядочены компилятором.

Результатом операции − является разность операндов. Выполняются обычные арифметические преобразования. Кроме того, из указателя может быть вычтено значение любого целочисленного типа, причем, проводятся те же самые преобразования, что и при операции сложения.

Если вычитаются два указателя на объекты одинакового типа, то результат преобразуется (делением на длину объекта) к типу `int`, представляя собой число объектов, разделяющих указываемые объекты. Если эти указатели не на объекты из одного и того же массива, то такое преобразование, вообще говоря, даст неожиданные результаты, потому что даже указатели на объекты одинакового типа не обязаны отличаться на величину, кратную длине объекта.

9.8.5 Операции сдвига

Операции сдвига ≪ и ≫ группируются слева направо. Для обеих операций проводятся обычные арифметические преобразования их операндов, каждый из которых должен быть целочисленного типа. Затем правый операнд преобразуется к типу `int`; результат имеет тип левого операнда. Результат не определен, если правый операнд отрицателен или больше или равен, чем длина объекта в битах.

Выражение-сдвига:
    выражение ≪ выражение
    выражение ≫ выражение

Значением выражения `E1≪E2` является `E1` (интерпретируемое как комбинация битов), сдвинутое влево на `E2` битов; освобождающиеся биты заполняются нулем. Значением выражения `E1≫E2` является `E1`, сдвинутое вправо на `E2` битовых позиций. Если `E1` имеет тип `unsigned`, то сдвиг вправо гарантированно будет логическим (заполнение нулем); в противном случае сдвиг может быть (и так и есть на PDP-11) арифметическим (освобождающиеся биты заполняются копией знакового бита).

9.8.6 Операции отношения

Операции отношения группируются слева направо, но этот факт не очень полезен; выражение `a<b<c` не означает того, что оно казалось бы должно означать.

Выражение-отношения:
    выражение < выражение
    выражение > выражение
    выражение <= выражение
    выражение >= выражение

Операции < (меньше), > (больше), <= (меньше или равно) и >= (больше или равно) все дают 0, если указанное отношение ложно, и 1, если оно истинно. Результат имеет тип `int`. Выполняются обычные арифметические преобразования. Могут сравниваться два указателя; результат зависит от относительного расположения указываемых объектов в адресном пространстве. Сравнение указателей переносимо только в том случае, если указатели указывают на объекты из одного и того же массива.

9.8.7 Операции равенства

Выражение-равенства:
    выражение == выражение
    выражение != выражение

Операции == (равно) и != (не равно) в точности аналогичны операциям отношения, за исключением того, что они имеют более низкий уровень старшинства. (Поэтому значение

выражения `a<b==c<d` равно 1 всякий раз, когда выражение `a<b` и `c<d` имеют одинаковое значение истинности).

Указатель можно сравнивать с целым, но результат будет машинно-независимым только в том случае, если целым является константа 0. Гарантируется, что указатель, которому присвоено значение 0, не указывает ни на какой объект и на самом деле оказывается равным 0; общепринято считать такой указатель нулем.

9.8.8 Побитовая операция «И»

Выражение-и:
    выражение & выражение

Операция & является ассоциативной, и включающие & выражения могут быть переупорядочены. Выполняются обычные арифметические преобразования; результатом является побитовая функция «и» операндов. Эта операция применима только к операндам целочисленного типа.

9.8.9 Побитовая операция исключающего «ИЛИ»

Выражение-исключающего-или:
    выражение ^ выражение

Операция ^ является ассоциативной, и включающие ^ выражения могут быть переупорядочены. Выполняются обычные арифметические преобразования; результатом является побитовая функция исключающего «или» операндов. Операция применима только к операндам целочисленного типа.

9.8.10 Побитовая операция включающего «ИЛИ»

Выражение-включающего-или:
    выражение | выражение

Операция | является ассоциативной, и содержащие | выражения могут быть переупорядочены. Выполняются обычные арифметические преобразования; результатом является побитовая функция включающего «или» операндов. Операция применима только к операндам целочисленного типа.

9.8.11 Логическая операция «И»

Выражение-логического-и:
    выражение && выражение

Операция && группируется слева направо. Она возвращает 1, если оба ее операнда отличны от нуля, и 0 в противном случае. В отличие от & операция && гарантирует

вычисление слева направо; более того, если первый операнд равен 0, то значение второго операнда вообще не вычисляется.

Операнды не обязаны быть одинакового типа, но каждый из них должен быть либо одного из основных типов, либо указателем. Результат всегда имеет тип `int`.

9.8.12 Операция логического «ИЛИ»

Выражение-логического-или:
    выражение || выражение

Операция || группируется слева направо. Она возвращает 1, если один из операндов отличен от нуля, и 0 в противном случае. В отличие от операции | операция || гарантирует вычисление слева направо; более того, если первый операнд отличен от нуля, то значение второго операнда вообще не вычисляется.

Операнды не обязаны быть одинакового типа, но каждый из них должен быть либо одного из основных типов, либо указателем. Результат всегда имеет тип `int`.

9.8.13 Условная операция

Условное-выражение:
    выражение ? выражение : выражение

Условные выражения группируются слева направо. Вычисляется значение первого выражения, и если оно отлично от нуля, то результатом будет значение второго выражения; в противном случае результатом будет значение третьего выражения. Если это возможно, проводятся обычные арифметические преобразования, с тем, чтобы привести второе и третье выражения к общему типу; в противном случае, если оба выражения являются указателями одинакового типа, то результат имеет тот же тип; в противном случае одно выражение должно быть указателем, а другое – константой 0, и результат будет иметь тип указателя. Вычисляется только одно из второго и третьего выражений.

9.8.14 Операция присваивания

Имеется ряд операций присваивания, каждая из которых группируется слева направо. Все операции требуют в качестве своего левого операнда L-значение, а типом выражения присваивания является тип его левого операнда. Значением выражения присваивания является значение, хранимое в левом операнде после того, как присваивание уже будет произведено. Две части составной операции присваивания являются отдельными лексемами.

Выражение-присваивания:
L-значение = выражение
L-значение += выражение
L-значение −= выражение
L-значение *= выражение
L-значение /= выражение
L-значение %= выражение
L-значение >>= выражение
L-значение <<= выражение
L-значение &= выражение
L-значение ^= выражение
L-значение |= выражение

Когда производится простое присваивание =, значение выражения заменяет значение объекта, на которое ссылается L-значение. Если оба операнда имеют арифметический тип, то перед присваиванием правый операнд преобразуется к типу левого операнда.

О свойствах выражения вида `E1 оп= E2`, где оп – одна из перечисленных выше операций, можно сделать вывод, если учесть, что оно эквивалентно выражению `E1 = E1 оп (E2)`; однако выражение `E1` вычисляется только один раз. В случае операций += и −= левый операнд может быть указателем, причем при этом (целочисленный) правый операнд преобразуется таким образом, как объяснено в п.9.8.4; все правые операнды и все отличные от указателей левые операнды должны иметь арифметический тип.

Используемые в настоящее время компиляторы допускают присваивание указателя целому, целого указателю и указателя указателю другого типа. Такое присваивание является чистым копированием без каких-либо преобразований. Такое употребление операций присваивания является непереносимым и может приводить к указателям, которые при использовании вызывают ошибки адресации. Тем не менее гарантируется, что присваивание указателю константы 0 дает нулевой указатель, который можно отличать от указателя на любой объект.

9.8.15 Операция запятая

Выражение-с-запятой:
выражение, выражение

Пара выражений, разделенных запятой, вычисляется слева направо и значение левого выражения отбрасывается. Типом и значением результата является тип и значение правого операнда. Эта операция группируется слева направо. В контексте, где запятая имеет специальное значение, как, например, в списке фактических аргументов функций (п.9.8.1) или в списках инициализаторов (п.9.9.6), операция запятая, может появляться только в круглых скобках; например, функция `f(a,(t=3,t+2),c)` имеет три аргумента, второй из которых имеет значение 5.

9.9 Описания

Описания используются для указания интерпретации, которую язык **C** будет давать каждому идентификатору; они не обязательно резервируют память, соответствующую идентификатору. Описания имеют форму:

Описание:
спецификаторы-описания список-описателей;

Описатели в списке описателей содержат описываемые идентификаторы. Спецификаторы описания представляют собой последовательность спецификаторов типа и спецификаторов класса памяти.

Спецификаторы-описания:
спецификатор-типа спецификаторы-описания;
спецификатор-класса-памяти спецификатор-описания;

Список должен быть самосогласованным в смысле, описываемом ниже.

9.9.1 Спецификаторы класса памяти

Ниже перечисляются спецификаторы класса памяти:

Спецификатор-класса-памяти:
`auto`
`static`
`extern`
`register`
`typedef`

Спецификатор `typedef` не реализует памяти и называется «спецификатором класса памяти» только по синтаксическим соображениям; это обсуждается в п.9.9.8. Смысл различных классов памяти был обсужден в п.9.5.

Описания `auto`, `static` и `register` служат также в качестве определений в том смысле, что они вызывают резервирование нужного количества памяти. В случае `extern` должно присутствовать внешнее определение (п.9.11) указываемых идентификаторов где-то вне функции, в которой они описаны.

Описание `register` лучше всего представлять себе как описание `auto` вместе с намеком компилятору, что описанные таким образом переменные будут часто использоваться. Эффективны только несколько первых таких описаний. Кроме того, в регистрах могут храниться только переменные определенных типов; на PDP-11 это `int`, `char` или указатель. Существует и другое ограничение на использование регистровых переменных: к ним нельзя применять операцию взятия адреса &. При разумном использовании регистровых описаний можно ожидать получения меньших по размеру и более быстрых программ, но улучшение в будущем генерирования кодов может сделать их ненужными.

Описание может содержать не более одного спецификатора класса памяти. Если описание не содержит спецификатора класса памяти, то считается, что он имеет значение `auto`, если описание находится внутри некоторой функции, и `extern` в противном случае. Исключение: функции никогда не бывает автоматическими.

9.9.2 Спецификаторы типа

Ниже перечисляются спецификаторы типа.

Спецификатор-типа:
    `char`
    `short`
    `int`
    `long`
    `unsigned`
    `float`
    `double`
    спецификатор-структуры-или-объединения
    определяющее-тип-имя

Слова `long`, `short` и `unsigned` можно рассматривать как прилагательные; допустимы следующие комбинации:

```
short int
long int
unsigned int
long float
```

Последняя комбинация означает то же, что и `double`. В остальном описание может содержать не более одного спецификатора типа. Если описание не содержит спецификатора типа, то считается, что он имеет значение `int`.

Спецификаторы структур и объединений обсуждаются в п.9.9.5; Описания с определяющими тип именами `typedef` обсуждаются в п.9.9.8.

9.9.3 Описатели

Входящий в описание список описателей представляет собой последовательность разделенных запятыми описателей, каждый из которых может иметь инициализатор.

Список-описателей:
    инициализируемый-описатель
    инициализируемый-описатель, список-описателей

    инициализируемый-описатель:
        описатель-инициализатор

Инициализаторы описываются в п.9.9.6. Спецификаторы и описания указывают тип и класс памяти объектов, на которые ссылаются описатели. Описатели имеют следующий синтаксис:

описатель:
    идентификатор
    (описатель)

```
* описатель
описатель ()
описатель [константное-выражение]
```

Группирование такое же как и в выражениях.

9.9.4 Смысл описателей

Каждый описатель рассматривается как утверждение того, что когда конструкция той же самой формы, что и описатель, появляется в выражении, то она выдает объект указанного типа и указанного класса памяти. Каждый описатель содержит ровно один идентификатор; это именно тот идентификатор, который и описывается.

Если в качестве описателя появляется просто идентификатор, то он имеет тип, указываемый в специфицирующем заголовке описания.

Описатель в круглых скобках идентичен описателю без круглых скобок, но круглые скобки могут изменять связи в составных описателях. Примеры смотри ниже.

Представим себе описание

```
T descriptor;
```

где T – спецификатор типа (подобный `int` и т.д.), а descriptor – описатель. Предположим, что это описание приводит к тому, что соответствующий идентификатор имеет тип «что-то T», где «что-то» пусто, если descriptor просто отдельный идентификатор (так что тип x в `int x` просто `int`). Тогда, если descriptor имеет форму `*x` то содержащийся идентификатор будет иметь тип «указатель на T».

Если descriptor имеет форму `x()` то содержащийся идентификатор имеет тип «функция, возвращающая T».

Если descriptor имеет форму `x[константное-выражение]` или `x[ ]` то содержащийся идентификатор имеет тип «массив T». В первом случае константным выражением является выражение, значение которого можно определить во время компиляции и которое имеет тип `int`.[1] Когда несколько спецификаций вида «массив» оказываются примыкающими, то создается многомерный массив; константное выражение, задающее границы массивов, может отсутствовать только у первого члена этой последовательности. Такое опускание полезно, когда массив является внешним и его фактическое определение, которое выделяет память, приводится в другом месте. Первое константное выражение может быть опущено также тогда, когда за описателем следует инициализация. В этом случае размер определяется по числу приведенных инициализируемых элементов.

Массив может быть образован из элементов одного из основных типов, из указателей, из структур или объединений или из других массивов (чтобы образовать многомерный массив).

Не все возможности, которые разрешены с точки зрения указанного выше синтаксиса, фактически допустимы. Имеются следующие ограничения: функции не могут возвращать массивы, структуры, объединения или функции, хотя они могут возвращать указа-

[1] Точное определение константного выражения дано в п.9.16.

тели на такие вещи; не существует массивов функций, хотя могут быть массивы указателей на функции. Аналогично, структуры или объединения не могут содержать функцию, но они могут содержать указатель на функцию.

В качестве примера рассмотрим описание

```
int i, *ip, f(), *fip(), (*pfi) ();
```

в котором описывается целое i, указатель на целое ip, функция f, возвращающая целое, функция `fip`, возвращающая указатель на целое, и `pfi` – указатель на функцию, которая возвращает целое. Особенно полезно сравнить два последних описателя. Связь в `*fip()` можно представить в виде `*(fip())`, так что описанием предполагается, а такой же конструкцией в выражении требуется обращение к функции `fip` и последующее использование косвенной адресации для выдачи с помощью полученного результата (указателя) целого. В описателе `(*pfi)()` дополнительные скобки необходимы, поскольку они точно так же, как и в выражении, указывают, что косвенная адресация через указатель на функцию выдает функцию, которая затем вызывается; эта вызванная функция возвращает целое.

В качестве другого примера приведем описание

```
float fa[17], *afp[17];
```

в котором описывается массив чисел типа `float` и массив указателей на числа типа `float`. Наконец,

```
static int x3d[3][5][7];
```

описывает статический трехмерный массив целых размером `3*5*7`. Более подробно, `x3d` является массивом из трех элементов; каждый элемент является массивом пяти массивов; каждый последний массив является массивом из семи целых. Каждое из выражений `x3d`, `x3d[i]`, `x3d[i][j]` и `x3d[i][j][k]` может разумным образом появляться в выражениях. Первые три имеют тип «массив», последнее имеет тип `int`.

9.9.5 Описание структур и объединений

Структура – это объект, состоящий из последовательности именованных членов. Каждый член может быть произвольного типа. Объединение – это объект, который в данный момент может содержать любой из нескольких членов. Спецификаторы структуры и объединения имеют одинаковую форму.

Спецификатор-структуры-или-объединения:
    структура-или-объединение {*список-описаний-структуры*}
        идентификатор структуры-или-объединения;

    Структура-или-объединение:
        `struct`
        `union`

Список-описаний-структуры является последовательностью описаний членов структуры (объединения):

Список-описаний-структуры:
    описание-структуры
    описание-структуры список-описаний-структуры

    описание-структуры:
        спецификатор-типа список-описателей-структуры

    список-описателей-структуры:
        описатель-структуры
        описатель-структуры, список-описателей-структуры

В обычном случае описатель структуры является просто описателем члена структуры или объединения. Член структуры может также состоять из специфицированного числа битов. Такой член называется также полем; его длина отделяется от имени поля двоеточием.

Описатель-структуры:
    описатель
    описатель: константное выражение
    : константное выражение

Внутри структуры описанные в ней объекты имеют адреса, которые увеличиваются в соответствии с чтением их описаний слева направо. Каждый член структуры, который не является полем, начинается с адресной границы, соответствующей его типу; следовательно в структуре могут оказаться неименованные дыры. Члены, являющиеся полями, помещаются в машинные целые; они не перекрывают границы слова. Поле, которое не умещается в оставшемся в данном слове пространстве, помещается в следующее слово. Поля выделяются справа налево на PDP-11 и слева направо на других машинах.

Описатель структуры, который не содержит описателя, а только двоеточие и ширину, указывает неименованное поле, полезное для заполнения свободного пространства с целью соответствия задаваемых извне схемам. Специальный случай неименованного поля с шириной 0 используется для указания о выравнивании следующего поля на границу слова. При этом предполагается, что «следующее поле» действительно является полем, а не обычным членом структуры, поскольку в последнем случае выравнивание осуществляется автоматически.

Сам язык не накладывает ограничений на типы объектов, описанных как поля, но от реализаций не требуется обеспечивать что-либо отличное от целых полей. Более того, даже поля типа `int` могут рассматриваться как не имеющие знака. На PDP-11 поля не имеют знака и могут принимать только целые значения. Во всех реализациях отсутствуют массивы полей и к полям не применима операция взятия адреса &, так что не существует и указателей на поля.

Объединение можно представить себе как структуру, все члены которой начинаются со смещения 0 и размер которой достаточен, чтобы содержать любой из ее членов. В каждый момент объединение может содержать не более одного из своих членов.

Спецификатор структуры или объединения во второй форме, т.е. один из

`struct` идентификатор {список-описаний-структуры};
`union` идентификатор {список-описаний-структуры};

описывает идентификатор в качестве ярлыка структуры (или ярлыка объединения), специфицированной этим списком. Последующее описание может затем использовать третью форму спецификатора, один из

`struct` идентификатор
`union` идентификатор

Ярлыки структур дают возможность определения структур, которые ссылаются на самих себя; они также позволяют неоднократно использовать приведенную только один раз длинную часть описания. Запрещается описывать структуру или объединение, которые содержат образец самого себя, но структура или объединение могут содержать указатель на структуру или объединение такого же вида, как они сами.

Имена членов и ярлыков могут совпадать с именами обычных переменных. Однако имена ярлыков и членов должны быть взаимно различными.

Две структуры могут иметь общую начальную последовательность членов; это означает, что тот же самый член может появиться в двух различных структурах, если он имеет одинаковый тип в обеих структурах и если все предыдущие члены обеих структур одинаковы.[1]

Вот простой пример описания структуры:

```
struct tnode {
  char tword[20];
  int count;
  struct tnode *left;
  struct tnode *right;
};
```

Такая структура содержит массив из 20 символов, целое и два указателя на подобные структуры. Как только приведено такое описание, описание

```
struct tnode s, *sp;
```

говорит о том, что s является структурой указанного вида, а sp является указателем на структуру указанного вида. При наличии этих описаний выражение

```
sp->count
```

ссылается к полю count структуры, на которую указывает sp; выражение

```
s.left
```

ссылается на указатель левого поддерева в структуре s, а выражение

```
s.right->tword[0]
```

ссылается на первый символ члена tword правого поддерева из s.

[1] Фактически компилятор только проверяет, что имя в двух различных структурах имеет одинаковый тип и одинаковое смещение, но если предшествующие члены отличаются, то конструкция оказывается непереносимой.

9.9.6 Инициализация

Описатель может указывать начальное значение описываемого идентификатора. Инициализатор состоит из выражения или заключенного в фигурные скобки списка значений, перед которыми ставится знак =.

```
Инициализатор:
    = выражение
    = {список-инициализатора}

    список-инициализатора:
        выражение
        список-инициализатора, список-инициализатора
```

Все выражения, входящие в инициализатор статической или внешней переменной, должны быть либо константными выражениями, описываемыми в п.9.16, либо выражениями, которые сводятся к адресу ранее описанной переменной, возможно смещенному на константное выражение. Автоматические и регистровые переменные могут быть инициализированы произвольными выражениями, включающими константы и ранее описанные переменные и функции.

Гарантируется, что неинициализированные статические и внешние переменные получают в качестве начальных значений 0; неинициализированные автоматические и регистровые переменные в качестве начальных значений содержат мусор.

Когда инициализатор применяется к скаляру (указателю или объекту арифметического типа), то он состоит из одного выражения, возможно заключенного в фигурные скобки. Начальное значение объекта находится из выражения; выполняются те же самые преобразования, что и при присваивании.

Когда описываемая переменная является агрегатом (структурой или массивом), то инициализатор состоит из заключенного в фигурные скобки и разделенного запятыми списка инициализаторов для членов агрегата. Этот список составляется в порядке возрастания индекса или в соответствии с порядком членов. Если агрегат содержит подагрегаты, то это правило применяется рекурсивно к членам агрегата. Если количество инициализаторов в списке оказывается меньше числа членов агрегата, то оставшиеся члены агрегата заполняются нулями. Запрещается инициализировать объединения или автоматические агрегаты.

Фигурные скобки могут быть описаны следующим образом. Если инициализатор начинается с левой фигурной скобки, то последующий разделенный запятыми список инициализаторов инициализирует члены агрегата; будет ошибкой, если в списке окажется больше инициализаторов, чем членов агрегата. Если однако инициализатор не начинается с левой фигурной скобки, то из списка берется только нужное для членов данного агрегата число элементов; оставшиеся элементы используются для инициализации следующего члена агрегата, частью которого является настоящий агрегат.

Последнее сокращение допускает возможность инициализации массива типа `char` с помощью строки. В этом случае члены массива последовательно инициализируются символами строки.

Например,

```
int x[] = { 1, 3, 5 };
```

описывает и инициализирует x как одномерный массив; поскольку размер массива не специфицирован, а список инициализатора содержит три элемента, считается, что массив состоит из трех членов.

Вот пример инициализации с полным использованием фигурных скобок:

```
float y[4][3] = {
  {1, 3, 5},
  {2, 4, 6},
  {3, 5, 7},
};
```

Здесь 1, 3 и 5 инициализируют первую строку массива y[0], а именно y[0][0], y[0][1] и y[0][2]. Аналогичным образом следующие две строчки инициализируют y[1] и y[2]. Инициализатор заканчивается преждевременно, и, следовательно массив y[3] инициализируется нулями. В точности такого же эффекта можно было бы достичь, написав

```
float y[4][3] = {
  1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Инициализатор для y начинается с левой фигурной скобки, но инициализатора для y[0] нет. Поэтому используется 3 элемента из списка. Аналогично следующие три элемента используются последовательно для y[1] и y[2]. Следующее описание

```
float y[4][3] = {
  {1}, {2}, {3}, {4}
};
```

инициализирует первый столбец y (если его рассматривать как двумерный массив), а остальные элементы заполняются нулями.

И наконец, описание

```
char msg[] = "syntax error on line %s\n";
```

демонстрирует инициализацию элементов символьного массива с помощью строки.

9.9.7 Имена типов

В двух случаях (для явного указания типа преобразования в конструкции перевода и для аргументов операции `sizeof`) желательно иметь возможность задавать имя типа данных. Это осуществляется с помощью «имени типа», которое по существу является описанием объекта такого типа, в котором опущено имя самого объекта.

Имя типа:
спецификатор-типа абстрактный-описатель

абстрактный-описатель:
пусто
(абстрактный-описатель)
* абстрактный описатель
абстрактный-описатель ()
абстрактный-описатель [константное выражение]

Во избежании двусмысленности в конструкции (абстрактный описатель) требуется, чтобы абстрактный описатель был не пуст. При этом ограничении возможно однозначно определить то место в абстрактном описателе, где бы появился идентификатор, если бы эта конструкция была описателем в описании. Именованный тип совпадает тогда с типом гипотетического идентификатора. Например, имена типов именуют соответственно типы:

`int`	целый
`int *`	указатель на целое
`int *[3]`	массив из трех указателей на целое
`int (*)[3]`	указатель на массив из трех целых
`int *()`	функция, возвращающая указатель на целое
`int (*)()`	указатель на функцию, возвращающую целое

9.9.8 typedef

Описания, в которых «класс памяти» специфицирован как `typedef`, не вызывают выделения памяти. Вместо этого они определяют идентификаторы, которые позднее можно использовать так, словно они являются ключевыми словами, имеющими основные или производные типы.

`typedef` определяющее-тип-имя идентификатор

В пределах области действия описания со спецификатором `typedef` каждый идентификатор, являющийся частью любого описателя в этом описании, становится синтаксически эквивалентным ключевому слову, имеющему тот тип, который ассоциирует с идентификатором в описанном в п.9.9.4 смысле. Например, после описаний

```
typedef int miles, *klicksp;
typedef struct {
  double re, im;
} complex;
```

конструкции

```
miles distance;
extern klicksp metricp;
complex z, *zp;
```

становятся законными описаниями; при этом типом `distance` является `int`, типом `metricp` – указатель на `int`, типом `z` специфицированная структура `struct complex` и типом `zp` – указатель на такую структуру.

Спецификатор `typedef` не вводит каких-либо совершенно новых типов, а только определяет синонимы для типов, которые можно было бы специфицировать и другим способом. Так в приведенном выше примере переменная `distance` считается имеющей точно такой же тип, что и любой другой объект, описанный в `int`.

9.10 Операторы

За исключением особо оговариваемых случаев, операторы выполняются последовательно.

9.10.1 Операторное выражение

Большинство операторов являются операторными выражениями, которые имеют форму

выражение;

Обычно операторные выражения являются присваиваниями или обращениями к функциям.

9.10.2 Составной оператор (или блок)

С тем чтобы допустить возможность использования нескольких операторов там, где ожидается присутствие только одного, предусматривается составной оператор (который также и эквивалентно называют «блоком»):

составной оператор:
{*список-описаний список-операторов* }

список-описаний:
описание
описание список-описаний

список-операторов:
оператор
оператор список-операторов

Если какой-либо идентификатор из списка-описаний был описан ранее, то во время выполнения блока внешнее описание подавляется и снова вступает в силу после выхода из блока.

Любая инициализация автоматических и регистрационных переменных проводится при каждом входе в блок через его начало. В настоящее время разрешается (но это плохая практика) передавать управление внутрь блока; в таком случае эти инициализации не выполняются. Инициализации статических переменных проводятся только один раз, когда начинается выполнение программы.

Находящиеся внутри блока внешние описания не резервируют памяти, так что их инициализация не разрешается.

9.10.3 Условные операторы

Имеются две формы условных операторов:

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

В обоих случаях вычисляется выражение и, если оно отлично от нуля, то выполняется первый подоператор. Во втором случае, если выражение равно нулю, выполняется второй подоператор. Как обычно, двусмысленность `else` разрешается связыванием `else` с последним встречающимся `if`, у которого нет `else`.

9.10.4 Оператор while

Оператор `while` имеет форму

```
while (выражение) оператор
```

Подоператор выполняется повторно до тех пор, пока значение выражения остается отличным от нуля. Проверка производится перед каждым выполнением оператора.

9.10.5 Оператор do

Оператор `do` имеет форму

```
do оператор while (выражение)
```

Оператор выполняется повторно до тех пор, пока значение выражения не станет равным нулю. Проверка производится после каждого выполнения оператора.

9.10.6 Оператор for

Оператор `for` имеет форму

`for` (*выражение-1; выражение-2; выражение-3*) *оператор*

Оператор `for` эквивалентен следующему

```
выражение-1
while (выражение-2) {
    оператор
    выражение-3
}
```

Таким образом, первое выражение определяет инициализацию цикла; второе специфицирует проверку, выполняемую перед каждой итерацией, так что выход из цикла происходит тогда, когда значение выражения становится нулем; третье выражение часто задает приращение параметра, которое проводится после каждой итерации.

Любое выражение или даже все они могут быть опущены. Если отсутствует второе выражение, то предложение с `while` считается эквивалентным `while`(1); другие отсутствующие выражения просто опускаются из приведенного выше расширения.

9.10.7 Оператор switch

Оператор `switch` (переключатель), вызывает передачу управления к одному из нескольких операторов, в зависимости от значения выражения. Оператор имеет форму

`switch` (выражение) оператор

В выражении проводятся обычные арифметические преобразования, но результат должен иметь тип `int`. Оператор обычно является составным. Любой оператор внутри этого оператора может быть помечен одним или более вариантным префиксом `case`, имеющим форму:

`case` константное выражение:

где константное выражение должно иметь тип `int`. Никакие две вариантные константы в одном и том же переключателе не могут иметь одинаковое значение. Точное определение константного выражения приводится в п.9.16.

Кроме того, может присутствовать самое большее один операторный префикс вида

`default`:

При выполнении оператора `switch` вычисляется входящее в него выражение и сравнивается с каждой вариантной константой. Если одна из вариантных констант оказывается равной значению этого выражения, то управление передается оператору, который следует за совпадающим вариантным префиксом. Если ни одна из вариантных констант не совпадает со значением выражения и если при этом присутствует префикс `default`, то управление передается оператору, помеченному этим префиксом. если ни один из вариантов не подходит и префикс `default` отсутствует, то ни один из операторов в переключателе не выполняется.

Сами по себе префиксы `case` и `default` не изменяют поток управления, которое беспрепятственно проходит через такие префиксы. Для выхода из переключателя смотрите оператор `break`, п.9.10.8.

Обычно оператор, который входит в переключатель, является составным. Описания могут появляться в начале этого оператора, но инициализации автоматических и регистровых переменных будут неэффективными.

9.10.8 Оператор break

Оператор `break` вызывает завершение выполнения наименьшего охватывающего этот оператор оператора `while`, `do`, `for` или `switch`; управление передается оператору, следующему за завершенным оператором.

9.10.9 Оператор continue

Оператор `continue` приводит к передаче управления на продолжающую цикл часть наименьшего охватывающего этот оператор оператора `while`, `do` или `for`; то есть на конец цикла. Более точно, в каждом из операторов

```
while (...) {
  ...
  contin: ;
}

do {
  ...
  contin: ;
} while(...);

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

Оператор `continue` эквивалентен оператору `goto contin`.[1]

9.10.10 Оператор возврата

Возвращение из функции в вызывающую программу осуществляется с помощью оператора `return`, который имеет одну из следующих форм

```
return;
return выражение;
```

В первом случае возвращаемое значение неопределенно. Во втором случае в вызывающую функцию возвращается значение выражения. Если требуется, выражение преобразуется к типу функции, в которой оно появляется, как в случае присваивания. Попадание на конец функции эквивалентно возврату без возвращаемого значения.

[1] За меткой `contin:` следует пустой оператор; см. п.9.10.13.

9.10.11 Оператор goto

Управление можно передавать безусловно с помощью оператора

```
goto идентификатор;
```

идентификатор должен быть меткой (п.9.10.12), локализованной в данной функции.

9.10.12 Помеченный оператор

Перед любым оператором может стоять помеченный префикс вида

```
идентификатор:
```

который служит для описания идентификатора в качестве метки. Метки используются только для указания места, куда передается управление оператором `goto`. Областью действия метки является данная функция, за исключением любых подблоков, в которых тот же идентификатор описан снова. Смотри п.9.12.

9.10.13 Пустой оператор

Пустой оператор имеет форму:

```
;
```

Пустой оператор оказывается полезным, так как он позволяет поставить метку перед закрывающей скобкой } составного оператора или указать пустое тело в операторах цикла, таких как `while`.

9.11 Внешние определения

C-программа представляет собой последовательность внешних определений. Внешнее определение описывает идентификатор как имеющий класс памяти `extern` (по умолчанию), или возможно `static`, и специфицированный тип. Спецификатор типа (п.9.9.2) также может быть пустым; в этом случае считается, что тип является типом `int`. Область действия внешних определений распространяется до конца файла, в котором они приведены, точно так же, как влияние описаний простирается до конца блока. Синтаксис внешних определений не отличается от синтаксиса описаний, за исключением того, что только на этом уровне можно приводить текст функций.

9.11.1 Внешнее определение функции

Определение функции имеет форму

определение-функции:
спецификаторы-описания описатель-функции *тело-функции*

Спецификаторами класса памяти, допускаемыми в качестве спецификаторов-описания, являются `extern` или `static`; о различии между ними смотри п.9.12.2. Описатель функции подобен описателю для «функции, возвращающей T», за исключением того, что он перечисляет формальные параметры определяемой функции.

Описатель-функции:
описатель (*список-параметров*)

список параметров:
идентификатор
идентификатор, список-параметров

Тело-функции имеет форму

тело-функции:
список-описаний составной-оператор

Идентификаторы из списка параметров и только они могут быть описаны в списке описаний. Любой идентификатор, тип которого не указан, считается имеющим тип `int`. Единственным допустимым здесь спецификатором класса памяти является `register`; если такой класс памяти специфицирован, то в начале выполнения функции соответствующий фактический параметр копируется, если это возможно, в регистр.

Вот 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` – спецификатор-типа, `max(a,b,c)` – описатель-функции, `int a,b,c;` – список-описаний формальных параметров, `{...}` – блок, содержащий текст оператора.

В языке **C** все фактические параметры типа `float` преобразуются к типу `double`, так что описания формальных параметров, объявленных как `float`, приспособлены прочесть параметры типа `double`. Аналогично, поскольку ссылка на массив в любом контексте (в частности в фактическом параметре) рассматривается как указатель на первый элемент массива, описания формальных параметров вила «массив T» приспособлены прочесть: «указатель на T». И наконец, поскольку структуры, объединения и функции не могут быть переданы функции, бессмысленно описывать формальный параметр как структуру, объединение или функцию (указатели на такие объекты, конечно, допускаются).

9.11.2 Внешние определения данных

Внешнее определение данных имеет форму

определение-данных:
    описание

Классом памяти таких данных может быть `extern` (в частности, по умолчанию) или `static`, но не `auto` или `register`.

9.12 Правила, определяющие область действия

Вся **C**-программа необязательно компилируется одновременно; исходный текст программы может храниться в нескольких файлах и ранее скомпилированные процедуры могут загружаться из библиотек. Связь между функциями может осуществляться как через явные обращения, так и в результате манипулирования с внешними данными.

Поэтому следует рассмотреть два вида областей действия: во-первых, ту, которая может быть названа лексической областью действия идентификатора и которая по существу является той областью в программе, где этот идентификатор можно использовать, не вызывая диагностического сообщения «неопределенный идентификатор»; и во-вторых, область действия, которая связана с внешними идентификаторами и которая характеризуется правилом, что ссылки на один и тот же внешний идентификатор являются ссылками на один и тот же объект.

9.12.1 Лексическая область действия

Лексическая область действия идентификаторов, описанных во внешних определениях, простирается от определения до конца исходного файла, в котором он находится. Лексическая область действия идентификаторов, являющихся формальными параметрами, распространяется на ту функцию, к которой они относятся. Лексическая область действия идентификаторов, описанных в начале блока, простирается до конца этого блока. Лексической областью действия меток является та функция, в которой они находятся.

Поскольку все обращения на один и тот же внешний идентификатор обращаются к одному и тому же объекту (см. п.9.12.2), компилятор проверяет все описания одного и того же внешнего идентификатора на совместимость; в действительности их область действия распространяется на весь файл, в котором они находятся.

Во всех случаях, однако, есть некоторый идентификатор, явным образом описан в начале блока, включая и блок, который образует функцию, то действие любого описания этого идентификатора вне блока приостанавливается до конца этого блока.

Напомним также (п.9.9.5), что идентификаторы, соответствующие обычным переменным, с одной стороны, и идентификаторы, соответствующие членам и ярлыкам структур и объединений, с другой стороны, формируют два непересекающихся класса, которые не

вступают в противоречие. Члены и ярлыки подчиняются тем же самым правилам определения областей действия, как и другие идентификаторы. Имена, специфицируемые с помощью `typedef`, входят в тот же класс, что и обычные идентификаторы. Они могут быть переопределены во внутренних блоках, но во внутреннем описании тип должен быть указан явно:

```
typedef float distance;
...
{
  auto int distance;
  ...
}
```

Во втором описании спецификатор типа `int` должен присутствовать, так как в противном случае это описание будет принято за описание без описателей с типом `distance`.[1]

9.12.2 Область действия внешних идентификаторов

Если функция ссылается на идентификатор, описанный как `extern`, то где-то среди файлов или библиотек, образующих полную программу, должно содержаться внешнее определение этого идентификатора. Все функции данной программы, которые ссылаются на один и тот же внешний идентификатор, ссылаются на один и тот же объект, так что следует позаботиться, чтобы специфицированные в этом определении тип и размер были совместимы с типом и размером, указываемыми в каждой функции, которая ссылается на эти данные.

Появление ключевого слова `extern` во внешнем определении указывает на то, что память для описанных в нем идентификаторов будет выделена в другом файле. Следовательно, в состоящей из многих файлов программе внешнее определение идентификатора, не содержащее спецификатора `extern`, должно появляться ровно в одном из этих файлов. Любые другие файлы, которые желают дать внешнее определение этого идентификатора, должны включать в это определение слово `extern`. Идентификатор может быть инициализирован только в том описании, которое приводит к выделению памяти.

Идентификаторы, внешнее определение которых начинается со слова `static`, недоступны из других файлов. Функции могут быть описаны как `static`.

9.13 Строки управления компилятором

Компилятор языка **C** содержит препроцессор, который позволяет осуществлять макроподстановки, условную компиляцию и включение именованных файлов. Строки, начинающиеся с #, общаются с этим препроцессором. Синтаксис этих строк не связан с остальным языком; они могут появляться в любом месте и их влияние распространяется (независимо от области действия) до конца исходного программного файла.

[1] Согласитесь, что лед здесь тонок.

9.13.1 Замена лексем

Управляющая компилятором строка вида

```
#define идентификатор строка-лексем
```

приводит к тому, что препроцессор заменяет последующие вхождения этого идентификатора на указанную строку лексем.[1]

Строка вида

```
#define идентификатор(идентификатор1,...,идентификаторN) строка лексем
```

где между первым «идентификатором» и открывающейся скобкой «(» нет пробела, представляет собой макроопределение с аргументами. Последующее вхождение первого идентификатора, за которым следует открывающая скобка «(», последовательность разделенных запятыми лексем и закрывающая скобка «)», заменяются строкой лексем из определения. Каждое вхождение идентификатора, упомянутого в списке формальных параметров в определении, заменяется соответствующей строкой лексем из обращения. Фактическими аргументами в обращении являются строки лексем, разделенные запятыми; однако запятые, входящие в закавыченные строки или заключенные в круглые скобки, не разделяют аргументов. Количество формальных и фактических параметров должно совпадать. Текст внутри строки или символьной константы не подлежит замене.

В обоих случаях замененная строка просматривается снова с целью обнаружения других определенных идентификаторов. В обоих случаях слишком длинная строка определения может быть продолжена на другой строке, если поместить в конце продолжаемой строки обратную косую черту \ .

Описываемая возможность особенно полезна для определения «объявляемых констант», как, например,

```
#define tabsize 100
int table[tabsize];
```

Управляющая строка вида

```
#undef идентификатор
```

приводит к отмене препроцессорного определения данного идентификатора.

9.13.2 Включение файлов

Строка управления компилятором вида

```
#include "filename"
```

приводит к замене этой строки на все содержимое файла с именем filename. Файл с этим именем сначала ищется в справочнике начального исходного файла, а затем в последовательности стандартных мест. В отличие от этого управляющая строка вида

[1] Обратите внимание на отсутствие в конце точки с запятой.

```
#include <filename>
```

ищет файл только в стандартных местах и не просматривает справочник исходного файла.

Строки `#include` могут быть вложенными.

9.13.3 Условная компиляция

Строка управления компилятором вида

```
#if константное выражение
```

проверяет, отлично ли от нуля значение константного выражения (см. п.9.8).

Управляющая строка вида

```
#ifdef идентификатор
```

проверяет, определен ли этот идентификатор в настоящий момент в препроцессоре, т.е. определен ли этот идентификатор с помощью управляющей строки `#define`.

9.14 Неявные описания

Не всегда является необходимым специфицировать и класс памяти и тип идентификатора в описании. Во внешних определениях и описаниях формальных параметров и членов структур класс памяти определяется по контексту. Если в находящемся внутри функции описании не указан тип, а только класс памяти, то предполагается, что идентификатор имеет тип `int`; если не указан класс памяти, а только тип, то идентификатор предполагается описанным как `auto`. Исключение из последнего правила дается для функций, потому что спецификатор `auto` для функций является бессмысленным[1]; если идентификатор имеет тип «функция, возвращающая T», то он предполагается неявно описанным как `extern`.

Входящий в выражение и неописанный ранее идентификатор, за которым следует скобка «(», считается описанным по контексту как «функция, возвращающая `int`».

9.15 Снова о типах

В этом разделе обобщаются сведения об операциях, которые можно применять только к объектам определенных типов.

[1] Язык **C** не в состоянии компилировать программу в стек.

9.15.1 Структуры и объединения

Только две вещи можно сделать со структурой или объединением: назвать один из их членов (с помощью операции « . » или `->`) или извлечь их адрес (с помощью унарной операции &). Другие операции, такие как присваивание им или из них и передача их в качестве параметров, приводят к сообщению об ошибке. В будущем ожидается, что эти операции, но не обязательно какие-либо другие, будут разрешены.

В п.9.8.1 говорится, что при прямой или косвенной ссылке на структуру (с помощью « . » или `->`) имя справа должно быть членом структуры, названной или указанной выражением слева. Это ограничение не навязывается строго компилятором, чтобы дать возможность обойти правила типов. В действительности перед « . » допускается любое L-значение и затем предполагается, что это L-значение имеет форму структуры, для которой стоящее справа имя является членом. Таким же образом, от выражения, стоящего перед `->`, требуется только быть указателем или целым. В случае указателя предполагается, что он указывает на структуру, для которой стоящее справа имя является членом. В случае целого оно рассматривается как абсолютный адрес соответствующей структуры, заданный в единицах машинной памяти.[1]

9.15.2 Функции

Только две вещи можно сделать с функцией: вызвать ее или извлечь ее адрес. Если имя функции входит в выражение не в позиции имени функции, соответствующей обращению к ней, то генерируется указатель на эту функцию. Следовательно, чтобы передать одну функцию другой, можно написать

```
int f();
...
g(f);
```

Тогда определение функции g могло бы выглядеть так:

```
g(funcp)
int(*funcp)();
{
  ...
  (*funcp)();
  ...
}
```

Обратите внимание, что в вызывающей процедуре функция f должна быть описана явно, потому что за ее появлением в g(f) не следует скобка «(».

[1] Такие структуры не являются переносимыми.

9.15.3 Массивы, указатели и индексация

Каждый раз, когда идентификатор, имеющий тип массива, появляется в выражении, он преобразуется в указатель на первый член этого массива. Из-за этого преобразования массивы не являются L-значениями. По определению операция индексация `[]` интерпретируется таким образом, что `E1[E2]` считается идентичным выражению `*((E1)+(E2))`. Согласно правилам преобразований, применяемым при операции `+`, если `E1` – массив, а `E2` – целое, то `E1[E2]` ссылается на `E2`-й член массива `E1`. Поэтому несмотря на несимметричный вид операция индексации является коммутативной.

В случае многомерных массивов применяется последовательное правило. Если `E` является n-мерным массивом размера `i*j*...*k`, то при появлении в выражении `E` преобразуется в указатель на (n-1)-мерный массив размера `j*...*k`. Если операция `*` либо явно, либо неявно, как результат индексации, применяется к этому указателю, то результатом операции будет указанный (n-1)-мерный массив, который сам немедленно преобразуется в указатель.

Рассмотрим, например, описание

```
int x[3][5];
```

Здесь x массив целых размера `3*5`. При появлении в выражении x преобразуется в указатель на первый из трех массивов из 5 целых. В выражении x[i], которое эквивалентно `*(x+i)`, сначала x преобразуется в указатель так, как описано выше; затем i преобразуется к типу x, что вызывает умножение i на длину объекта, на который указывает указатель, а именно на 5 целых объектов. Результаты складываются, и применение косвенной адресации дает массив (из 5 целых), который в свою очередь преобразуется в указатель на первое из этих целых. Если в выражение входит и другой индекс, то та же самая аргументация применяется снова; результатом на этот раз будет целое.

Из всего этого следует, что массивы в языке **C** хранятся построчно (последний индекс изменяется быстрее всего) и что первый индекс в описании помогает определить общее количество памяти, требуемое для хранения массива, но не играет никакой другой роли в вычислениях, связанных с индексацией.

9.15.4 Явные преобразования указателей

Разрешаются определенные преобразования, с использованием указателей, но они имеют некоторые зависящие от конкретной реализации аспекты. Все эти преобразования задаются с помощью операции явного преобразования типа; см. п.9.8.2 и 9.9.7.

Указатель может быть преобразован в любой из целочисленных типов, достаточно большой для его хранения. Требуется ли при этом `int` или `long`, зависит от конкретной машины. Преобразующая функция также является машинно-зависимой, но она будет вполне естественной для тех, кто знает структуру адресации в машине. Детали для некоторых конкретных машин приводятся ниже.

Объект целочисленного типа может быть явным образом преобразован в указатель. Такое преобразование всегда переводит преобразованное из указателя целое в тот же самый указатель, но в других случаях оно будет машинно-зависимым.

Указатель на один тип может быть преобразован в указатель на другой тип. Если преобразуемый указатель не указывает на объекты, которые подходящим образом выравнены в памяти, то результирующий указатель может при использовании вызывать ошибки адресации. Гарантируется, что указатель на объект заданного размера может быть преобразован в указатель на объект меньшего размера и снова обратно, не претерпев при этом изменения.

Например, процедура распределения памяти могла бы принимать запрос на размер выделяемого объекта в байтах, а возвращать указатель на символы; это можно было бы использовать следующим образом.

```
extern char *alloc();
double *dp;
dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

Функция `alloc` должна обеспечивать (машинно-зависимым способом), что возвращаемое ею значение будет подходящим для преобразования в указатель на `double`; в таком случае использование этой функции будет переносимым.

Представление указателя на PDP-11 соответствует 16-битовому целому и измеряется в байтах. Объекты типа `char` не имеют никаких ограничений на выравнивание; все остальные объекты должны иметь четные адреса.

На HONEYWELL 6000 указатель соответствует 36-битовому целому; слову соответствует 18 левых битов и два непосредственно примыкающих к ним справа бита, которые выделяют символ в слове. Таким образом, указатели на символы измеряются в единицах 2^{16} байтов; все остальное измеряется в единицах 2^{18} машинных слов. Величины типа `double` и содержащие их агрегаты должны выравниваться по четным адресам слов (0 по модулю 2^{19}). ЭВМ IBM 370 и INTERDATA 8/32 сходны между собой. На обеих машинах адреса измеряются в байтах; элементарные объекты должны быть выровнены по границе, равной их длине, так что указатели на `short` должны быть кратны двум, на `int` и `float` – четырем и на `double` – восьми. Агрегаты выравниваются по самой строгой границе, требуемой каким-либо из их элементов.

9.16 Константные выражения

В нескольких местах в языке **C** требуются выражения, которые после вычисления становятся константами: после вариантного префикса `case`, в качестве границ массивов и в инициализаторах. В первых двух случаях выражение может содержать только целые константы, символьные константы и выражения `sizeof`, возможно связанные либо бинарными операциями

```
+  -  *  /  .  %  &  |  ^  <<  >>  ==  !=  <  >  <=  >=
```

либо унарными операциями

```
-  ~
```

либо тернарной операцией « ? : ».

Круглые скобки могут использоваться для группировки, но не для обращения к функциям.

В случае инициализаторов допускается бóльшая свобода; кроме перечисленных выше константных выражений можно также применять унарную операцию & к внешним или статическим объектам и к внешним или статическим массивам, имеющим в качестве индексов константное выражение. Унарная операция & может быть также применена неявно, в результате появления неиндексированных массивов и функций. Основное правило заключается в том, что после вычисления инициализатор должен становится либо константой, либо адресом ранее описанного внешнего или статического объекта плюс или минус константа.

9.17 Соображения о переносимости

Некоторые части языка **C** по своей сути машинно-зависимы. Следующие ниже перечисление потенциальных трудностей хотя и не являются всеобъемлющими, но выделяет основные из них.

Как показала практика, вопросы, целиком связанные с аппаратным оборудованием, такие как размер слова, свойства плавающей арифметики и целого деления, не представляют особенных затруднений. Другие аспекты аппаратных средств находят свое отражение в различных реализациях. Некоторые из них, в частности, знаковое расширение (преобразующее отрицательный символ в отрицательное целое) и порядок, в котором помещаются байты в слове, представляют собой неприятность, которая должна тщательно отслеживаться. Большинство из остальных проблем этого типа не вызывает сколько-нибудь значительных затруднений.

Число переменных типа `register`, которое фактически может быть помещено в регистры, меняется от машины к машине, также как и набор допустимых для них типов. Тем не менее все компиляторы на своих машинах работают надлежащим образом; лишние или недопустимые регистровые описания игнорируются.

Некоторые трудности возникают только при использовании сомнительной практики программирования. Писать программы, которые зависят от каких-либо этих свойств, является чрезвычайно неразумным.

Языком не указывается порядок вычисления аргументов функций; они вычисляются справа налево на PDP-11 и VAX-11 и слева направо на остальных машинах. Порядок, в котором происходят побочные эффекты, также не специфицируется.

Так как символьные константы в действительности являются объектами типа `int`, допускается использование символьных констант, состоящих из нескольких символов. Однако, поскольку порядок, в котором символы приписываются к слову, меняется от машины к машине, конкретная реализация оказывается весьма машинно-зависимой.

Присваивание полей к словам и символов к целым осуществляется право налево на PDP-11 и VAX-11 и слева направо на других машинах. Эти различия незаметны для изолированных программ, в которых не разрешено смешивать типы (преобразуя, например, указатель на `int` в указатель на `char` и затем проверяя указываемую память), но должны

учитываться при согласовании с накладываемыми извне схемами памяти.

Язык, принятый на различных компиляторах, отличается только незначительными деталями. Самое заметное отличие состоит в том, что используемый в настоящее время компилятор на PDP-11 не инициализирует структуры, которые содержат поля битов, и не допускает некоторые операции присваивания в определенных контекстах, связанных с использованием значения присваивания.

9.18 Анахронизмы

Так как язык **C** является развивающимся языком, в старых программах можно встретить некоторые устаревшие конструкции. Хотя большинство версий компилятора поддерживает такие анахронизмы, они в конце концов исчезнут, оставив за собой только проблемы переносимости.

В ранних версиях **C** для проблем присваивания использовалась форма =оп, а не оп=, приводя к двусмысленностям, типичным примером которых является

```
x=-1
```

где x фактически уменьшается, поскольку операции = и − примыкают друг к другу, но что вполне могло рассматриваться и как присваивание −1 к x.

Синтаксис инициализаторов изменился: раньше знак равенства, с которого начинается инициализатор, отсутствовал, так что вместо

```
int x = 1;
```

использовалось

```
int x 1;
```

изменение было внесено из-за инициализации

```
int f (1+2)
```

которая достаточно сильно напоминает определение функции, чтобы смутить компиляторы.

9.19 Сводка синтаксических правил

Эта сводка синтаксиса языка **C** предназначена скорее для облегчения понимания и не является точной формулировкой языка.

9.19.1 Выражения

Основными выражениями являются следующие:

выражение:
- первичное-выражение
- * выражение
- & выражение
- – выражение
- ! выражение
- ~ выражение
- ++ L-значение
- –– L-значение
- L-значение ++
- L-значение ––
- `sizeof`(выражение)
- (имя типа) выражение
- выражение бинарная-операция выражение
- выражение ? выражение : выражение
- L-значение операция-присваивания выражение
- выражение, выражение

первичное выражение:
- идентификатор
- константа
- строка
- (выражение)
- первичное-выражение (*список выражений*)
- первичное-выражение [выражение]
- L-значение . идентификатор
- первичное выражение –> идентификатор

L-значение:
- идентификатор
- первичное-выражение [выражение]
- L-значение . идентификатор
- первичное-выражение –> идентификатор
- * выражение
- (L-значение)

Операции первичных выражений

```
()  []  .  ->
```

имеют самый высокий приоритет и группируются слева направо. Унарные операции

```
*  &  -  !  ~  ++  --  sizeof(имя типа)
```

имеют более низкий приоритет, чем операции первичных выражений, но более высокий, чем приоритет любой бинарной операции. Эти операции группируются справа налево. Все бинарные операции и условная операция группируются слева направо[1] и их приоритет убывает в следующем порядке:

Бинарные операции:

```
*    /    %
+    -
≫    ≪
<    >    <=   >=
==   !=
&
~
|
&&
||
? :
```

Все операции присваивания имеют одинаковый приоритет и группируются справа налево. Операции присваивания:

```
=   +=   -=   *=   ?=   %=   >>=   <<=   &=   ~=   |=
```

Операция запятая имеет самый низкий приоритет и группируется слева направо.

9.19.2 Описания

Описание:

спецификаторы-описания список-инициализируемых-описателей

спецификаторы-описания:

спецификатор-типа спецификаторы-описания
спецификатор-класса-памяти спецификаторы-описания

спецификатор-класса-памяти:

`auto`
`static`
`extern`
`register`
`typedef`

спецификатор-типа:

`char`
`short`
`int`
`long`

[1] Условная операция группируется справа налево; это изменение внесено в язык в 1978г.(прим.перевод.)

`unsigned`
`float`
`double`
спецификатор-структуры-или-объединения
определяющее-тип-имя

список-инициализируемых-описателей:
инициализируемый-описатель
инициализируемый-описатель, список-инициализируемых-описателей
инициализируемый-описатель, *описатель-инициализатор*

описатель:
идентификатор
(описатель)
`*` описатель
описатель ()
описатель [*константное выражение*]

спецификатор-структуры-или-объединения:
`struct` список-описателей-структуры
`struct` идентификатор {список-описаний-структуры}
`struct` идентификатор
`union` {список-описаний-структуры}
`union` идентификатор {список-описаний-структуры}
`union` идентификатор

список-описаний-структуры:
описание-структуры
описание-структуры список-описаний-структуры

описание структуры:
спецификатор-типа список-описателей-структуры

список-описателей-структуры:
описатель-структуры
описатель-структуры,список-описателей-структуры

описатель-структуры:
описатель
описатель: константное выражение
: константное-выражение

инициализатор:
= выражение
= {список-инициализатора}

список инициализатора:
выражение
список-инициализатора, список-инициализатора
{список-инициализатора}

имя-типа:
спецификатор-типа абстрактный-описатель

абстрактный-описатель:
пусто
{абстрактный-описатель}
`*` абстрактный-описатель
абстрактный-описатель ()
абстрактный-описатель [*константное-выражение*]

определяющее-тип-имя:
идентификатор

9.19.3 Операторы

составной-оператор:
{ *список-описаний список-операторов* }

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; выражение-2; выражение-3*) оператор
`switch` (выражение) оператор
`case` константное-выражение: оператор
`default`: оператор
`break`;
`continue`;
`return`;
`return` выражение;
`goto` идентификатор;
идентификатор: оператор
;

9.19.4 Внешние определения

Программа:
    внешнее-определение
    внешнее-определение программа

внешнее-определение:
    определение-функции
    определение-данных

определение-функции:
    спецификатор-типа описатель-функции тело-функции

описатель-функции:
    описатель (список-параметров)

список-параметров:
    идентификатор
    идентификатор, список-параметров

тело-функции:
    список-описаний-типа оператор-функции

оператор-функции:
    {*список описаний* список-операторов}

определение данных:
    `extern` *спецификатор типа список инициализируемых описателей*;
    `static` *спецификатор типа список инициализируемых описателей*;

9.19.5 Препроцессор

`#define` идентификатор строка-лексем
`#define`
`#define` идентификатор(идентификатор1,. . . ,идентификаторN) строка лексем
`#undef` идентификатор
`#include` ”имя-файла”
`#include` <имя-файла>
`#if` константное-выражение
`#ifdef` идентификатор
`#ifndef` идентификатор
`#else`
`#endif`
`#line` константа идентификатор

10 Последние изменения языка C (15 ноября 1978 г.)

10.1 Присваивание структуры

Структуры могут быть присвоены, переданы функциям в качестве аргументов и возвращены функциям. Типы участвующих операндов должны оставаться теми же самыми. Другие правдоподобные операторы, такие как сравнение на равенство, не были реализованы.

В реализации возвращения структур функциями на PDP-11 имеется коварный дефект: если во время возврата происходит прерывание и та же самая функция реентерабельно вызывается во время этого прерывания, то значение возвращаемое из первого вызова, может быть испорчено. Эта трудность может возникнуть только при наличии истинного прерывания, как из операционной системы, так и из программы пользователя, прерывания, которое существенно для использования сигналов; обычные рекурсивные вызовы совершенно безопасны.

10.2 Тип перечисления

Введен новый тип данных, аналогичный скалярным типам языка паскаль. К спецификатору-типа в его синтаксическом описании в Приложении А следует добавить спецификатор-перечисления с синтаксисом:

спецификатор-перечисления:
    `enum` список-перечисления
    `enum` идентификатор список-перечисления
    `enum` идентификатор

список-перечисления:
    перечисляемое
    список-перечисления, перечисляемое

перечисляемое:
    идентификатор
    идентификатор = константное выражение

Роль идентификатора в спецификаторе-перечисления полностью аналогична роли ярлыка структуры в спецификаторе-структуры; идентификатор обозначает определенное перечисление. Например, описание

```
enum color { red, white, black, blue };
 . . .
enum color *cp, col;
```

объявляет идентификатор `color` ярлыком перечисления типа, описывающего различные цвета и затем объявляет cp указателем на объект этого типа, а col – объектом этого типа.

Идентификаторы в списке-перечисления описываются как константы и могут появиться там, где требуются (по контексту) константы. Если не используется вторая форма перечисляемого (с равенством =), то величины констант начинаются с 0 и возрастают на 1 в соответствии с прочтением их описания слева на право. Перечисляемое с присвоением = придает соответствующему идентификатору указанную величину; последующие идентификаторы продолжают прогрессию от приписанной величины.

Все ярлыки перечисления и константы могут быть различными и непохожими на ярлыки и члены структур даже при условии использования одного и того же множества идентификаторов.

Объекты данного типа перечисления рассматриваются как объекты, имеющие тип, отличный от любых типов и контролирующая программа `lint` сообщает об ошибках несоответствия типов. В реализации на PDP-11 со всеми перечисляемыми переменными оперируют так, как если бы они имели тип `int`.