БАКАЛАВРИАТ

Д.В. Парфенов

ЯЗЫК СИ

кратко и ясно

УЧЕБНОЕ ПОСОБИЕ

ВЫСШЕЕ ОБРАЗОВАНИЕ — БАКАЛАВРИАТ
серия основана в 1996 г.

Д.В. ПАРФЕНОВ

ЯЗЫК СИ

КРАТКО И ЯСНО

УЧЕБНОЕ ПОСОБИЕ

Допущено УМО по классическому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлениям подготовки 01.03.02 «Прикладная математика и информатика» и 02.03.02 «Фундаментальная информатика и информационные технологии»
Регистрационный номер рецензии 088-4/42-14 от 24.03.2014

Электронно-Библиотечная Система
znanium.com

Москва
ИНФРА-М
2023

УДК 519.682.2(075.8)
ББК 32.973-018.1я73
П18

ФЗ № 436-ФЗ	Издание не подлежит маркировке в соответствии с п. 1 ч. 4 ст. 11

Парфенов Д.В.

П18 Язык Си: кратко и ясно : учебное пособие / Д.В. Парфенов. — Москва : ИНФРА-М, 2023. — 320 с. — (Высшее образование: Бакалавриат).

ISBN 978-5-16-017910-0 (print)
ISBN 978-5-16-110920-5 (online)

В учебном пособии представлена новейшая версия языка программирования Си, ставшего международным стандартом. Совмещены подходы учебного пособия и справочника, что облегчает студентам изучение материала. Особое внимание уделено стилю программирования, его эффективности и выразительности.

Для студентов технических вузов, обучающихся по направлениям подготовки 01.03.02 «Прикладная математика и информатика» и 02.03.02 «Фундаментальная информатика и информационные технологии».

УДК 519.682.2(075.8)
ББК 32.973-018.1я73

ISBN 978-5-16-017910-0 (print)
ISBN 978-5-16-110920-5 (online)

ООО «Научно-издательский центр ИНФРА-М»
127214, Москва, ул. Полярная, д. 31В, стр. 1
Тел.: (495) 280-15-96, 280-33-86. Факс: (495) 280-36-29
E-mail: books@infra-m.ru http://www.infra-m.ru

Подписано в печать 29.06.2022.
Формат 60×90/16. Бумага офсетная. Гарнитура Petersburg.
Печать цифровая. Усл. печ. л. 20,0.
ППТ10. Заказ № 00000

ТК 279800-1896456-240414

Отпечатано в типографии ООО «Научно-издательский центр ИНФРА-М»
127214, Москва, ул. Полярная, д. 31В, стр. 1
Тел.: (495) 280-15-96, 280-33-86. Факс: (495) 280-36-29

Предисловие

Эта книга задумана как дорога новичка к профессиональному пониманию языка программирования Си. Профессионал должен уметь отвечать не только на вопрос «Как это сделать?», но и на вопрос «Почему?». Для этого необходимо глубокое понимание изящных и логичных механизмов языка, избавляющее от потребности заучивать множество следствий: они автоматически вытекают из немногих основных принципов. Надеюсь, что настоящее учебное пособие сыграет роль «учебника алгоритмического языка», в котором значительное внимание уделяется также стилю программирования, эффективности и выразительности.

Поскольку область применения Си как одного из лидеров среди языков программирования очень широка и охватывает сферы от управления производственными процессами до средств поддержки виртуальной реальности, от создания операционных систем до сложных математических вычислений, автор пытался избежать привязки повествования и многочисленных примеров к какой-либо конкретной предметной области, чтобы сделать изложение одновременно компактным, универсальным, достаточно полным, но не перегруженным мелочами, систематическим и в то же время четко построенным по принципу «от простого к сложному».

Известно немало хороших пособий по языку Си — ведь его популярность огромна. Человек, взявший в руки эту книгу, обязательно задаст вопрос: «Зачем мне именно эта книга?» Но даже беглого просмотра достаточно, чтобы убедиться, что предлагаемое пособие существенно отличается от других, представленных в печати, как способом подачи материала, так и кру́гом рассматриваемых вопросов. Дело не только в том, что в пособии представлена новейшая версия языка, ставшая международным стандартом в конце 2011 г.

Многолетний опыт чтения данного курса в Московском государственном техническом университете радиотехники,

электроники и автоматики (МГТУ МИРЭА) убедил автора в том, что обращение к учебному пособию как к справочнику является неотъемлемой чертой современного процесса освоения программирования. Создавая свои первые программы, студенты стремятся найти ответы на частные вопросы в первую очередь в том же источнике, который ввел их в суть дела. Поэтому представляется, что такая книга должна совмещать подходы учебника и справочника, а они заметно противоречат друг другу. Отсюда вытекает нетрадиционно выстроенное изложение, четко структурированное и активно привлекающее компактную схематическую запись языковых правил. Насколько автору удалось реализовать задуманное, судить читателям.

Введение. Базовые понятия программирования

В1. Основные сведения об аппаратной части компьютеров

Современные компьютеры — электронные устройства, предназначенные для ввода, хранения, обработки и вывода представимой в электронном виде информации. Их функции в зависимости от назначения могут сильно различаться: научные, инженерные, экономические и другие расчеты, математическое моделирование поведения различных систем, интеллектуальное управление механизмами и процессами (например, течением химических реакций), обработка, поиск, передача, анализ и синтез информации (текстовой, визуальной, звуковой и т.д.), средства создания виртуальной и расширенной реальности (синтез визуальных, слуховых и тактильных воздействий, имитирующих явления реального мира или фантастических миров). Столь различающиеся задачи определяют многообразие вычислительных средств.

По физическому способу организации вычислений различают цифровые, аналоговые и квантовые вычислители. Не вдаваясь в описание их различий, сосредоточимся на цифровых компьютерах как наиболее распространенных в настоящее время и практически значимых. В зависимости от круга решаемых задач различают компьютеры:

- ◊ узкоспециализированные — предназначены для единственного применения, например управления режимом работы двигателя автомобиля;
- ◊ специализированные — нацелены на решение множества задач одного типа, например игровая приставка к телевизору;
- ◊ широкого применения (например, персональный компьютер, высокопроизводительный вычислительный кластер — связка мощных компьютеров).

Назначение определяет конкретные используемые аппаратные и программные средства компьютера.

Аппаратные средства – это совокупность всех электронных схем, механизмов и элементов конструкций, составляющих «физическую» часть вычислителя (англ. hardware[1], в повседневной речи часто именуемые «железом» компьютера), а программные средства – исчерпывающее описание правил обработки информации силами аппаратных средств (англ. software). В качестве предельно грубой, но наглядной аналогии можно уподобить аппаратные средства телу человека, а программные – полному набору правил его поведения.

Успешное программирование предполагает базовое понимание ключевых аппаратных средств типичного персонального компьютера.

Оперативное запоминающее устройство (ОЗУ, оперативная память произвольного доступа, англ. random access memory – RAM) – это микросхемы разделенной на ячейки одинакового объема памяти. В такую память можно целиком или частично относительно быстро записать данные или программы в произвольной последовательности, а потом прочитать ту же информацию в том же или другом порядке. Доступ может быть осуществлен непосредственно к одной произвольной ячейке памяти. Для попадания в требуемую ячейку используется *адресация* – сопоставление по некоторому правилу ячейкам памяти номеров – *адресов*. Адрес однозначно определяет конкретную ячейку памяти. При выключении компьютера содержимое такой памяти обычно полностью утрачивается. В дальнейшем говоря о памяти, будем почти всегда подразумевать именно такую ее разновидность.

На самом деле имеется еще один важный вид памяти – кэш-память (англ. cache memory, от фр. caché – скрытый, спрятанный). Назначение этой памяти, многократно (от десятков до тысяч раз) более быстрой по сравнению с оперативной, – промежуточное хранение наиболее востребованных данных и команд для быстрой доставки их в процессор и из процессора. «Скрытый» характер памяти проявляется в невоз-

[1] Чтобы упростить понимание довольно специфических сообщений компилятора и сборщика и ориентации в многочисленных программистских текстах, в скобках приводятся английские варианты ключевых терминов.

можности адресовать то, что в ней лежит: специальная подсистема компьютера сама решает, что и на какое время там разместить, и осуществляет автоматический доступ в нее так, как будто информация размещена в обычной оперативной памяти. Профессиональному программисту необходимо знать о наличии (а еще лучше алгоритмы работы) кэш-памяти, чтобы реализовать эффективные стратегии работы с памятью.

Центральный процессор (центральный процессорный модуль, англ. central processor unit – CPU, в дальнейшем для краткости – процессор) или несколько взаимосвязанных процессоров – это микросхемы с высокой степенью интеграции (сложностью), непосредственно осуществляющие обработку информации. Они пошагово считывают из оперативной памяти программы – последовательности команд обработки информации, описывающие, что делать, и исполняют их – меняют данные по этим правилам. Данные могут размещаться в памяти или в так называемых регистрах процессора. *Регистр* – одна или несколько связанных между собой ячеек памяти, расположенных внутри процессора. В отличие от обычной оперативной памяти у них обычно нет адресов, но скорость обращения к информации бывает еще выше, чем у быстрой кэш-памяти. Поскольку различные разновидности даже совместимых процессоров оснащены различающимися наборами регистров и некоторые регистры могут отсутствовать или быть по-разному устроены, у программистов далеко не всегда есть возможность пользоваться регистрами непосредственно. В действительности это ограничение несущественно и не мешает создавать переносимые программы, такие, которые могут выполняться на разных компьютерах, не принимая во внимание их не очень значительные отличия.

Постоянные запоминающие устройства включают накопители на жестких магнитных дисках (англ. hard disk drives – HDD), накопители на гибких магнитных дисках (дискетах, англ. floppy disk drives – FDD), накопители на магнитных лентах (стримеры, англ. streamers), разнообразные оптические накопители (MO, MD, CD-ROM, DVD-ROM, DVD-RAM, BD и др.), flash-память, твердотельные накопители (англ. solid state drives, SSD), гибридные, сетевые устройства хранения информации и множество менее распространенных устройств. Запи-

санная на них информация сохраняется и после выключения питания компьютера. Она может быть прочитана впоследствии, а на некоторых типах носителей в дальнейшем полностью или частично стерта и перезаписана какой-то другой информацией. Большинство этих устройств характеризуется последовательным доступом к информации, который заключается в следующем: *1*) устанавливается исходная точка начала нужного фрагмента информации; *2*) фрагмент информации записывается от начала до конца последовательно – блок за блоком; *3*) точно в такой же последовательности он и считывается. К последовательным устройствам относятся также клавиатура (символы набираются друг за другом), текстовое окно консоли (области экрана, куда программа осуществляет вывод текста) и большинство внешних портов для подключения компьютера к другим устройствам – USB (универсальная последовательная шина, англ. universal serial bus), SATA (Serial AT Attachment), COM-порт (последовательный коммуникационный порт), Ethernet, IEEE 1394 (FireWire), Bluetooth и т.д. Все устройства такого рода собирательно назовем *последовательными*. Стандартная библиотека языка Си включает мощные и универсальные средства работы с такими устройствами.

Несмотря на важность для функционирования компьютера и других аппаратных средств, при изучении программирования на языках высокого уровня, таких, как Си, в общем случае нет смысла акцентировать на них внимание. Имеющиеся средства языка достаточны для работы с ними вследствие единообразия его механизмов.

В2. Классификация программного обеспечения

Программные средства являются совокупностью взаимодействующих друг с другом отдельных программ, подобно тому, как аппаратные средства состоят из связанных определенным образом друг с другом выполняющих различные функции компонентов – процессоров, источников вторичного электропитания, жестких дисков и др. Программы могут обмениваться как информацией (данными), так и указа-

ниями к конкретным действиям (командами). По назначению среди программных средств выделяют:

- ◊ *прикладное программное обеспечение*, непосредственно решающее возлагаемые на вычислительную систему задачи, но нуждающееся для этого в определенных ресурсах (организация доступа к памяти и жесткому диску, получение процессорного времени, передача информации другим программам и т.д.);
- ◊ *системное программное обеспечение*, необходимое для предоставления прикладным программам требующихся им ресурсов, координирующее их выполнение и осуществляющее ввод и вывод результатов их работы во внешнюю среду, а также отвечающее за безопасность и сохранность данных и программ. К системным программам относят также средства операционных систем, образующие пользовательский инструментарий по работе с данными и прикладными программами.

В подавляющем большинстве современных компьютеров функции прикладных и системных программ четко разделены, что позволяет упростить их создание и повысить эффективность работы. Действительно, если нескольким различным программам требуется выводить информацию на дисплей, можно, конечно, в каждую из них заложить все необходимые возможности обращения к обеспечивающей это аппаратной части. Однако при этом, во-первых, почти одинаковый фрагмент программного кода придется помещать в каждую программу, вследствие чего ее объем и сложность возрастают; во-вторых, решать, как эти программы будут избегать конфликтов при одновременном обращении к одним и тем же аппаратным средствам. Высока вероятность, что при добавлении новой программы или изменении какой-то из имеющихся, а также при замене части аппаратных средств придется переделывать все программное обеспечение. Для того чтобы избежать этой проблемы, прикладные программы пишут так, чтобы они обращались друг к другу и аппаратным средствам компьютера не напрямую, а через системное программное обеспечение посредством единообразного абстрактного способа передачи данных, называемого *программным интерфейсом приложения* (англ. application program interface – API). Удаление, добавление или замена одной из программ или компонент аппаратной части при такой организации вычислений минимально влияют на остальные прикладные про-

граммы, и их переделка не потребуется. Системное программное обеспечение по выполняемым функциям тоже неоднородно. Условно можно выделить следующие компоненты:

- *драйверы* (англ. drivers) – обычно небольшие узкоспециализированные программы, единственным назначением которых является работа с определенным узким классом аппаратных средств – видеоадаптером данной модели, DVD-ROM определенного стандарта и т.д. Они дают возможность прочим программам использовать возможности аппаратуры просто и единообразно, даже «не зная» подробностей ее функционирования;
- *ядро операционной системы* (англ. operating system kernel) занимается распределением предоставляемых драйверами ресурсов между прикладными программами, поскольку ресурсы компьютера ограниченны и запросы программ могут не быть удовлетворены в максимальной степени в любой момент времени. Ядро также контролирует исполнение самих программ, позволяет их запускать, приостанавливать, прекращать. Вместе с программными надстройками ядро дает возможность программам контролируемо обмениваться информацией. По сути дела прикладные программы обращаются преимущественно к операционной системе с запросами передачи информации друг другу и во внешнюю по отношению к вычислительной системе среду;
- *надстройка ядра*, расширяющая его возможности, оптимизирующая работу приложений и до некоторой степени имитирующая отсутствующие *виртуальные* (англ. virtual – «кажущиеся») аппаратные ресурсы посредством возложения их функций на имеющиеся реальные устройства. В качестве примера можно назвать виртуальную память, виртуальный DVD-ROM и т.д.;
- графическая надстройка и *графический интерфейс пользователя* (англ. graphical user interface – GUI), способствующие удобному взаимодействию человека с компьютером.

В3. Программирование и его аспекты

Каким бы ни было системное и прикладное программное обеспечение, логику его работы необходимо описать формальным и не допускающим двусмысленного тол-

кования способом, а затем перевести на «понятный» процессорам компьютера язык – в *машинный код* (англ. absolute code). Это и составляет предмет программирования. Написание программ – сложный многоступенчатый процесс, обычно допускающий множество вариантов реализации. Понятно, что удобнее создавать программы, не формируя их непосредственно в понятном процессору машинном коде, а излагая логику работы средствами языка, напоминающего сильно упрощенный и формализованный человеческий язык. Однако человеческий язык как таковой не может быть использован для написания программ, хотя бы в силу его неоднозначности и трудноуловимого компьютером контекста. Практически применимые для программирования языки называют *алгоритмическими языками* или *языками программирования* (англ. algorithmic language, programming language). Написанная на таком языке программа, как правило, не может быть выполнена процессором компьютера непосредственно, поэтому необходим переход к командам процессора. Наибольшее распространение получили системы с интерпретацией и с компиляцией текста программы.

При *интерпретации* (англ. interpretation, устный перевод) текст программы пошагово обрабатывается специальной программой – интерпретатором. Она анализирует программу, обнаруживает в ее тексте инструкции, что и как следует делать, переводит каждую инструкцию независимо от других в последовательность команд процессора и выполняет их. Интерпретация проста, но не учитывает глубинные свойства алгоритма исполняемой программы и практически лишена возможности ускорить ее выполнение путем не приводящей к изменению результата перестановки инструкций программы или небольших модификаций структуры данных. Такие приемы называют *оптимизацией кода* (англ. code optimization) программы, часто полезны в смысле ускорения работы программы и уменьшения объема используемых аппаратных ресурсов. Иногда выигрыш от оптимизации бывает очень значительным – до нескольких десятков и даже сотен раз. До некоторой степени оптимизация программы (ее можно уподобить эквивалентным преобразованиям математического выражения с целью упрощения его вычисления) может быть выполнена программистом, что, однако, часто требует от него значительных умствен-

ных усилий и большого времени. Поэтому естественно возложить хотя бы часть оптимизации на сам компьютер при получении исполнимого кода.

Компиляция (кодогенерация, англ. compilation) позволяет это сделать и состоит из следующих этапов:

- *препроцессирование* (англ. preprocessing) – предварительная обработка текста программы для упрощения написания программы и дальнейшей обработки. При этом текст программы преобразуется к виду, более удобному для последующей работы компилятора, например из него удаляются все комментарии, а символические константы заменяются их числовыми значениями;
- собственно *компиляция* (англ. compiling) – анализ текста части программы, общая проверка его соответствия правилам языка, поиск и реализация возможностей оптимизации и порождение исполнимого кода. Если программа состоит из нескольких отдельных модулей, реализующих законченные функции, компиляция осуществляется для каждого из них независимо. Результатом являются объектные модули (англ. object module), состоящие в основном из исполнимого процессором кода и описаний внешних логических связей (англ. dependencies);
- *сборка* модулей (линковка, связывание, компоновка, редактирование связей, англ. linkage) – объединение объектных файлов и, возможно, файлов с дополнительной информацией, необходимой для исполнения программы (ресурсами), в один или несколько исполнимых файлов, являющихся готовой программой. При этом проверяется, насколько стыкуемые объектные модули «подходят» друг к другу, т.е. смогут ли они корректно обмениваться данными между собой и с операционной системой.

В4. Место Си среди алгоритмических языков программирования

Существующие языки программирования условно подразделяют на несколько уровней в соответствии со степенью их абстракции:

- низкого уровня, к которому относят главным образом разнообразные ассемблеры (англ. assembly language) и макроассемблеры (англ. macro assembler). Ассемблеры являются исторически первыми «настоящими» языками программирования. Как правило, они жестко привязаны к системе команд процессора, почти буквально повторяя ее в более человекочитаемом виде. Программирование на ассемблере чрезвычайно трудоемко и применяется нечасто, хотя позволяет создавать предельно эффективные программы;
- среднего-высокого уровня, наиболее яркими и долгоживущими представителями которых являются такие языки, как Си (C), Fortran, Pascal, Basic. Эти языки уже практически не зависят от аппаратных средств компьютера и позволяют программисту сочетать высокую эффективность программ с относительным комфортом их написания. Значительная доля существующего системного и прикладного программного обеспечения создана с их применением;
- высокого уровня – как правило, это объектно-ориентированные языки программирования, предоставляющие еще более высокий уровень абстракции – создание целостных структур данных, называемых объектами, и определение правил работы с этими объектами. Среди наиболее перспективных и значимых для промышленности и науки объектно-ориентированных языков программирования – C++, Objective-C, Java, Ada;
- сверхвысокого уровня – метаязыки, нацеленные на решение специализированных задач очень высокой сложности, такие, как Mathworks Matlab. Эффективность программ, написанных на этих языках, обычно заметно ниже, чем в случае языков более низкого уровня, что сторицей окупается малым временем разработки.

Первая версия языка Си появилась в 1971–1972 гг. и называется K&R по фамилиям создателей – Брайана Кернигана (Brian Wilson Kernighan) и Денниса Ритчи (Dennis MacAlistair Ritchie). В течение прошедших десятилетий язык эволюционировал, стал более гибким и выразительным. Актуальная на момент написания данного пособия версия называется C11, поскольку она была утверждена в качестве международного стандарта ISO/IEC 9899:2011 в 2011 г. С самого начала Си был разработан как средство создания высокоэффективного ма-

шинного кода и сразу нацелен на оптимизирующую компиляцию. Несмотря на солидный возраст, позиции Си в мире программирования продолжают оставаться очень прочными, а его идеи оказались столь удачными, что на его основе возникло несколько других популярных языков, среди которых в первую очередь выделяют C++, Java, PHP, C#, Objective-C.

Таким образом, знание Си чрезвычайно полезно и при освоении языков-потомков, а также изучении общих принципов создания профессионального программного обеспечения. Далее рассмотрим наиболее важные его элементы.

В5. Условные обозначения и особенности описания языка

Чтобы дать наиболее широкое представление о выразительности и мощи языка, конструкции Си в данной книге по возможности приводятся сразу в самой общей алгоритмизированной форме. Для краткости изложения используется несложная система обозначений:

- `равноширинным_шрифтом` в тексте выделены конструкции языка, например ключевые слова, имена переменных и фрагменты программ;
- знак ≡ обозначает тождественность выражений или понятий слева и справа от него;
- пара фигурных скобок, набранная жирным равноширинным шрифтом **`{ }`**, содержит последовательность обязательных элементов языковой конструкции;
- пара квадратных скобок, набранная жирным равноширинным шрифтом **`[ ]`**, содержит необязательные элементы языковой конструкции;
- пара угловых скобок, набранная жирным равноширинным шрифтом **`< >`**, содержит выбор одного из нескольких обязательных элементов языковой конструкции, перечисленных через запятую;
- латинская буква, набранная приподнятым жирным равноширинным шрифтом после квадратных или фигурных скобок, например **`[]`**$^{\mathtt{m}}$, указывает на возможность повторения заключенной в скобках языковой конструкции `m` раз.

В6. Ныряем в язык

Дальнейшее освоение языка требует анализа приведенных примеров и постепенного «втягивания» в самостоятельное программирование. Для его облегчения ниже даются базовые сведения о конструкции простейших программ, которые пока надо принять на веру. Ближе к середине книги их смысл полностью прояснится.

Текст небольшой Си-программы весь помещается в простой (без разметки) текстовый файл с расширением `".c"`. Вверху файла, начиная с самой левой позиции, пишут `#include <stdio.h>`. Это дает возможность пользоваться функцией `printf(…)`, выводящей на экран содержимое переменных программы. Самая простая программа состоит из одной функции, которая запускается при старте программы. Ее обязательно надо назвать `main` (англ. главный). После слова `main` следует пара пустых круглых скобок, которые говорят о том, что этой функции ничего не передается извне. Все содержимое функции (как говорят, ее *тело*, англ. function body) заключено в пару фигурных скобок и завершается оператором выхода из программы `return 0;`. В начале тела функции вводятся необходимые переменные. Всякая переменная должна иметь свой тип и имя. Важнейшие пока типы – это `int` (целое число, англ. integer) и `double` (действительное число). Определение переменной имеет общий вид:

```
тип_переменной имя_переменной=начальное_значение;
```

Далее следует пример простейшей программы, выводящей на экран строку текста, содержащую число 2014, вставляемое из переменной `k` в заданное место текста при помощи формата (способа распечатки) `%d`. Переменная `k` в функции вывода на экран `printf` размещена через запятую после выводимого текста, заключенного в двойные кавычки:

```
#include <stdio.h>

int main()
{
```

```
int k=2014;
printf("Здравствуй, мир! Я - программа %d года.",k);
return 0;
}
```

В7. Символы, лексемы и разделители

Символ (англ. character) – одиночный знак, не несущий сам по себе, в отрыве от контекста, никакого смысла, но служащий средством записи информации. В этом он подобен букве, цифре или знаку препинания обычного (естественного, не машинного) языка. Совокупность всех допустимых в данной системе записи символов называется *алфавитом*. Например, алфавит какого-либо человеческого языка – это всевозможные буквы этого языка, из которых складываются слова, алфавит записи чисел – это цифры; алфавит арифметических операций – их знаки '+', '-', '•', '/' и т.д. Во многих современных языках для удобства чтения вводятся вспомогательные разделительные символы, например пробел ' ', запятая ',', точка '.', точка с запятой ';', двоеточие ':', дефис '-', а также различаются прописные и строчные буквы. Допустимыми символами языка Си являются:

- ◊ все буквы латинского (английского) алфавита, при этом прописные и строчные буквы всегда считаются различными;
- ◊ арабские цифры 0, 1, ..., 9;
- ◊ знаки арифметических операций: сложение '+', вычитание '-', умножение '*' (звездочка вместо обычно принятых на письме точки или косого крестика), деление (прямая косая черта, прямой слеш, англ. slash) '/';
- ◊ знаки операций сравнения и присваивания: '<', '=', '>';
- ◊ левые и правые скобки: '(', ')', '[', ']', '{', '}' всегда используются парно для визуального и смыслового выделения фрагмента текста;
- ◊ пробельные символы: пробел ' ', табуляция (получается в текстовом редакторе нажатием на клавишу Tab, отображается в тексте в виде нескольких слитных пробелов, обозначается '→|'), переход на новую строку (получается в текстовом редакторе нажатием на клавишу Enter, обозначается '↵');

◊ служебные символы:
тильда (волна, англ. tilde) '~';
восклицательный знак (англ. exclamation sign) '!';
вопросительный знак (англ. question sign) '?';
решетка (диез, октоторп, англ. octothorp, number sign)[1] '#';
знак процента (англ. percent sign) '%';
крышка (циркумфлекс, англ. hat, circumflex) '^' (в некоторых системах – стрелка вверх '↑');
амперсанд (коммерческое 'и', англ. ampersand) '&';
подчеркивание (англ. underscore) '_';
обратная косая черта (обратный слеш, англ. back slash) '\';
вертикальная черта (англ. pipe) '|' (в некоторых шрифтах изображается с разрывом посередине);
запятая (англ. comma) ',';
точка (англ. point) '.';
точка с запятой (англ. semicolon) ';';
двоеточие (англ. colon) ':';
двойная кавычка (англ. double quotation mark) '"';
одинарная кавычка (апостроф, англ. single quotation mark) '''.

Все прочие символы, которые можно ввести с клавиатуры, в том числе буквы национального алфавита (прописные и строчные буквы также различаются), имеют очень ограниченное использование, связанное с пользовательским интерфейсом создаваемой программы и комментариями программиста.

Как во многих письменных человеческих языках, символы делятся на описательные и разделительные. Некоторые символы могут играть обе эти роли в зависимости от контекста. Разделительными символами (кратко – *разделителями*) могут служить знаки операций, скобки, служебные и пробельные символы. Все остальные символы – только описательные.

Лексемами называются имеющие самостоятельное осмысленное значение сочетания символов (лексема может состоять из одного символа). С точки зрения препроцессора и компилятора лексемы являются нерасчленимыми «кирпичиками», из которых состоит текст программы на алгоритмическом языке. К лексемам относятся все разделительные символы, кроме пробельных, а также состоящие из латинских букв и цифр име-

[1] Строго говоря, диез и октоторп – разные символы, но в программистской литературе их считают одним и тем же вследствие сходного начертания.

на и значения констант, имена переменных и зарезервированные слова, описывающие конструкции языка.

В8. Атомы языка Си

Лексемы в Си бывают следующих видов:

- ◊ зарезервированные управляющие конструкции языка, обозначения операций и ключевые слова – все они задают базовые типы данных, структуру и ход выполнения программы. Их написание и использование жестко фиксированы;
- ◊ имена переменных, констант, функций и макроопределений – вводимые программистом последовательности латинских букв, цифр и символов подчеркивания, причем первым символом может быть только буква или символ подчеркивания[1]. Не допускается совпадение имен с зарезервированными ключевыми словами. Практически все системы программирования допускают имена не длиннее 32 символов, а большинство современных средств разработки не накладывает ограничений на их длину;
- ◊ значения констант весьма разнообразны, как и форма их представления;
- ◊ символы операций – описывают способ получения результата выражения в процессе работы программы;
- ◊ комментарии, которыми программист может оснащать программу с целью пояснения коллегам ее работы.

Операция (англ. operator) – единичный, не разделимый на более мелкие шаг вычислительного процесса. Результатом выполнения операции всегда является некоторое *значение* – число или указатель. К операциям относятся: арифметические действия, сравнение двух величин, присваивание переменной ее нового значения, взятие или разадресация указателя и другие элементарные действия. Операции являются базовыми элементами, из которых собираются более сложные конструкции, например выражения. Большинство операций выполняется за один такт процессора, поэтому они не могут быть «разорваны» на части в

[1] Многие системы программирования воспринимают символ подчеркивания в начале имени переменной, константы или функции как обозначение того факта, что такие объекты обладают специальными свойствами.

смысле не только синтаксиса языка, но и минимального кванта вычислительного процесса. Именно в последнем случае говорят об атомарности (англ. atomicity) операций при вычислениях[1].

Операнд (англ. operand) – константа, переменная или состоящее из них выражение, значение которого на момент употребления можно однозначно определить. Операнд является необходимой составной частью операции или оператора. Для всякой операции фиксировано число ее операндов. При этом говорят, что операция осуществляется над одним или более операндами. Это означает, что результат операции определяется значениями ее операндов и только ими. Например, в выражении `3*a` константа `3` и переменная `a` являются операндами операции умножения.

Выражение (англ. expression) – допустимое в языке программирования осмысленное сочетание операций и операндов, приводящее к получению искомого значения (число или указатель). Выражения состоят из операций, операндами которых выступают константы, переменные и результаты вычисления других выражений, называемых вложенными по отношению к данному, объемлющему выражению, а также вызовов функций. Последовательность использования операций для вычисления значения выражения формируется при помощи жестко заданных приоритетов этих операций по умолчанию и применения круглых скобок, в точности так, как в математике. Выражения являются одной из важнейших составных частей программы. Приведем примеры простых выражений: `a/b`, `x*y+v*w-16.9`, `g++`, `a*(b*sin(k*m)+c)`. Все они при подстановке значений операндов дают некоторое число в качестве результата.

Оператор (англ. statement) – инструмент управления ходом выполнения программы, часто в зависимости от значений данных ему операндов. С помощью операторов записывается управляющая логика алгоритма программы, поскольку даже в простых программах обычно не удается ограничиться последовательным вычислением и присваиванием значений ряда выражений. Подобно выражениям, операторы могут вкладываться друг в друга для получения более сложного *составного оператора*.

---

[1] Эта атомарность, однако, не имеет никакого отношения к атомам языка.

B9. Зарезервированные слова

Минимальный стандартный список зарезервированных слов таков: `auto`, `break`, `case`, `char`, `complex`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `inline`, `int`, `long`, `main`, `register`, `restrict`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`. В некоторых реализациях этот набор расширен и добавлены новые зарезервированные слова, например `_Alignas`, `_Alignof`, `_Atomic`, `_Bool`, `_Complex`, `exit`, `_Generic`, `_Imaginary`, `_Noreturn`, `ptrdiff_t`, `size_t`, `ssize_t`, `_Static_assert`, `_Thread_local`. Рекомендуется избегать введения собственных переменных, констант, функций и макроопределений с названиями, совпадающими даже с дополнительными зарезервированными словами, а также с именами переменных, констант, функций и макроопределений стандартной библиотеки.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Найдите разделители в тексте простейшей программы, приведенном в § B6.
2. Отыщите зарезервированные слова в тексте простейшей программы, приведенном в § B6.
3. Найдите имя переменной в тексте простейшей программы, приведенном в § B6.
4. Освойте процесс получения простейшей программы исполнимого кода из исходного текста в вашей системе программирования. Откомпилируйте и запустите ее.

ГЛАВА 1 БАЗОВЫЕ ТИПЫ ДАННЫХ И ИХ ДВОИЧНОЕ ПРЕДСТАВЛЕНИЕ

1.1. Беззнаковые целые числа

В подавляющем большинстве современных компьютеров используется двоичное представление всех объектов в памяти – и программ, и данных. Это означает, что всякий фрагмент исполнимого процессором кода или обрабатываемых данных является последовательностью двоичных цифр – нулей и единиц. Как это оказывается возможным? Очевидно, что все нужные объекты можно пронумеровать целыми числами. Ответим на вопрос: как целые числа представить в двоичном виде (англ. binary representation)? Обозначим жирной латинской буквой с верхним индексом $\mathbf{a}^K$ последовательно идущие в памяти K нулей и единиц, кодирующих вместе один объект, и назовем такую форму его представления двоичным вектором длиной K. Элемент вектора $\mathbf{a}^K$, стоящий в нем на k-м месте, обозначим a_k (k-й бит двоичного представления объекта $\mathbf{a}^K$). Очевидно, всякий a_k может принимать два значения – 0 или 1, $k = 0, 1, ..., K - 1$.

Всякое неотрицательное целое число $N < 2^K$ может быть однозначно представлено в виде:

$$N = \sum_{k=0}^{K-1} a_k 2^k .$$

При записи двоичных чисел a_{K-1} называют старшим разрядом и располагают слева, подобно тому, как более «весомые» цифры в десятичной записи числа стоят слева. Младший разряд a_0 размещают крайним справа, он отвечает за наиболее низкую степень числа два: $2^0 = 1$. Преобразование из десятичного представления в двоичное можно выполнить несколькими способами. Самым наглядным является последовательное деление числа пополам и сохранение остатков:

а) делят исходное число на два. Остаток может принимать значение 0 или 1. Его записывают в старший разряд двоичного числа. Частное от деления запоминают для следующего шага;

б) делят на два результат предыдущего деления и остаток размещают в следующем разряде, соответствующем на единицу меньшей степени числа 2. Так поступают до тех пор, пока в результате очередного деления не получится число, меньшее 2. Его записывают в самый младший разряд и останавливают процедуру.

Например, перевод десятичного числа 379_{10} в двоичную систему дает результат 101111011_2, где нижний индекс указывает на способ представления числа, называемый *системой счисления*. Дополнение числа нулями слева, очевидно, не меняет результат: 000000101111011, поскольку при соответствующих степенях двойки коэффициенты остаются нулевыми.

Обратное преобразование из двоичного вида в десятичный тоже можно осуществить по-разному. Необычный, но наглядный способ соответствует известной из математики схеме Горнера:

а) можно пропустить все старшие нулевые разряды, так как они не влияют на значение, как и в случае десятичного представления;

б) возьмем самый старший ненулевой разряд двоичного числа. Если он же является младшим, это и есть десятичное значение. Конец преобразования;

в) пока имеется следующий, более младший разряд, умножим текущий результат на 2, добавим к результату умножения содержимое этого, более младшего разряда и снова запомним результат.

Арифметические действия над числами в двоичной форме производятся по тем же правилам, что и для десятичных чисел. Единственное отличие обусловлено тем, что в двоичном представлении всего две цифры. Поэтому, если текущий результат превосходит единицу, осуществляется перенос в следующий по старшинству разряд, или, если он оказывается менее нуля, происходит заимствование единицы из старшего разряда.

Например: **1.** Вычтем в двоичном представлении $326_{10}=101000110_2$ из $412_{10}=110011100_2$:

```
                 110011100
                -101000110
заем из старшего разряда:  001010110  (0 = нет заема, 1 = есть заем)
результат:                 001010110₂ = 86₁₀.
```

2. Умножим $45_{10} = 101101_2$ на $9_{10} = 1001_2$. Помня о коммутативности умножения, умножим, как в математике, число с бóльшим числом единиц на число с меньшим числом единиц. Умножим 101101 сначала на 1_2, что оставляет первый сомножитель неизменным, затем на 1000_2, что эквивалентно сдвигу первого сомножителя влево на три позиции с дописыванием нулей справа, и сложим результаты:

```
   101101₂ x 0001₂ = 000101101₂
  +101101₂ x 1000₂ = 101101000₂
перенос в старший разряд: 000101000  (0 = нет переноса, 1 = есть перенос)
результат:                110010101₂ = 405₁₀.
```

1.2. Знаковые целые числа

Введем в двоичное представление числа еще один крайний бит, который разместим слева, назовем его знаковым и обозначим a_K. Он хранит информацию о знаке числа, что дает возможность представить произвольное целое число N:

$$N = (-1)^{a_K} \sum_{k=0}^{K-1} a_k 2^k .$$

Такое представление, называемое за простоту *прямым кодом*, обладает заметным неудобством для процессора: необходимо по-разному действовать при сложении и вычитании чисел в зависимости от того, совпадают их знаки или нет. Поэтому реальные электронные вычислители пользуются другим представлением целых знаковых чисел – в так называемом *дополнительном коде* (англ. two's complement). Для неотрицательных чисел дополнительный код совпадает с прямым кодом. Чтобы перевести отрицательное число в дополнительный код,

к нему добавляют 2^K в двоичном виде. Поскольку 2^K превышает по абсолютной величине даже самое большое по модулю хранимое отрицательное число, результат сложения получается заведомо положительным. Процессор умеет его отличить от «настоящего положительного» числа. Дело в том, что у знакового целого старший знаковый разряд $a_K = 0$, когда его значение положительно, и $a_K = 1$ при отрицательном значении. Операция сложения отрицательного числа с 2^K в двоичной арифметике чрезвычайно проста:

а) записывают в двоичном виде *модуль* отрицательного числа; так, для числа –20 при 8-битовом хранении: $20_{10} = 00010100_2$;

б) заменяют все двоичные цифры на обратные (при этом говорят, что число превращают в обратный, инверсный код): $\overline{0}\,\overline{0}\,\overline{0}\,\overline{1}\,\overline{0}\,\overline{1}\,\overline{0}\,\overline{0}_2 = 11101011_2$. Эта операция осуществляется для каждой двоичной цифры независимо, как говорят, побитово. Побитовая инверсия положительного двоичного числа, не превышающего $2^{K-1}-1$, в двоичной арифметике эквивалентна вычитанию этого числа из $(2^{K-1}-1)$.

в) добавляют к получившемуся результату единицу (не побитово, а арифметически, т.е. по правилам двоичной арифметики, с возможными переносами между разрядами): $11101011_2 + 00000001_2 = 11101100_2$. Это и есть дополнительный код числа –20.

Все эти операции процессор способен выполнять значительно быстрее, чем отдельно обработку знаков и затем выбор между сложением и вычитанием модулей. Сложение чисел в дополнительном коде происходит единообразно для положительных и отрицательных чисел.

Например:
$14_{10} + (-20_{10}) = 00001110_2 + 11101100_2 = 11111010_2$.

Наличие в знаковом разряде единицы говорит о том, что результат сложения отрицателен. В этом случае для возврата к десятичному представлению модуля результата выполняют преобразования *а*–*в* в обратном порядке. Сначала вычитают единицу (а это всегда возможно, так как старший знаковый разряд содержит единицу и есть откуда «занимать» при необходимости):

$11111010_2 - 00000001_2 = 11111001_2$.

Далее обращаем все биты: $\overline{1}\,\overline{1}\,\overline{1}\,\overline{1}\,\overline{1}\,\overline{0}\,\overline{0}\,\overline{1}_2 = 00000110_2 = 6$. С учетом знакового бита результат отрицателен и равен –6.

1.3. Действительные числа

Действительные числа в памяти компьютера хранят в так называемых форматах с фиксированной (англ. fixed point) и плавающей (англ. floating point) точкой[1]. Форматы с фиксированной точкой довольно просто реализуются при помощи целых чисел, но их рассмотрение лежит вне рамок данного пособия, поскольку не имеет прямого отношения к языку Си. В числе с плавающей точкой можно выделить три части разного назначения – знаковый бит S, мантиссу $0 \le M < 2$ и целочисленную экспоненту E. Пусть состоящие из нулей и единиц векторы $a^K \equiv \{a_1, a_2, \ldots, a_K\}$ и $b^L \equiv \{b_1, b_2, \ldots, b_L\}$ – соответственно двоичные представления мантиссы и экспоненты, тогда действительное число N можно в общем случае приближенно закодировать так:

$$N = (-1)^s \left(\sum_{k=1}^{K} a_k 2^{-k+1} \right) 2^E,$$

где $E = \sum_{l=1}^{L} b_l 2^l - 2^{L-1}$.

Подстановка дает

$$N = (-1)^s \sum_{k=1}^{K} a_k 2^{-k+1} 2^{\sum_{l=1}^{L} b_l 2^l - 2^{L-1}}.$$

Способ укладки и интерпретации битов в переменных с плавающей точкой в большинстве ныне действующих вычислительных средств подчиняется принятой в 2008 г. новой редакции международного стандарта IEEE floating point standard 754. Согласно стандарту, расположение битов таково:

---

[1] В большинстве англоязычных стран целая часть числа на письме отделяется от дробной части точкой, а не запятой, как принято в русской традиции.

бит знака S	L битов экспоненты: ближе к знаковому биту расположен старший, $l = L$	K битов мантиссы: ближе к экспоненте расположен старший, $k = K$

Правило интерпретации битов:

если $M = 0$ и $E = 0$, то $N = 0$,
иначе, если $M = 0$ и $E = 0$ (т.е. $\mathbf{b}^L$ состоит из одних единиц), такое число кодирует $\pm\infty$ в зависимости от значения S (пишут inf, Inf, от англ. infinity);
иначе, если $M \neq 0$ и $E = 0$ (т.е. $\mathbf{b}^L$ состоит из одних единиц), это означает, что содержимое переменной не является допустимым числом (пишут NaN, NAN, от англ. not a number);
иначе это обычное число N, расшифровываемое по приведенной выше формуле.

Для короткого действительного типа `float` характерно $L = 8$, $K = 23$, а для представления действительных чисел с двойной точностью `double` типично $L = 11$, $K = 52$. Реализация введенного позже типа `long double` различна на разных платформах. Встречаются варианты $L = 11$, $K = 52$; $L = 15$, $K = 64$; $L = 15$, $K = 112$ и др.

Важно отметить, что какой бы тип с плавающей точкой ни использовался, действительное число им представляется в общем случае приближенно, с некоторой погрешностью. На практике это означает, что некоторые, на первый взгляд правильные, операции с числами с плавающей точкой могут приводить к результату с низкой точностью или вовсе бессмысленному, как говорят, численно нестабильному (англ. numerically unstable). Таково, например, вычитание очень близких действительных чисел. Его результат обладает тем более низкой точностью (которую часто характеризуют количеством значащих цифр, англ. significant digits, valid digits), чем ближе друг к другу значения уменьшаемого и вычитаемого. Подобного рода явления нередки, очень опасны и именуются потерей точности (англ. precision loss), а их детальное рассмотрение выходит далеко за рамки языка Си.

С представлением символов (букв, цифр, знаков) дело обстоит значительно проще: все допустимые символы упорядочиваются (подобно тому, как это делается в алфавите), и каждому из них присваивают числовой код – от нуля до некоторого

максимального значения, определяемого объемом алфавита. Этот код сохраняют в виде целого числа, которое переводят в двоичный вид, как для «настоящих» чисел. Аналогично поступают с командами процессора компьютера, каждой из которых тоже однозначно сопоставлен свой двоичный вектор.

Поскольку многие типы данных требуют для своего хранения довольно длинных двоичных векторов (от 8 до 128 бит и даже более), удобно группировать по несколько битов так, чтобы имелась возможность обращаться к ним не последовательно друг за другом, а одновременно ко всей группе как к единому целому. Получили распространение четыре размера такой группы: полубайт (нибл, англ. nibble, 4 последовательно идущих бита), байт (англ. byte, 8 последовательно идущих битов), токен (англ. token, обычно 16 последовательно идущих битов) и машинное слово (англ. machine word, computer word, full word). Длина машинного слова зависит от типа процессора и обычно составляет от 16 до 128 последовательно идущих битов. В программистской практике чаще других используются байт и машинное слово, поскольку первая мера данных является стандартной машинно независимой, а вторая характеризует наиболее естественный для процессора и потому быстро обрабатываемый размер данных.

1.4. Базовые типы данных

Понимая двоичное представление различных видов информации, легко увидеть разницу между традиционными базовыми типами данных в языке Си[1].

Символьный тип (`char`) вводится описателями типа (разновидность ключевых слов) `char`, `signed char`, `unsigned char` и предназначен для хранения одного любого символа алфавита или небольшого числа. При этом дополнительный описатель `signed char` (`unsigned char`) указывает, что символ хранится как знаковое целое число (как неотрицательное целое число), а «голый» описатель `char` оставляет выбор знакового или беззнакового

[1] Введенные относительно недавно специальные целочисленные типы для операций с размерами объектов и указателями рассматриваются отдельно в § 1.7.

способа хранения на усмотрение компилятора. Для хранения значения всякой переменной символьного типа отпускается один и тот же объем памяти вне зависимости от ее значения. Это необходимо, чтобы в переменную потенциально мог поместиться любой символ. Сколько именно памяти требуется для хранения символа, зависит от конкретной системы программирования и обычно составляет 1 или 2 байта. Этот объем, как и для всех прочих целых чисел, не зависит от того, выбран знаковый или беззнаковый вариант типа. Точное значение, выраженное в числе байт, можно выяснить с помощью выражения `sizeof`.

Короткий целый тип (`short`) вводится описателями `short` (аналог `signed short`) и `unsigned short` и предназначен для хранения относительно небольших целых чисел. Дополнительный описатель `unsigned` оказывает то же действие, что для типа `char`. Принципиальное отличие от символьного типа состоит в том, что `short` без описателя `unsigned` – это всегда *знаковое* число, которое может принимать отрицательные значения. Подобно типу `char`, объем памяти для типа `short` не стандартизован, наиболее распространенным значением является 2 байта. Принципиально выполняется лишь одно соотношение: `sizeof(short)≥sizeof(char)`.

Целый тип (`int`) вводится описателями `int` (аналоги `signed` и `signed int`) или `unsigned` (аналог `unsigned int`) в зависимости от способности принимать отрицательные значения и предназначен для хранения относительно больших целых чисел. Переменная типа `int` – всегда знаковое число, если явно не указан спецификатор `unsigned`. Как и для всех остальных типов, объем памяти для типов `int` и `unsigned` не предопределен, часто это 2 или 4 байта. Гарантируется только соотношение размеров: `sizeof(int)≥sizeof(short)`.

Длинный целый тип (`long`) также существует в двух вариантах с описателями `long` (аналог `signed long`) или `unsigned long` в зависимости от способности принимать отрицательные значения и предназначен для хранения длинных целых чисел. Переменная типа `long` – всегда знаковое число. Объем памяти составляет наиболее часто 4 байта. Более строго: `sizeof(long)≥sizeof(int)`.

Сверхдлинный целый тип (`long long`) введен относительно недавно. Его появление связывают с распространением 64-раз-

рядных процессоров, поэтому его длина равна 8 байт почти во всех компьютерах, где этот тип поддерживается. Подобно более коротким типам, также имеются два варианта — `long long` и `unsigned long long` и справедливо неравенство `sizeof(long long)≥sizeof(long)`.

Тип с плавающей точкой одинарной точности (`float`) предусматривает хранение произвольных вещественных чисел, значение которых может быть представлено в соответствующем формате. Длина обычно 4 байта.

Тип с плавающей точкой двойной точности (`double`) отличается от `float` бóльшими длинами мантиссы и экспоненты, что увеличивает как точность представимых величин, так и их диапазон. Длина обычно 8 байт. По аналогии с целыми типами, гарантируется немногое: `sizeof(double)≥sizeof(float)`.

Тип с плавающей точкой повышенной точности (`long double`) введен относительно недавно по мере появления и внедрения 64-разрядных, 128-разрядных и 256-разрядных процессоров и характеризуется еще более высокой точностью и более широким диапазоном поддерживаемых значений. Длина обычно 8, 10, 12 или 16 байт. К сожалению, данный формат до сих пор не поддерживается некоторыми процессорами и системами программирования (компиляторами и сборщиками), поэтому, руководствуясь соображениями переносимости программ с одной системы на другую, к его использованию следует относиться очень осторожно. Соотношение длин: `sizeof(long double)≥sizeof(double)`.

Тип указателя (англ. pointer) занимает особое место среди типов данных языка Си. Он позволяет обращаться к элементу любого заданного типа не непосредственно, а при помощи ссылки на него. По сути дела указатель является абстрактным адресом в памяти компьютера, по которому расположен нужный элемент данных. В зависимости от реализации этот адрес в форме особого беззнакового целого числа может совпадать с физическим адресом в памяти, а может не иметь с ним ничего общего. Не следует делать какие-либо предположения о конкретных значениях и свойствах указателя, а также о его устройстве и размещении в памяти, поскольку это ведет к непереносимости программ с платформы на платформу и многочисленным ошибкам при

программировании. Так, для некоторых компьютеров неверно распространенное представление о том, что указатель — это `unsigned int`. Более того, реализация самого указателя на различные типы данных может быть различной. Главная причина отказа от практики привязки указателей к адресам и использования их конкретных значений в программе состоит в том, что эта привязка ничем не оправдана: язык Си оснащен весьма мощным набором средств работы с указателями, чтобы всегда обходиться без этого. Указатели необходимы для удобной адресации больших массивов данных и эффективного обмена информацией между частями программы.

Типичные (но не обязательно соответствующие данной компьютерной платформе) свойства базовых типов символьных и числовых данных приведены в табл. 1.1.

Таблица 1.1. Типичные свойства базовых типов символьных и числовых данных

Название формата	Характер значений	Стандартная длина, байт	Типичная длина, байт	Типичный диапазон значений	Точность представления значения
`char`	Целый	Самая малая из всех типов	1 2	–128...127, или 0...255 –32768...32767 или 0...65535	Точно
`signed char`	Целый знаковый	Как у `char`	1 2	–128...127 или –32768...32767	Точно
`unsigned char`	Целый беззнаковый	Как у `char`	1 2	0...255 или 0...65535	Точно
`short`	Целый знаковый	Не меньше, чем у `char`	2	–32768...32767	Точно
`unsigned short`	Целый беззнаковый	Как у `short`	2	0...65535	Точно

Окончание табл. 1.1

Название формата	Характер значений	Стандартная длина, байт	Типичная длина, байт	Типичный диапазон значений	Точность представления значения
`int`	Целый знаковый	Не меньше, чем у `short`	2 4	–32768...32767 или –2147483648... 2147483647	Точно
`unsigned`	Целый беззнаковый	Как у `int`	2 4	0...65535 или 0...4294967295	Точно
`long`	Целый знаковый	Не меньше, чем у `int`	4	–2147483648... 2147483647	Точно
`unsigned long`	Целый беззнаковый	Как у `long`	4	0...4294967295	Точно
`long long`	Целый знаковый	Не меньше, чем у `long`	8	–9223372036854775808 ... 9223372036854775807	Точно
`unsigned long long`	Целый беззнаковый	Как у `long long`	8	0... 18446744073709551615	Точно
`float`	Вещественный	Не меньше, чем `long`	4	$K = 23$, $L = 8$, максимальное значение $\pm(2^{128} - 2^{104}) \approx \pm 3{,}4028235 \times 10^{38}$	Относительная погрешность $\pm 2^{-126} \approx \pm 1{,}17 \times \times 10^{-38}$
`double`	Вещественный	Не меньше, чем `float`	8	$K = 52$, $L = 15$, максимальное значение $\pm(2^{1024} - 2^{971}) \approx \pm 1{,}79769313486 \times 10^{308}$	Относительная погрешность $\pm 2^{-1022} \approx \approx \pm 2{,}225 \times \times 10^{-308}$
`long double`	Вещественный	Не меньше, чем `double`	10	$K = 64$, $L = 15$, максимальное значение $\approx \pm 1{,}1897 \times 10^{4932}$, встречаются и другие варианты	Относительная погрешность $\approx \pm 3{,}362 \times \times 10^{-4932}$

1.5. Модификаторы и спецификаторы типов

Описатели базовых типов содержат информацию об объеме памяти, занимаемом переменными и константами, и способе их представления. Иногда эти сведения требуется дополнить указанием на дополнительные возможности и особенности хранилища, для чего используются модификаторы (англ. extended type specifier) и спецификаторы хранения (расширенные описатели, англ. storage-class specifier), которые при необходимости могут располагаться перед основным описателем типа.

Модификаторы `signed` и `unsigned` отвечают за возможность хранить в константах и переменных отрицательные числа, а модификатор `const` — за невозможность случайно повредить значение.

Спецификаторы хранения: `auto`, `extern`, `register`, `static`, `volatile` позволяют задавать способ размещения переменной или константы в памяти компьютера и вытекающие отсюда ее свойства.

Знаковые модификаторы `unsigned` и `signed`. Для возможности хранить в переменной или константе положительные и отрицательные значения или только положительные (но зато почти в 2 раза большего диапазона) используются модификаторы соответственно `signed` и `unsigned`.

Например:

```
signed char x=-18;
unsigned long mask=0x00FF0000;
```

При этом:

- `char` без модификатора может быть как знаковым, так и беззнаковым в зависимости от компилятора и его настроек. Это можно выяснить в тексте самой программы и логично сделать силами препроцессора, поскольку знаковость `char` является свойством компилятора, а не платформы, на которой будет исполняться программа;
- `short`, `int`, `long`, `long long` без модификатора всегда знаковые;

- `float, double, long double` не могут иметь модификатора `signed`, будучи всегда знаковыми;
- модификаторы `signed` и `unsigned`, если присутствуют, стоят непосредственно перед именем базового типа. Все прочие модификаторы и спецификаторы хранения пишутся левее.

Константный модификатор `const` указывает на то, что вводимый объект данного типа не может быть впоследствии изменен (является константным). Отсюда следует, что присвоить единственное неизменное значение константе можно лишь в момент ее создания, при определении. Все не связанные с попытками поменять его значение действия над этим объектом по-прежнему возможны в любое время. Константность объекта нельзя включить, а затем выключить, это его постоянное свойство. Модификатор `const` применим как к любым встроенным, так и к любым производным (введенным программистом) типам.

Например:

```
struct xxl {double g; short h; double s;};
signed char a[]="Просто строка текста";
const signed char x=-18, *pa=&a[0];
const struct xxl p_8={1.7, 2, -123.456};
```

Стандартное место размещения модификатора `const` левее знакового модификатора, если он есть.

Спецификатор хранения `register`. Наиболее быстрой памятью компьютера являются регистры его процессора, и именно их значениями процессор преимущественно оперирует, когда производит вычисления. Имеются и ограничения:

- регистр процессора нельзя адресовать иначе как по его имени, поэтому невозможно определить указатели на размещенные в регистрах переменные и константы;
- если предполагается, что через вводимую переменную будет осуществляться обмен данными между процессором и каким-либо другим устройством (другим процессором, портом, контроллером и т.д.), размещение этой переменной в регистре исключает ее видимость извне данного процессора;
- обычно регистров процессора очень мало (несколько десятков—несколько сотен), и разместить в них все данные програм-

мы, как правило, не удается. Поэтому можно только рекомендовать компилятору поместить в регистры малую часть наиболее активно участвующих в вычислениях данных. Это могло бы ускорить программу, так как исключается потребность во множестве обращений к относительно медленной оперативной памяти.

Указание спецификатора `register` (обычно перед самым левым из модификаторов типа переменной или константы) означает просьбу программиста расположить данный объект в регистре, например: `register unsigned long i;`

Очевидно, запрашиваемый объект должен полностью помещаться в одном регистре процессора, в связи с чем спецификатор хранения `register` разумно применять только по отношению к малым типам данных. Однако при этом нет гарантий, что компилятор последует этой рекомендации. Поэтому нельзя делать никаких предположений об истинном месте нахождения данных со спецификатором хранения `register`. К регистровым данным неприменим указатель. В стандарте языка Си нет четких указаний на совместимость `register` с другими спецификаторами хранения. Однако во многих практических системах программирования предполагается, что он может применяться только к автоматическим переменным и константам или присутствовать в списке формальных параметров функции, указывая на просьбу программиста передавать соответствующий параметр в тело функции через регистр процессора.

Спецификатор хранения `auto`. Если программистом не написано иного, обычно переменные и константы располагаются на стеке (англ. stack). Это означает, что при выходе из блока, где они определены, они безвозвратно утрачиваются. В большинстве случаев такое поведение приемлемо и даже удобно, поэтому такой способ выделения памяти используется по умолчанию и называется *автоматическим*. Если необходимо явно указать стековый способ хранения переменной или константы, перед ее определением размещают слово `auto`.

Например:

```
#define PI 3.1415926535
auto float x=4*PI;
```

Всякий раз при прохождении через этот фрагмент кода на стеке будет отводиться место для переменной `x` и осуществляться ее инициализация посчитанным наперед значением `4*PI`.

Спецификатор хранения `extern`. Часто в больших программах бывает нужно ввести *глобальные* переменные или константы, область существования и видимости (англ. scope) которых не ограничивается одним файлом, а распространяется на несколько файлов исходного текста (англ. source code) программы. Если при этом определить этот глобальный объект только в одном из файлов, он не будет виден из других. Если его определить в каждом из файлов, где он нужен, во всяком из соответствующих объектных файлов будет его независимая копия. При попытке собрать объектные файлы воедино сборщик (линковщик, компоновщик, редактор связей, линкёр, англ. linker, linkage editor) столкнется с многократным определением одного и того же глобального объекта в разных файлах. Эти определения он не в состоянии ни соединить, ни обособить друг от друга, так как компилятором уже сгенерирован соответствующий код порождения нескольких независимых копий объекта.

Данная проблема разрешается путем однократного определения (англ. definition) объекта и повторения его объявления (декларации, англ. declaration) в других файлах со словом `extern`. Этот спецификатор превращает определение переменной или константы в ее декларацию, предотвращая выделение под нее памяти и помещая ее имя в таблицу внешних связей для сборщика. Именно по этой причине `extern` несовместим со спецификаторами `auto`, `register` и `static`, сопутствующими определению объекта, а не его декларации.

Например:
файл `my_1.c`:

```
...
unsigned long Counter;
...
```

файл `my_2.c`:

```
...
extern unsigned long Counter;
...
```

При обработке файла `my_1.c` компилятор выделяет место под переменную `Counter`, а при компиляции `my_2.c` эта же переменная лишь декларируется как внешняя, что позволяет сборщику успешно осуществить компоновку исполнимого файла из двух объектных модулей.

Спецификатор хранения `static`. Выделение памяти на стеке при входе в блок и ее утилизация при выходе из него создает накладные расходы. Если гарантируется, что одновременно в блок может войти только один модуль программы, местные временные переменные и константы можно сделать неудаляемыми, т.е. они существуют с момента первого захода в блок (фактически с момента запуска программы) до конца программы, а доступны только изнутри этого блока. Более того, между двумя следующими друг за другом входами в блок эти данные гарантированно сохраняют свои значения неизменными. Они позволяют хранить состояние данных от предыдущего пребывания в данном блоке. Переменные и константы с такими свойствами называются *статическими*. Для их определения перед типом размещают спецификатор хранения `static`.

Например:

```
...
{ static double last_value=0; ... }
...
```

Важно помнить, что инициализация статических объектов выполняется ровно один раз – в момент первого захода в блок. В зависимости от реализации компилятора память для статической переменной может выделяться в момент запуска программы либо в специально отведенной области статических данных, либо на самой вершине стека (англ. stack top), либо в основании кучи (англ. heap base). Никаких предположений о месте размещения статических переменных делать нельзя. С использованием статических переменных связана опасность неумышленного одновременного или последовательного обращения к одной и той же переменной из одного и того же кода, запускаемого на разных процессорах. При этом значение переменной может быть изменено незапланированным образом. В то же время использование статической переменной для

обмена между различными фрагментами кода ненадежно, поскольку о таком ее использовании «не знает» компилятор и нет возможности гарантировать правильную передачу информации в результате выполняемой им оптимизации кода. В частности, компилятор может на некоторое время перекладывать содержимое статического объекта в регистры или кэш-память, и в эти периоды в оперативной памяти будет длительное время находиться устаревшее значение.

Функция, которую можно корректно вызвать независимо и одновременно на одном или различных процессорах для обработки разных данных, называется *реентерабельной* (реентерантной, англ. reenterant). Это свойство является в высшей степени полезным, а зачастую и необходимым в современных параллельных вычислительных системах. Неосмотрительное использование спецификатора хранения `static` способно лишить функцию этого полезного качества, но позволяет получить для нее выигрыш в быстродействии.

Спецификатор `static` применим и вне блоков – там, где создаются глобальные объекты. Цель `static` при этом иная – сделать так, чтобы объект был виден только внутри данного файла исходного текста. Дело в том, что по умолчанию всякий объект, определенный вне блоков, полагается глобальным не только в рамках (англ. scope) данного программного модуля, но и всей программы. Компилятор автоматически попытается оповестить о его существовании все модули, которые потом будут объединяться в программу. Но в этих внешних модулях могут присутствовать «собственные» одноименные объекты, что приведет к *конфликту имен* (англ. name clash, naming conflict) и ошибке сборки программы. Избежать конфликта имен, искусственно сужая область видимости объекта до одного файла, – в этом вторая миссия спецификатора `static`. Все глобальные для данного файла объекты, которые не планируется передавать в программные модули в других файлах, рекомендуется объявлять статическими.

Спецификатор хранения `volatile` (англ. переменчивый) отключает всевозможные виды оптимизации кода компилятором по отношению к данной переменной или константе, что, как правило, используется для обмена информацией между несколькими устройствами или (реже) программными модулями

через этот объект в памяти. Только указание `volatile` гарантирует, что изменение значения переменной одним модулем может быть сразу обнаружено другим, исполняемым нередко даже на другом процессоре. Поскольку часто исходные тексты общающихся друг с другом модулей размещены в разных файлах, сочетание `extern volatile` является распространенным.

Опыт профессионального использования `volatile` таков, что этот спецификатор практически обязателен при работе с аппаратными средствами компьютера, отображенными в его адресное пространство. В то же время для организации обмена информацией между программными модулями и синхронизации их работы гораздо более высокие эффективность и надежность обеспечивают специализированные методы, общие для различных языков программирования и выходящие за рамки языка Си.

1.6. Взаимодействие модификаторов и спецификаторов хранения

В табл. 1.2 представлены правила совместного использования более одного модификатора и спецификатора хранения.

Таблица 1.2. Правила совместного использования модификаторов и спецификаторов хранения

Модификатор или спецификатор хранения	Место расположения данных	Совместимость с другими модификаторами и спецификаторами хранения	Типичное место размещения модификатора и спецификатора хранения
`unsigned`	Везде	Все базовые целочисленные типы без `signed` и любые спецификаторы хранения	Непосредственно перед именем базового типа
`signed`	Везде	Все базовые целочисленные типы без `unsigned` и любые спецификаторы хранения	Непосредственно перед именем базового типа

Продолжение табл. 1.2

Модификатор или спецификатор хранения	Место расположения данных	Совместимость с другими модификаторами и спецификаторами хранения	Типичное место размещения модификатора и спецификатора хранения
`const`	Везде	Любые типы и спецификаторы хранения, кроме `volatile`	Непосредственно перед знаковым модификатором, если он есть, иначе – перед именем типа
`auto`	Текущее положение указателя стека	Любые типы и спецификаторы хранения, кроме `register`, `static` и `extern`. Возможно использовать отдельно спецификатор `extern` в декларации существования глобальной переменной или константы, в другом месте текста программы определенной как `auto`	Перед первым из модификаторов, если он есть, иначе – перед именем типа
`extern`	В другом модуле	Любые типы. Совместим с `volatile`, несовместим с `auto` (см. примечание выше), `register` и `static` (антиподом которого является)	Самый первый в списке модификаторов
`register`	Регистр процессора (если компилятор сочтет допустимым)	Все базовые целочисленные типы, в некоторых реализациях также типы с плавающей точкой и другие небольшие объекты, помещающиеся в одном регистре, которые иначе оказались бы автоматическими. Несовместим с `auto`, `extern`, `static`, `volatile`	Перед первым из модификаторов, если он есть, иначе – перед именем типа

Окончание табл. 1.2

Модификатор или спецификатор хранения	Место расположения данных	Совместимость с другими модификаторами и спецификаторами хранения	Типичное место размещения модификатора и спецификатора хранения
`static`	В блоке статических объектов, основании стека или в куче в зависимости от реализации	Любые типы. Совместим с `volatile`, несовместим с `auto`, `register` и `extern` (антиподом которого является)	Перед первым из модификаторов, если он есть, иначе — перед именем типа
`volatile`	В куче, на стеке или в блоке статических объектов	Любые типы. Совместим с `auto`, `extern`, `static`, несовместим с `register`	Перед другими модификаторами, следом за `extern`, если он есть

1.7. Узкоспециализированные типы

В новом стандарте языка Си «законодательно закреплены» целочисленные типы, при помощи которых можно корректно оперировать размерами объектов и указателями. Причина их появления кроется в следующем: не гарантируется, что обычные, даже очень длинные, типы достаточны для хранения таких значений, которые могут возникнуть:

- при вычислении размера очень большого объекта;
- при определении разницы размеров объектов;
- при вычитании двух указателей.

Дело в том, что в современных вычислительных системах максимальный поддерживаемый размер адресного пространства столь огромен, что может не помещаться даже в такие типы, как `long long` и `unsigned long long`, отражающие предельные возможности стандартного арифметического блока процессора, но не его адресного блока. Поэтому введены три новых типа, назначение и свойства которых описаны в табл. 1.3.

К этим типам неприменимы модификаторы `long`, `signed` и `unsigned`. Возможность пользования новыми типами в некоторых системах разработки программного обеспечения встроенная, а в других необходимо включить соответствующий заголовочный файл:

```
#include <stddef.h>
```

Таблица 1.3. Свойства и назначение узкоспециализированных типов

Имя типа	Свойства и назначение	Пример

`size_t`	Является беззнаковым и способен вместить размер любого объекта (структуры, массива и т.д.), каким бы большим он ни был. Значение именно этого типа возвращает операция sizeof. Константа `SIZE_MAX` из заголовочного файла stdint.h содержит максимальное значение, принимаемое типом `size_t`	`size_t l=0;`
		`long double a[N];`
		…
		`l=sizeof(a);`

`ssize_t`	Знаковый компаньон `size_t`. Гарантированно может принять разность размеров любых объектов	`long double a[N], b[L];`
		`ssize_t k;`
		…
		`k = sizeof(b)-sizeof(a);`

`ptrdiff_t`	Является знаковым и гарантированно может сохранить разность произвольных указателей в адресном пространстве программы	`long double a[N], b[L];`
		`ptrdiff_t pd;`
		`pd=&b[L-1]-&a[0];`

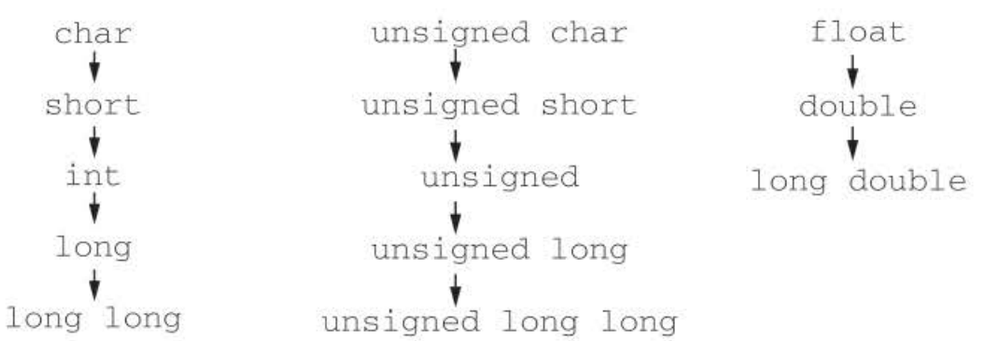
КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Зачем в Си введено несколько типов целочисленных данных и зачем несколько типов данных с плавающей точкой?
2. Как гарантировать знаковое (беззнаковое) представление `char` в программе?
3. Реализуйте быструю проверку принадлежности символа английскому алфавиту. Учтите наличие прописных и строчных букв.
4. При переходе от типа `float` к `double` и далее к `long double` относительная погрешность представления действительных чисел уменьшается быстрее, чем увеличивается возможность хранить огромные или крошечные значения. Что это дает?
5. Может ли переменная со спецификатором `static` иметь еще спецификатор `volatile`? Почему?
6. Может ли статическая переменная `(static)` быть внешней `(extern)`? Почему?
7. Покажите на примере, как происходит потеря точности в операциях с плавающей точкой при вычитании близких чисел.

ГЛАВА 2 ПЕРЕМЕННЫЕ И КОНСТАНТЫ

2.1. Стандартные преобразования типов

При строгой типизации всех переменных и констант в Си часто возникает потребность преобразовать данное одного типа в данное другого типа для корректного использования в выражении или обеспечения удобства хранения в памяти. Кроме того, тип данных диктует виды применимых операций (см. гл. 4). В Си имеется операция явного преобразования типа, однако ее повсеместное применение существенно снизило бы читаемость исходного текста программы. Поэтому предусмотрена допустимая последовательность *преобразований типов по умолчанию* (англ. implicit type conversion, coercion). Ее идея в Си очень проста: значение переменной или константы некоторого типа заведомо не испортится, если его разместить в переменной типа с бóльшими возможностями, заведомо способной сохранить ее значение без потерь. Такое преобразование называется *безопасным* (англ. promotion). Отсюда следуют цепочки преобразований, производимых компилятором по умолчанию и заведомо неспособных исказить исходное значение:



Важно понимать, что стандартные преобразования сохраняют *значение*, а не его двоичное представление, которое для различных типов может различаться.

Все прочие преобразования численных типов таят потенциальную опасность исказить исходное значение. Заведомо небезопасными (англ. unsafe), в частности, являются преобразования:

- ◊ достаточно длинного целого типа в тип с плавающей точкой, поскольку мантисса последнего может «не удержать» в себе все значащие цифры исходного целого, из-за чего потеряется точность;
- ◊ типа с плавающей точкой в целый – сразу по двум причинам: потеря дробной части и риск порчи значения вследствие преобразования большего поддерживаемого диапазона значений к меньшему;
- ◊ знакового целого в беззнаковое, если не гарантируется неотрицательность первого;
- ◊ беззнакового целого в знаковое, поскольку при той же разрядности беззнаковые типы в состоянии хранить числа, почти в 2 раза большие по модулю.

Поэтому ответственность за последствия таких преобразований программист должен брать на себя. Многими средствами разработки подобные ситуации по умолчанию не обнаруживаются, что требует от автора программы повышенного внимания при составлении выражений. В случаях наиболее рискованных преобразований компилятор может выдавать предупреждение или даже ошибку. Чтобы эти преобразования все-таки осуществить, от программиста требуется их явное оформление, что обсуждается в § 4.12. Стандартные преобразования типов указателей очень специфичны и поэтому рассматриваются отдельно в гл. 3, посвященной массивам и указателям.

2.2. Назначение переменных и констант

Свойства переменных и констант. Переменные (англ. variables) являются элементарными местами расположения изменяемых данных в памяти компьютера и обладают в языке Си следующими свойствами:

- в переменную можно неограниченное число раз записывать произвольные допустимые значения. При этом говорят, что переменной присвоено соответствующее значение. Запись первого значения называется инициализацией переменной; она может быть произведена прямо в месте введения переменной в программу;
- значение переменной можно читать тоже произвольное число раз. При этом всякий раз будет прочитано ее последнее записанное значение. Отсюда следует, что первому чтению обязательно должна предшествовать инициализация, иначе переменная будет содержать произвольное значение, которое случайно оказалось в памяти по данному адресу, т.е. так называемый «мусор» (англ. stray, garbage);
- записанное значение остается неизменным до тех пор, пока поверх него не будет записано новое или переменная не будет уничтожена. При записи нового значения старое безвозвратно утрачивается;
- всякая переменная имеет тип, предопределенный программистом с помощью описателя типа данных. В переменную нельзя записать значение несвойственного ей типа, не осуществив преобразование типа – явное или неявное. Типом переменной определяются ее размер в памяти компьютера, диапазон принимаемых значений и применимые к ней операции;
- описание типа переменной должно присутствовать в программе до первого использования переменной, лучше всего между началом программного блока, заключенного между фигурными скобками `'{'` и `'}'`, где вводится новая переменная, и первым исполнимым оператором этого блока (независимо от того, использует ли этот оператор объявленную переменную). Расположение описаний типов не в начале, а в произвольном месте блока, но все равно до первого использования переменной допустимо, хотя и не относится к хорошему стилю программирования, поскольку такие описания оказываются разбросанными по тексту блока и затрудняют понимание программы. Исключением из правила являются глобальные переменные, определенные вне какой-либо функции и блока. Их определения принято размещать в начале файла;

◊ уничтожение переменной происходит автоматически при достижении закрывающей фигурной скобки того программного блока, где была объявлена переменная (если она не имеет спецификатора хранения `static`).

Ниже показано, как определяются переменные.

Например:

```
{
double TotalWeight; /* определение переменной типа double
с именем TotalWeight */

int aBc_2000U=33; /* определение переменной типа int с
именем aBc_2000U и присвоение ей значения 33 */

char k, SMS=0; /* определение двух переменных типа char и
присвоение второй из них значения 0 */
}
```

Бóльшая часть сказанного выше относится и к константам, но со следующей оговоркой: значение присваивается константе ровно один раз, именно в момент ее определения. Неинициализированная константа обычно отмечается компилятором как ошибка. Значение константы, будучи скопированным в переменную, может независимо меняться в этой переменной, что никак не отражается на исходной константе.

Наименование переменных и констант. Всякая переменная и константа имеет свое уникальное в рамках программного блока имя, и по этому имени к ней обычно обращаются. Имя необходимо придумывать по тем же правилам, что и все имена в Си: оно должно быть уникальной последовательностью букв, цифр и символа подчеркивания '_', причем первый символ имени – всегда буква или подчеркивание. Существует и второй способ обращения к переменной и константе – с помощью указателя, детально рассмотренный в гл. 3.

По выбору имен имеется несколько проверенных временем рекомендаций; они отражают разные вкусы и могут несколько противоречить друг другу, а каким из них следовать, выбирает программист:

- для часто изменяемых переменных, а также тех, назначение которых неоднократно меняется по ходу работы программы, выбирают короткие имена, например однобуквенные. Таковы временные, вспомогательные переменные и переменные циклов (счетчики);
- переменные и константы, обозначающие похожие по смыслу и использованию объекты, логично называть похожим образом (но так, чтобы их не перепутать);
- редко используемые переменные и константы, а также те, смысл которых не сразу ясен читающему программу человеку, называют длинно, чтобы из названия можно было догадаться о назначении;
- начала разных слов и сокращения в длинном имени для удобства чтения принято выделять при помощи заглавных букв или предшествующего символа подчеркивания;
- не стоит применять имена, не содержащие строчных букв, так как имена из заглавных букв и символа подчеркивания обычно используют для названия констант препроцессора;
- скорость компиляции программы незначительно падает при использовании длинных имен, а на скорости исполнения программы это вообще не сказывается;
- некоторые старые компиляторы (большинство из них создано до 2000 г.) игнорируют в именах символы после 31-го. Поэтому они не могут отличить друг от друга имена, различающиеся только в следующих за 31-м символах, что приводит к ошибкам при компиляции или неверной работе программ;
- при описании переменной в комментарии указывают ее назначение; это поможет коллегам при чтении программы (а иногда и самому автору через пару лет после написания программы).

Декларации и определения переменных и констант. Практически во всех современных профессиональных программах немыслимо все необходимое разместить в одном файле, так как он получится необозримым. Поэтому неизбежно наличие части переменных, констант и функций в других файлах. Чтобы обращаться к ним из данного файла, нужно как-то объяснить компилятору, что эти удаленные объекты существуют,

иначе всякую попытку их использования компилятор будет воспринимать как ошибку. Действительно, работая с данным файлом, он не может «увидеть» другие модули программы и не знает, что представляют собой эти внешние объекты. Для этого используют так называемые *декларации* (англ. declarations). Декларация переменной или константы обязана в точности повторять ее определение, данное в другом файле, с двумя отличиями:

- самым первым словом декларации должен быть спецификатор хранения `extern`;
- в декларации недопустима инициализация.

Декларация сообщает компилятору и сборщику о точном виде внешнего по отношению к данному файлу объекта, что позволяет правильно собрать программу.

Очевидно, при обработке декларации не происходит создания такого объекта – лишь формируется на него ссылка. Поэтому вполне справедливо будет сказать, что, увидев декларацию, компилятор на ее основе не создает никакого исполнимого кода в отличие от определения, заставляющего его выделить под вновь создаваемый объект память, правильно его оформить и инициализировать по требованию программиста.

2.3. Числовые, символьные и строковые константы

Для инициализации переменных и в процессе работы программы часто используются константные значения, далее для краткости называемые константами[1], – объекты, значения которых записаны явно и изменению не подлежат. В языке Си константы могут быть одиночные (символьные, числовые и строковые) или так называемые *ограниченные константные выражения*, в которых требуемая константа вычисляется из нескольких одиночных констант по заданному

[1] Путаницы между понятиями «константа как неизменяемая переменная» и «константа как константное значение или выражение» не происходит, поскольку из контекста всегда ясно, о чем идет речь.

программистом правилу. Результат вычисления, очевидно, неизменно один и тот же. Таким образом, состоящее только из констант выражение тоже можно считать константой. Подобно переменным, константы обязательно имеют тип. Тип и форма представления константы определяются компилятором автоматически по ее значению и способу его записи, поскольку Си предоставляет несколько вариантов записи одного и того же константного значения:

- если константа заключена в одинарные кавычки, она символьная (одиночный символ);
- если константа заключена в двойные кавычки, она строковая (константная строка символов);
- если константа не заключена ни в одинарные, ни в двойные кавычки, она числовая.

Символьная константа – это всего лишь иная форма записи целой числовой константы, более удобная для работы с текстами, в которых с точки зрения хранения в памяти каждый символ – это просто целое число. При этом важно, что привязка числового значения к символу зависит от используемой кодировки языка и фиксированной не является. Несимвольная числовая константа допускает несколько форм записи:

- если она содержит десятичную точку или какие-либо другие признаки, указывающие на ее вещественный тип, она действительная, а иначе – целая;
- целая константа, начинающаяся с пары символов `"0x"` или `"0X"`, имеет шестнадцатеричную форму записи, подробно описанную ниже;
- целая константа, начинающаяся с нуля, за которым следуют другие восьмеричные цифры, имеет восьмеричную форму записи, также в деталях рассмотренную далее;
- в противном случае целая константа записана в десятичном виде.

Тип целочисленной константы определяется ее значением и возможным вспомогательным буквенным суффиксом (табл. 2.1). Согласно стандарту языка Си, для определения типа из соответствующей ячейки таблицы берется самый верхний тип, в который «помещается» числовое значение константы с учетом знаковости.

Таблица 2.1. Определение типа константы

Буквенный суффикс	Десятичная форма записи	Восьмеричная или шестнадцатеричная форма записи

Отсутствует	`int`
	`long`
	`long long`
		`int`
		`unsigned`
		`long`
		`unsigned long`
		`long long`
		`unsigned long long`

`u` или `U`	`unsigned`
	`unsigned long`
	`unsigned long long`	

`l` или `L`	`long`
	`long long`
		`long`
		`unsigned long`
		`long long`
		`unsigned long long`

{`u` или `U`} и {`l` или `L`}	`unsigned long` `unsigned long long`	
`ll` или `LL`	`long long`	`long long` `unsigned long long`
{`u` или `U`} и {`ll` или `LL`}	`unsigned long long`	

Общий смысл изложенных в табл. 2.1 правил таков: если значение константы может быть без потерь присвоено переменной некоторого типа из приведенных, имеющей минимально достаточный для этого размер, то тип константы — это тип такой переменной. Например, константа 23 имеет тип `int`, а константа 4000000000 — тип `long long` при условии, что тип `long long` позволяет хранить такое число, а знаковые типы `int` и `long` — нет. Это проверяется с помощью первых двух формул гл. 1 и выражения `sizeof`.

В табл. 2.2 приводятся формы записи целых, вещественных, символьных и строковых констант.

Таблица 2.2. Формы записи констант

Тип константы	Форма записи	Примечание
Целый беззнаковый	{десятичная_цифра}^{n}[<u или U>][<l или L или ll или LL>] — десятичная константа. Если константа ненулевая, первая десятичная_цифра не может быть нулем (иначе она будет считана как восьмеричная). Если требуется принудительно сделать константу типа long, сразу за ее последней цифрой ставится суффикс L или l (любой разделитель между ними недопустим). Аналогично поступают для типа long long (суффикс LL или ll). При наличии также суффиксов L, l, LL или ll суффикс U или u должен стоять первым 0{восьмеричная_цифра_от_0_до_7}^{n} – восьмеричная константа 0<x или X>{шестнадцатеричная_цифра_от_0_до_F}^{n} – шестнадцатеричная константа. Здесь n – число цифр в константе, а строчные или прописные латинские буквы кодируют цифры больше 9: A≡10, B≡11, C≡12, D≡13, E≡14, F≡15	Тип (длина) константы определяется ее числовым значением и использованными суффиксами
Целый знаковый	[-]{десятичная_цифра}^{n}[l или L или ll или LL] У такой константы может быть только десятичная форма записи. Если требуется принудительно сделать константу типа long, сразу за ее последней цифрой ставится латинская L или l. Аналогично поступают для типа long long (суффикс LL или ll). Любой разделитель между цифрами и буквами недопустим	
С плавающей точкой	[-][десятичная_цифра]^{m} [.][десятичная_цифра]^{f} [<e или E>[-]{десятичная_цифра}^{x}] [<f или F или l или L>] Здесь m, f, x – число цифр соответственно в целой, дробной части мантиссы и экспоненте. Целая и дробная части мантиссы и экспонента не могут отсутствовать одновременно. Более того, либо десятичная точка, либо буква e или E должны присутствовать, чтобы сообщать, что константа с плавающей точкой, а не целочисленная. Без суффикса константа получает тип double. Суффикс f или F принуждает сделать константу типа float, а суффикс l или L – типа long double. Любой разделитель между цифрами и буквой недопустим	

Продолжение табл. 2.2

Тип константы	Форма записи	Примечание
Символьный	Первая форма: 'символ', где символ может быть почти произвольным (зависит от реализации), кроме: а) апострофа ' ‘', б) обратной косой черты '\', в) перехода на новую строку '↵' Вторая форма: 'мнемокод', где мнемокод может принимать одно из следующих значений: \' – код символа апострофа \\ – код символа обратной косой черты слеша '\' \? – код символа вопросительного знака \" – код символа двойной кавычки \a – код, генерирующий звуковой сигнал (от англ. **a**udio, работает не во всех системах) \b – код символа забоя предшествующего ему символа (от англ. **b**ackspace) \f – код символа перехода в начало новой страницы (от англ. **f**orm **f**eed) \n – код символа перехода в начало новой строки (от англ. **n**ew line) \r – код символа возврата каретки (от англ. carriage **r**eturn) \t – код символа горизонтальной табуляции (от англ. **t**abulation) \v – код символа вертикальной табуляции (от англ. **v**ertical tabulation) \↵ – код игнорирования переноса строки на следующую строку текста программы для удобства написания Третья форма: '\восьмеричная_константа' или '\шестнадцатеричная_константа' Значение константы не должно превосходить возможности сохранения в символьном типе	В зависимости от формы константа содержит 1) символ между обрамляющими апострофами (первая форма), или 2) управляющий мнемокод после слеша между обрамляющими апострофами (вторая форма), или 3) целое число после слеша между обрамляющими апострофами (третья форма)

Окончание табл. 2.2

Тип константы	Форма записи	Примечание
Строковый	"{символьная_константа}^{n}" В результате получается одномерный массив – строка, состоящая из $n + 1$ последовательно идущего символа. Каждый символ можно адресовать отдельно точно так же, как элемент обычного массива. Последним элементом массива является автоматически добавляемый нулевой символ '\0', который служит признаком конца строки	

Таким образом, говоря, например, о шестнадцатеричной (гексадецимальной, англ. hexadecimal) константе, подразумевают не какой-то особый способ хранения или свойства константы, а просто использованный программистом при ее определении способ задания значения. Шестнадцатеричные константы позволяют при помощи всего одной цифры записать содержимое половины байта (полубайта, нибла, англ. nibble). Действительно, имеется 16 различных шестнадцатеричных цифр: 0...1 и A...F; в то же время 4 бита могут хранить тоже $2^4 = 16$ различных значений. Шестнадцатеричная запись констант очень удобна для записи масок и в адресной арифметике.

Например, для выделения второго справа байта из четырехбайтового числа можно записать:

```
x=x & 0x0000FF00;
```

или то же самое более компактно и эффективно (незначащие нули слева можно опускать и для шестнадцатеричных, и для восьмеричных констант):

```
x&=0xFF00;
```

Восьмеричные (октальные, англ. octal) константы используются реже и удобны для работы с тройками последовательно идущих битов. В Си отсутствуют константы в двоичной записи, так как она порождает очень длинные последовательности нулей и единиц, плохо читаемые человеком. С одной стороны, шестнадцатеричная и восьмеричная формы записи констант отражают двоичное представление данных в компьютере. С другой стороны, они обеспечивают компактную и хорошо понимаемую (после недолгого привыкания) запись в отличие от громоздкой двоичной записи.

Пример, приведенный ниже, иллюстрирует использование суффиксов, принудительно уточняющих тип констант:

```
#include <stdio.h>

#define A1 1234567890u
#define A2 1234567890U
#define A3 12345678901
#define A4 1234567890L
#define A5 123456789011
#define A6 1234567890LL
#define A7 1234567890ul
#define A8 1234567890UL
#define A9 1234567890ull
#define AA 1234567890ULL

#define B1 1.234567890f
#define B2 1.234567890F
#define B3 1.2345678901
#define B4 1.234567890L

int main()
{
unsigned a1=1234567890u;
unsigned a2=1234567890U;
long a3=12345678901;
long a4=1234567890L;
long long a5=123456789011;
long long a6=1234567890LL;
unsigned long a7=1234567890ul;
unsigned long a8=1234567890UL;
unsigned long long a9=1234567890ull;
unsigned long long aa=1234567890ULL;

float b1=1.234567890f;
float b2=1.234567890F;
long double b3=1.2345678901;
long double b4=1.234567890L;

printf("\n  A1=%u; A2=%u; A3=%ld; A4=%ld; A5=%lld;\
 A6=%lld; A7=%lu; A8=%lu; A9=%llu; AA=%llu",
  A1,A2,A3,A4,A5,A6,A7,A8,A9,AA);

printf("\n  a1=%u; a2=%u; a3=%ld; a4=%ld; a5=%lld;\
```

```
 a6=%lld; a7=%lu; a8=%lu; a9=%llu; aa=%llu",
  a1,a2,a3,a4,a5,a6,a7,a8,a9,aa);

printf("\n  b1=%g; b2=%g; b3=%Lg; b4=%Lg",b1,b2,b3,b4);

printf("\n  B1=%g; B2=%g; B3=%Lg; B4=%Lg\n",
  B1,B2,B3,B4);

return 0;
}
```

Символьные константы (англ. character constants) предназначены главным образом для операций с отдельными символами в строке. Их использование позволяет сделать код переносимым, т.е. работоспособным безотносительно к тому, какая кодовая страница (кодировка текста) используется. Действительно, в настоящее время применяется не менее четырех кодировок русских букв (UTF = Unicode, KOI8R = Unix, codepage 1251 = Windows, codepage 866 = DOS) и заранее не всегда ясно, какая будет использована в программе. Приведенный ниже фрагмент программы использует тот факт, что во всякой кодовой таблице коды букв упорядочены по алфавиту[1]. Он проверяет, является ли символ `letter` маленькой кириллической буквой; если является, то делает эту букву заглавной:

```
if (letter >= 'а' && letter <= 'я')
  letter += 'A' - 'a';
```

2.4. Константы и инициализация

Из констант и переменных можно составлять арифметические, логические и комбинированные выражения и присваивать их значения переменным (соблюдая правила преобразования типов). Если выражение состоит целиком из констант, оно называется константным и рассматривается как единая константа, значение которой вычисляется, как правило, еще на этапе компиляции.

---

[1] Имеются немногие исторически возникшие исключения. Например, код буквы 'ё' в некоторых кодировках не находится между кодами букв 'е' и 'ж'.

Запишем операцию присваивания:

```
имя_переменной_или_константы=
константа_или_константное_выражение
```

Непосредственно перед знаком равенства и после него можно ввести произвольное число пробелов для улучшения читаемости.

Сходным образом осуществляется инициализация переменных – присваивание начальных значений в месте введения в программу в определении.

```
описатель_типа_данных {имя_переменной [=
инициализирующая_константа_или_константное_выражение]}
[, имя_переменной [=
инициализирующая_константа_или_константное_выражение]]n;
```

При этом `описатель_типа_данных` может включать в себя рассмотренные прежде модификаторы и спецификаторы хранения.

Инициализацию в месте определения переменной рекомендуется проводить всегда, когда только возможно, даже если где-то потом планируется изменить значение переменной еще до первого ее чтения. Инициализация в момент определения обычно лишь незначительно замедляет программу, зато гарантирует, что в вычислительный процесс не вступят переменные со случайными значениями, непредсказуемым образом влияющие на поведение и результаты выполнения программы. Ошибки, связанные с неинициализированными переменными, чрезвычайно сложно отыскивать, а они, к сожалению, весьма распространены.

В языке Си имеется пришедший из C++ способ записать константу более изящно, чем это делалось выше. Ее можно не только записать в виде цифр, символов и мнемокода, но и дать ей имя как переменной с той лишь разницей, что впоследствии значение переменной можно поменять, а значение константы – нет. Для этого используется конструкция с рассмотренным выше модификатором `const`:

```
const описатель_типа_данных {имя_константы =
```

```
инициализирующая_константа_или_константное_выражение}
[,имя_константы =
инициализирующая_константа_или_константное_выражение]n;
```

Стоит еще раз подчеркнуть, что забыть инициализировать константу невозможно: об этом напомнит компилятор сообщением об ошибке. Другим, более старым методом сопоставления константе имени является использование препроцессора (см. гл. 10).

2.5. Понятие блока программы

Исполнимый код, заключенный между парой соответствующих друг другу фигурных скобок, называется *блоком* программы (англ. block of code). Хотя фигурные скобки используются также в определениях и декларациях структур, смесей, битовых полей, элементов перечислимого типа и при инициализации массивов и структур, все эти конструкции лишь описывают данные и не содержат ни одного исполнимого оператора, поэтому не являются блоками.

В частности, блоком является *тело* всякой функции. Но и внутри тела функции может быть множество блоков. Блоки можно (а часто и нужно) вкладывать друг в друга, чтобы таким образом наглядно отразить иерархическое устройство программы.

Например:

```
{int i=0, j=-10;
…
for (i=0; i>j; i--)
    {
    double a=2.4;
    …
    a+=i;
    …
    }
…
}
```

Содержимое внутренних вложенных блоков у программистов принято для наглядности сдвигать на одинаковое число позиций вправо по отношению к содержимому объемлющего блока. Это позволяет четко видеть структуру программы. Важнейшими частными случаями блока являются тело функции, составной оператор (см. одноименный параграф в гл. 6) и тело оператора `switch`.

2.6. Область определения и область видимости

Внутри любого блока доступны все переменные и константы объемлющих блоков. Каждый блок обладает очень полезным свойством: в его начале (перед первым исполнимым оператором) можно размещать определения «своих собственных» переменных и констант, доступ к которым невозможен извне блока[1]. Такие объекты называются *локальными* для данного блока. Их область видимости – данный блок и все вложенные в него блоки.

Если локальные объекты определены в блоке как автоматические, они будут создаваться заново всякий раз, когда программа входит в данный блок. При выходе из него они разрушаются, поэтому их никак (даже с помощью указателя) нельзя использовать вне блока. На создание и уничтожение таких объектов требуется некоторое число действий, автоматически добавляемых компилятором в исполнимый код. Однако такая постоянная работа с памятью может заметно тормозить программу, если блок является телом цикла или вмонтирован в тело объемлющего цикла. Следует рассмотреть возможность превращения внутренних переменных во внешние, т.е. перенесения их в один из объемлющих блоков.

Локальные объекты, определенные как статические, порождаются и инициализируются единожды – в момент первого попадания в блок и сохраняются до завершения программы. Между соседними обращениями к блоку их значения остаются

[1] Строго говоря, более или менее корректный доступ извне возможен только к внутренним статическим объектам блока при помощи указателя. Хорошим и безопасным стилем программирования такие трюки, однако, порицаются, поскольку их ничем нельзя оправдать.

неизменными (если их не передавать по указателям и менять вовне). Эффективность такой реализации выше из-за отсутствия повторяемых операций с памятью, но содержащая этот блок функция перестает быть реентерабельной, т.е. пригодной для независимых друг от друга параллельных вызовов. Поэтому стоит очень тщательно подумать, прежде чем размещать спецификатор хранения `static` внутри блока.

В виде блока обычно оформляют функционально законченную часть программы, в связи с чем временные переменные и константы, используемые только в этой части, нужно из соображений удобочитаемости программы стремиться сделать локальными. Если обращения к блоку происходят не чрезмерно часто, организация даже множества местных временных переменных оправданна, так как улучшает понятность программы, изолирует данные и бережет от ошибок.

2.7. Маскирование переменных и констант

Предположим, одноименные переменные или константы имеются в данном и внешнем блоках, как в таком полном примере:

```
#include <stdio.h>

long i=9; /* в функции main экранирована местными переменными */

int main()
{
int i=100;
double a=11.11111;
printf("\n Внутри блока функции main: i=%d; a=%g",i,a);
  {
  long double i=-1000.0;
  printf("\n Внутри самого вложенного блока: i=%Lg;\
  a=%g",i,a);
  }
printf("\n Снова внутри блока функции main: i=%d;\
  a=%g",i,a);
}
```

Очевидно, переменная `a` доступна во вложенном блоке, а переменная `i` из внутреннего блока не существует во внешнем блоке. Однако какая переменная во внутреннем блоке понимается под переменной `i` – внутренняя или внешняя (они имеют даже разный тип)? Ответ на этот вопрос прост (в чем рекомендуется убедиться, откомпилировав и запустив эту программу): внутренняя переменная всегда «заслоняет собой», как говорят, экранирует все одноименные ей внешние переменные, которые во внутреннем блоке становятся невидимыми. Тем не менее они продолжают существовать, значения их сохраняются и при возврате во внешний блок они снова доступны. В случае крайней необходимости с одноименной переменной из окружающего блока можно работать, передав во вложенный блок на нее указатель, но этот трюк не рекомендуется, поскольку ведет к плохо структурированному тексту программы, в котором легко допустить ошибку и не заметить ее.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. В каких случаях требуется явное преобразование типа?
2. Объясните результат попытки такого присваивания:
   ```
   unsigned s=-1;
   ```
3. Удастся ли посредством «хитрого» приведения типа изменить содержимое константы:
   ```
   const int a=6; int *pi; pi=(int*)&a; *pi=7;
   ```
   Почему?
4. Имеются внешняя глобальная переменная `S` и блок `B`, в теле которого определена локальная переменная *S*. Как внутри `B` просто осуществить одновременный доступ к обеим переменным?
5. Попытайтесь средствами Си реализовать механизм, при помощи которого в многопоточной среде (все потоки одной программы используют общую память и могут одновременно выполняться на различных ядрах процессора) можно гарантированно выяснить, свободен или занят в данный момент времени объект (например, массив), разделяемый (используемый по очереди) несколькими потоками. Используйте спецификатор `volatile`. Сколько глобальных переменных необходимо и достаточно для решения задачи?

ГЛАВА 3 МАССИВЫ И УКАЗАТЕЛИ

3.1. Одномерные массивы

Обсуждавшиеся выше встроенные типы данных представляют собой одиночные числа или символы, т.е. *скалярные величины*. Полезно иметь средство объединения однородных скалярных величин в группы для совместного хранения и обработки. Такая группа называется *массивом* (англ. array). При создании массива требуется задать число элементов, находящихся в массиве, и их организацию в плане последовательности хранения и доступа.

Вектор (англ. vector) – одномерный массив, простейший случай объединения множества однотипных скалярных объектов. Чтобы выделить память под необходимое число элементов заданного типа и дать массиву имя, используют определение.

Например:

```
int a[10];
```

В этом примере начиная с места такого определения в программу введен массив, состоящий из 10 элементов типа `int` и имеющий общее имя `a`. Натуральное (т.е. целое положительное) число в квадратных скобках задает число элементов, которое называют *размером* (англ. dimension) массива. Всякий элемент массива можно адресовать, указав его порядковый номер в массиве, называемый *индексом* (англ. index), в квадратных скобках.

Например:

```
a[0]=12; a[1]=-100; x=2*a[0]-a[1];
```

Как видно из приведенного примера, в языке Си индексы массивов начинаются с нуля. Это нужно запомнить, поскольку

имеется немало языков программирования (например, Fortran, Basic, Octave), где нумерация элементов стартует с единицы. Не менее важно избегать обращения к несуществующим элементам массива. Так, было бы грубой ошибкой в нашем примере обращаться за пределы массива:

```
a[-1]=4; x=a[10];
```

Первый элемент имеет нулевой индекс, а последний — девятый, т.е. на единицу меньше размера массива, так как индексация начинается с нуля. Попытки обращаться к отсутствующим элементам массива, как правило, приводят к негативным последствиям: на этих местах в памяти находятся другие данные, не имеющие отношения к массиву. Данные можно повредить при попытке записать в массив, а при попытке чтения результат непредсказуем, поскольку читаются другие данные или «мусор». При этом обращение за пределы массива часто не пресекается компилятором. Поскольку индексы элементов могут вычисляться достаточно сложно, вставка в программу кода, проверяющего корректность всякого обращения, привела бы к сильному замедлению работы. Более того, в некоторых сложных случаях такие проверки вообще беспомощны.

3.2. Многомерные массивы

У вектора, как известно из математики, одно измерение, а у матрицы их два. Язык Си вообще не имеет ограничений в отношении значения размерности массива. Общее число измерений — размерность (англ. dimensionality) и число элементов вдоль каждого измерения (англ. dimension) указываются при определении массива.

Например, определение
```
long double c[2][10][100];
```
создает массив размерности 3.

Индексация размерностей в Си устроена *справа налево*, и младший индекс является самым правым, а старший — са-

мым левым. Это аналогично расположению разрядов числа: наиболее значащий — слева, наименее значащий — справа. Все элементы массива гарантированно размещаются в памяти последовательно, в нашем примере от `c[0][0][0]` до `c[1][9][99]`. При смещении в памяти от первого элемента к последнему индексы изменяются следующим образом: самый правый меняется быстрее всего, медленнее сменяются значения среднего индекса и совсем медленно увеличиваются значения крайнего левого индекса. Это легко запомнить по аналогии с табло любого цифрового счетчика. Выделяя память для многомерных массивов, всегда следует задавать себе вопрос, сколько места для него потребуется. Для этого нужно перемножить все размеры массива вдоль каждой из его размерностей и результат умножить на размер, определяемый выражением одного элемента `sizeof(имя_типа_элемента_массива)`. В данном примере объем памяти в байтах составит `2*10*100*sizeof(long double)`. Вследствие перемножения размеров общее место для хранения массива растет очень быстро с увеличением размера вдоль каждой размерности. В общем случае описание массива без инициализации его элементов имеет вид:

```
[спецификаторы_хранения_всего_массива]
[модификаторы_хранения_элемента_массива]
имя_типа_элемента_массива
имя_массива {[размер_вдоль_размерности]}^n;
```

Не всякий спецификатор хранения может быть использован совместно с массивом; так, очевидно, массив нельзя разместить в регистре. Возможность указать остальные спецификаторы зависит от конкретной системы. Имя типа элемента может включать в себя произвольные допустимые модификаторы (к введенным программистом новым типам приемлем только модификатор `const`). Имя массива подчиняется общим требованиям к имени в Си. Согласно действующему стандарту, параметр `размер_вдоль_размерности` не обязан быть константой или константным выражением. Допустимы переменные и выражения с переменными. Главное, чтобы значение размерности получилось осмислен-

ным, целым и положительным, в частности требуется, чтобы все участвующие в его вычислении объекты были правильно инициализированы.

3.3. Инициализация массива

Массив можно объявить константным. В этом случае его элементы доступны только на чтение.

Например:

```
x*=a[3]-a[4]/y; /* синтаксически допустимо */
a[5]+=x; /* приведёт к сообщению об ошибке */
```

Приведенное ниже «наивное» определение константного массива

```
const float b[7];
```

вне всяких сомнений, вызовет сообщение об ошибке. Ее причина вполне очевидна: программист определяет массив, элементы которого нельзя в дальнейшем менять, забыв задать им значения. Список значений для инициализации приводится в том месте, где массив определяется, т.е. под него отводится память. Значения перечисляются через запятую, и их совокупность заключена в фигурные скобки.

Например:

```
const float b[7]={-0.08, -0.2, 0.14, 55.8, 2.7e4,
-1.9e-12, 9.73};
```

Именно в такой последовательности они будут присвоены элементам по возрастанию индекса с нуля. Всякое значение в *списке инициализации* должно быть константой или константным выражением. Сам массив может быть константным или нет, что определяется наличием модификатора `const` перед его именем. В новых реализациях допускаются выражения с участием переменных в качестве инициализирующего значения. Необходимо лишь, чтобы к данному моменту значения

всех входящих в такие выражения переменных были правильно заданы.

Аналогично можно действовать для задания значений элементам массива большей размерности, объединяя группы значений, относящихся к одной размерности, в дополнительные фигурные скобки. Самые внешние скобки соответствуют крайней левой размерности, а скобки наибольшей глубины вложенности – крайней правой.

Например:

```
char syms[2][3][2]={{{'a', 'A'}, {' ', '\t'},
  {'z', 'Z'}},{{'ы', 'Ы'}, {'\0', '\b'}, {'я', 'Я'}}};
```

Размещение значений при этом таково:

элемент массива	значение	элемент массива	значение
syms[0][0][0]	'a'	syms[0][0][1]	'A'
syms[0][1][0]	' '	syms[0][1][1]	'\t'
syms[0][2][0]	'z'	syms[0][2][1]	'Z'
syms[1][0][0]	'ы'	syms[1][0][1]	'Ы'
syms[1][1][0]	'\0'	syms[1][1][1]	'\b'
syms[1][2][0]	'я'	syms[1][2][1]	'Я'

Такой способ инициализации называется *структурным*, поскольку заключенные в фигурные скобки группы присваиваемых значений отражают структуру массива. Легко убедиться, что размещение значений в списке инициализации в точности соответствует порядку расположения элементов массива в оперативной памяти. Этим фактом можно воспользоваться, перечислив инициализирующие выражения в той же последовательности, но без *вложенных* фигурных скобок.

Например:

```
char syms[2][3][2]={'a', 'A', ' ', '\t',
  'z', 'Z', 'ы', 'Ы', '\0', '\b', 'я', 'Я'};
```

Подобная инициализация носит название *неструктурной*, поскольку следует логике размещения элементов в памяти,

а не конструкции массива. Ее не рекомендуется использовать для массивов размерности более единицы, так как проще потерять место привязки присваиваемого значения к элементу и вставить лишний инициализатор или пропустить необходимый. Список инициализации «съедет», и все последующие элементы попадут не на свои места. Возможно, компилятор этого не заметит. Дело в том, что список инициализации может быть меньше суммарного количества элементов в массиве. Так сделано для возможности *частичной инициализации* массива, позволяющей присвоить значения некоторому числу *первых* элементов, затем пропустить остальные.

Например:

```
float b[7]={-0.08, -0.2, 0.14, 55.8, 2.7e4};
```

явно инициализирует только первые пять элементов массива `b`. Согласно стандарту языка Си, при использовании частичной инициализации оставшиеся элементы инициализируются компилятором неявно путем записи в них нулей, приведенных к типу элементов массива[1]. Интересные возможности открывает *частичная структурная* инициализация; так, можно задать некоторое число элементов в начале, затем пропустить несколько элементов в середине массива и инициализировать последующие. При неструктурной инициализации такое невозможно из-за того, что отсутствует механизм привязки значений к элементам. Структурная инициализация позволяет это сделать, если пропущенные инициализирующие выражения размещаются в конце блока, ограниченного фигурными скобками. Именно фигурные скобки помогают компилятору сориентироваться, что куда записывать.

Например, в результате:

```
char syms[2][3][2]={{{'a', 'A'}, {' '},
  {'z', 'Z'}},{{'ы', 'Ы'}, { }, {'я', 'Я'}}};
```

[1] Однако в некоторых системах разработки автоматическое заполнение нулями не происходит, и после частичной инициализации в пропущенных элементах массива находится «мусор». Поэтому здесь не будет лишней программистская осторожность и проверка.

пропущенные `syms[0][1][1]`, `syms[1][1][0]` и `syms[1][1][1]` будут содержать символы с нулевым кодом. Если же инициализирующих элементов задано больше, чем предусмотрено размером массива, в подавляющем большинстве средств разработки выдается ошибка компиляции. Во многих системах программирования разрешается опускать значения размера массива вдоль его самой старшей (левой) размерности, если инициализация четко задает это количество. Так, в результате определения массива той же самой организации, что и в предыдущем примере

```
char syms_complete_initialization[][3][2]=
  {{{'a', 'A'}, {'b', 'B'}, {'c', 'C'}},
   {{'d', 'D'}, {'e', 'E'}, {'f', 'F'}}};
```

под все элементы трехмерного массива выделяется место и производится их явная инициализация.

3.4. Строковые массивы и константы

Особое место в Си отводится символьным (строковым, литеральным, англ. character) массивам, которые служат носителем фрагментов текста. Действительно, объединение символов в вектор приспособлено для этого как нельзя лучше. Выделение памяти под строковый массив фиксированной длины выполняется, как обычно:

```
char phrase[100];
```

Этот массив можно инициализировать частично.

Например:

```
char phrase[100]="Это пример строковой константы";
```

Данный способ инициализации присущ только символьным строкам. Так, место выделено под 100 символов, а осмысленные значения получили лишь первые 31. Выражение, стоящее справа от знака присваивания, является строковой кон-

стантой. Текст строковой константы всегда заключается в пару двойных кавычек в отличие от одиночного символа, помещаемого в пару одиночных кавычек. Подсчет символов к строковой константе из примера дает 30. Еще один символ при инициализации строковой константой в массив добавляется на ее конце. Этот символ имеет код 0, обозначается `'\0'` и служит маркером конца текста. Все идущее вслед за ним считается «мусором». Этот символ можно вставить вручную.

Например:

```
phrase[10]= '\0';
```

Попытка распечатать такую строку:

```
printf("%s", phrase);
```

даст вывод:

```
Это пример
```

поскольку `printf`, как практически все функции стандартной библиотеки Си, работающие со строками, понимают нулевой символ как конец текста и его не печатают, как и все последующие символы. Если программист намерен инициализировать весь символьный массив и выделить под него столько символов, сколько занимает инициализирующая строковая константа (плюс концевой нулевой символ), язык Си его освобождает от необходимости считать символы. Достаточно лишь «намекнуть» компилятору, что это строковый вектор при помощи пары пустых квадратных скобок, а остальное тот сделает сам.

Например:

```
char phrase[]="Это пример строковой константы";
```

Под строку здесь выделен 31 элемент типа `char`. Если при описании строковая константа не помещается в одной строке в том смысле, что она заметно выходит за рамки правой границы экрана текстового редактора при выравнивании текста по ле-

вой границе, имеется способ перенести ее на новую строку так, чтобы этот перенос не попал внутрь самой константы и не нарушил ее структуру. Для этого используется знак '\' (называемый обратной, или падающей, косой чертой, обратным слешем, англ. slash), означающий, что продолжение строковой константы находится в начальной позиции следующей строки.

Например:

```
char str[]="Это пример очень-очень длинной строки, \
которая вполне может не помещаться вся сразу в поле\
 зрения текстового редактора программиста и поэтому \
её приходится переносить и, возможно, неоднократно";
```

При этом все, что находится в строках определения такой константы после символа '\', будет проигнорировано (отнесено к комментариям).

3.5. Сущность и простейшие применения указателя

Зная имя некоторой переменной, при помощи одноместной операции & можно получить указатель на нее.

```
тип имя_переменной;
…
&имя_переменной
```

Это открывает возможность присвоить значение указателя другой переменной подходящего для этого типа и адресовать исходную переменную уже при помощи указателя:

```
тип *имя_переменной_указателя;
…
имя_переменной_указателя=&имя_переменной;
…
*имя_переменной_указателя=новое_значение;
```

В результате новое_значение присваивалось *по указателю*, но попало в переменную имя_переменной так, как будто

было загружено в нее непосредственно. Эти действия полезно представлять себе следующим образом: переменная-указатель имеет специальный тип, пригодный только для хранения адреса переменной исходного типа имя_переменной. На момент своего определения она ни к какому объекту не привязана. В ней находится вместо адреса случайный «мусор». Оператор присваивания

```
имя_переменной_указателя=&имя_переменной;
```

получает и загружает в нее начальный адрес переменной имя_переменной. Одноместная операция «звездочка», наоборот, дает доступ к переменной, расположенной по адресу, к которому она применяется. Очевидно, чтобы это значение правильно записать или прочесть, указатель должен «знать», на объект какого рода он указывает. Так, переменные типов int и float могут занимать в памяти компьютера одинаковое место, но их содержимое следует интерпретировать различно. Говорят, что операция «звездочка» *разадресовывает* (разыменовывает) указатель, к которому применяется. Конструкция

```
*имя_переменной_указателя
```

выступает в качестве альтернативного имени переменной и может находиться по обе стороны знака равенства, т.е. быть левым и правым значением. Очень важно осознавать, что само по себе определение указателя:

- ◊ не инициализирует его (т.е. в нем находится случайный «мусор»);
- ◊ не выделяет память под объект, для указания на который он предназначен. Память выделяется для хранения только самого указателя.

Проиллюстрируем приведенные выше абстрактные правила операций с указателем.

Например:

```
#include <stdio.h>

int main()
```

```
{
double a=17.5, *p3=0;
double *p1=&a, b=-100.2, *p2=0;
      /* p1, p2 и p3 - указатели на тип float, из них пока
      только p1 указывает на конкретный объект - перемен-
      ную a */
p2=&b; /* теперь p2 указывает на переменную b */
printf("\n a=%f, *p1=%f",a,*p1); /* к переменной a можно
      обратиться двумя способами */
printf("\n b=%f, *p2=%f",b,*p2); /* и к переменной b мож-
      но обратиться двумя способами */
p3=p1; p1=p2; p2=p3; /* значения однотипных указателей
      можно присваивать друг другу: при помощи вспомога-
      тельной переменной-указателя p3 поменяли местами
      содержимое p1 и p2 */
printf("\n после обмена указателями:\n a=%f,\
 *p2=%f",a,*p2);
printf("\n b=%f, *p1=%f",b,*p1);
a=28.28; *p1=-3800.0038; /* значение переменной b поменя-
      ли посредством указателя p1 на нее */
printf("\n изменено значение a напрямую: a=%f,\
 *p2=%f",a,*p2);
printf("\n изменено значение b косвенно,\
 по указателю p1: b=%f, *p1=%f",b,*p1);
return 0;
}
```

Начинающие программисты очень часто путают операции `&` и `*`. Шуточное мнемоническое правило позволяет запомнить их использование везде, кроме определений и деклараций: «**з**вездочка дает **з**начение, **а**мперсанд дает **а**дрес». В определениях и декларациях применяется только `*`, указывающая, что правее нее стоит переменная или константа со смыслом указателя.

3.6. Инициализация указателя

Использование неинициализированного указателя — одно из наиболее разрушительных действий (или бездействий?!) программиста. Как и всякая неинициализи-

рованная переменная (за исключением статических и глобальных, при отсутствии явной инициализации неявно инициализируемых нулем соответствующего типа), указатель исходно содержит «мусор». При попытке обратиться по такому «шальному» указателю случайная последовательность битов будет воспринята как правильный адрес и по этому адресу произойдет попытка что-то прочитать или записать. Данная область памяти может содержать другую информацию или вообще не принадлежать обратившейся к ней программе. Попытка прочитать скорее всего останется некоторое время незамеченной, но программа далее будет работать неправильно, попытка записать наиболее вероятно будет обнаружена средствами «самозащиты» операционной системы и вызовет аварийное завершение программы с сообщением типа: «Программа выполнила недопустимую операцию и будет завершена».

Отсюда следует вывод: всякий указатель перед первой разадресацией (т.е. перед первым обращением к памяти с его помощью) должен обязательно быть инициализирован:

```
char cx[]="Собака и волк - родственники";
char *px=&cx[16];
```

Это лучше всего сделать непосредственно в месте его определения, даже если его значение позже планируется изменить (например, с помощью функции динамического выделения памяти, англ. dynamic memory allocation). Переменному или константному указателю можно присвоить только значения следующих видов:

- ◊ другой правильный указатель того же типа или выражение, дающее корректный указатель, который должен адресовать реально существующий объект или массив объектов того же типа;
- ◊ явно преобразованный к данному типу указатель на `void`, адресующий кусок фактически выделенной памяти, например полученный от функции динамического выделения памяти,
- ◊ целочисленный нуль.

Хотя обычно считается, что указатель – длинное беззнаковое число, недопустимо присваивать ему какое-либо числовое значение, кроме нуля, даже если откуда-то известно, что это правильный адрес. Причина в том, что нет никаких гарантий, что правильность этого адреса сохранится и в будущем, и то, на что он указывает, не будет перемещено компилятором на другое место.

Если инициализирующее константное или неконстантное выражение еще не может быть вычислено к моменту определения указателя, хороший стиль программирования предписывает сначала присвоить указателю нулевое значение.

Например:

```
unsigned short N=0;
long double *a=0;
…
N=…; /* выражение, дающее целое положительное число */
a=(long double*)malloc(N*sizeof(long double));
…
```

В Си нулевое значение указателя обозначает его недопустимость, а при попытке с его помощью обратиться включается стандартный обработчик, генерирующий сообщение о невозможности операций над нулевым указателем, что диагностируется намного проще, чем описанные выше скрытые проблемы. Правила языка таковы, что нуль автоматически преобразуется к типу произвольного указателя.

3.7. Указатели и массивы. Адресная арифметика

Понятия указателя и массива связаны теснейшим образом. Например, имя одномерного массива – это всегда указатель на его начальный элемент. Для вектора-массива способы записи его начального адреса

`&имя_массива[0]` и просто `имя_массива` эквивалентны. Посредством указателя можно обратиться к произвольному элементу массива.

Например, определим массив `g` и переменную-указатель `pointer_to_unsigned_long_long` подходящего типа:

```
unsigned long long g[10],
  *pointer_to_unsigned_long_long;
```

Сделаем так, чтобы она указывала на начало массива:

```
pointer_to_unsigned_long_long=&g[0];
```

Привязка типа указателя к типу объекта, на который он указывает, нужна не только для правильного обращения с данными, но также и для так называемой *адресной арифметики*. Кроме уже рассмотренной разадресации `*`, к указателю применимы операции, данные в табл. 3.1.

Таблица 3.1. Операции, применимые к указателю

Операция	Допустимый второй операнд	Значение результата	Примечание
`+` `+=`	Целое число	Указатель, смещенный на столько объектов, на которые он указывает, сколько задает второе слагаемое. Очень важно, что целое число задает не количество байтов, а количество объектов	Исходный указатель называют *базовым адресом* (или сокращенно базой), так как он задает исходный адрес, а второй операнд именуется *смещением*, так как он определяет, на сколько размеров объекта нужно сдвинуться от базового адреса. Если смещение положительно, результирующий адрес больше базового, а если оно отрицательно, происходит смещение в сторону меньших адресов
`-` `-=`	Целое число	Указатель, смещенный на столько объектов, на которые он указывает, сколько задает второе слагаемое, но в обратную сторону, чем при сложении	Обычно используется, если дан базовый указатель на конец объекта, а не на его начало

Продолжение табл. 3.1

Операция	Допустимый второй операнд	Значение результата	Примечание
`-` (двухместная операция)	Другой указатель того же типа	Целое число, равное количеству объектов данного типа, заключенных между двумя указателями (а не числу байтов!)	Основные применения – выяснение размера массива или выявление наличия выравнивания (см. § 5.7)
`++указатель`	Нет	Указатель на следующий объект того же самого типа (а не на следующий байт!)	Как и для обычных переменных, эквивалентно: `указатель=указатель+1`
`указатель++`	Нет	Указатель на тот же объект, а сам указатель адресует следующий объект того же типа	То же
`--указатель`	Нет	Указатель на предыдущий объект того же самого типа (а не на предыдущий байт!)	Как и для обычных переменных эквивалентно: `указатель=указатель-1`
`указатель--`	Нет	Указатель на тот же объект, а сам указатель адресует уже *предыдущий* объект того же самого типа	То же
Все операции сравнения: `<` `<=` `>` `>=` `==` `!=`	Другой указатель	Логическое значение, зависящее от результата сравнения	Используется для проверки (не)совпадения указателей или выяснения взаимного расположения объектов в памяти. Наиболее часто это проверка, не один ли и тот же объект адресуют два указателя

Окончание табл. 3.1

Операция	Допустимый второй операнд	Значение результата	Примечание
Операции проверки на равенство `==` и неравенство `!=`	Целочисленный нуль	Проверка указателя на предмет допустимости его значения	В Си имеется всеобщая договоренность, что нуль является обозначением негодного значения указателя. Исторически это связано с тем, что почти нет компьютеров, у которых по нулевому адресу находилось бы что-нибудь, доступное программисту. Произвольный ненулевой указатель трактуется как допустимый, и по нему может быть произведена запись или выборка информации. Поэтому важно не забывать инициализировать указатели. «Мусор», оставшийся в выделенной под указатель переменной, влечет операции чтения и записи с привлечением случайного адреса и непредсказуемыми последствиями, своего рода игра в «русскую рулетку по-программистски»

При использовании адресной арифметики изменение указателя всегда «само собой» происходит в единицах того типа, на который этот указатель указывает. Именно для этого компилятору требуется знать тип указателя. Компилятор сам «разбирается», на сколько байтов ему нужно сместиться, программист от этих забот освобожден (если только не прибегает к специальным потенциально опасным трюкам).

Вернемся к примеру с указателем на начало массива и рассмотрим один из самых распространенных случаев использования адресной арифметики. Добавление к переменной `pointer_to_unsigned_long_long` неотрицательного целого числа дает указатель, смещенный от базового указателя на равное этому числу количество объектов вправо.

Например, в результате приращения

```
pointer_to_unsigned_long_long+=2;
```

`pointer_to_unsigned_long_long` принимает значение, равное `&g[2]`. Разадресация дает содержащее по этому адресу значение `g[2]`:

```
*pointer_to_unsigned_long_long
```

Приведенная ниже полная программа демонстрирует, что три варианта следующего цикла эквивалентны:

```
#include <stdio.h>

int main()
{
unsigned long long g[10],
  *pointer_to_unsigned_long_long=&g[0];
int i;

for (i=0;i<10;++i) g[i]=i*i;
for (i=0;i<10;++i) *(g+i)=i*i;
for (i=0;i<10;++i)
  *(pointer_to_unsigned_long_long+i)=i*i;

for (i=0;i<10;++i)
  printf("\n g[%d]=%llu;    *(g+%d)=%llu;\
    *(pointer_to_unsigned_long_long+%d)=%llu",
    i,g[i],i,*(g+i),
    i,*(pointer_to_unsigned_long_long+i));
return 0;
}
```

Значения указателя совершенно необязательно должны последовательно увеличиваться, примером чего служит альтернативный вариант программы:

```
#include <stdio.h>

int main()
{
unsigned long long g[10];
unsigned long long
  *pointer_to_unsigned_long_long=&g[9];
int i,k;

for (i=0;i<10;++i) g[i]=i*i;
```

```
for (i=9;i>=0;--i) *(g+i)=i*i;
for (i=0;i<10;i++)
  *(pointer_to_unsigned_long_long-i)=(9-i)*(9-i);

for (i=0,k=9;i<10;++i,--k)
  printf("\n g[%d]=%llu;    *(g+%d)=%llu;\
    *(pointer_to_unsigned_long_long-%d)=%llu",
    i,g[i],i,*(g+i),
    k,*(pointer_to_unsigned_long_long-k));
return 0;
}
```

Важно лишь следить за адекватностью использования указателя, в частности, чтобы его значение не выходило за рамки диапазона адресов элементов массива. В данном примере `&g[0]` и `&g[9]`.

Если известно, что два указателя адресуют элементы одного и того же массива, их разность — это число элементов между ними:

```
#include <stdio.h>

int main()
{
unsigned long xxl[1000], *p1=0, *p2=0;
ptrdiff_t pdiff; /* специальный целый тип для хранения
разности указателей */

p1=&xxl[111]; p2=&xxl[333];
printf("\nЧисло элементов массива между указателями:\
 p2-p1=%td\n",pdiff=p2-p1);
return 0;
}
```

3.8. Комбинированные операции над указателем

В табл. 3.2 сведены некоторые характерные для Си сочетания операций над указателями, действие которых очень важно понимать, поскольку они порождают очень эффективный машинный код.

Таблица 3.2. Операции над указателями

Выражение	Действие	Возвращаемое значение
`*указатель++`	Обе операции имеют одинаковый приоритет и выполняются справа налево, поэтому происходит приращение указателя на единицу и взятие того, что он перед этим адресовал	Значение текущего (а не следующего) элемента массива, может быть левым и правым значением в выражении
`*указатель--`	Взять текущий элемент массива и затем сдвинуть указатель на предыдущий элемент	Значение текущего (а не предыдущего) элемента массива, может быть левым и правым значением в выражении
`*++указатель`	Сначала прирастить на единицу указатель, затем взять то, что он теперь адресует	Значение следующего элемента массива, может быть левым и правым значением в выражении
`*--указатель`	Сначала уменьшить на единицу указатель, затем взять то, что он теперь адресует	Значение предыдущего элемента массива, может быть левым и правым значением в выражении
`++*указатель` `--*указатель`	Сначала разадресовать указатель, а затем прирастить или уменьшить значение элемента, на который он указывает	Уменьшенное или увеличенное на единицу значение по тому же указателю
`(*указатель)++` `(*указатель)--`	Сначала разадресовать указатель, а затем прирастить или уменьшить значение элемента, на который он указывает	Значение по тому же указателю до его уменьшения или увеличения на единицу

`*указатель= выражение`
`*указатель+= выражение`
и другие комбинированные операции с присваиванием (§ 4.7)
	Сначала вычислить выражение справа от присваивания, затем разадресовать указатель и, наконец, выполнить присваивание или комбинированную операцию с присваиванием	Новое значение по тому же указателю

Все приведенные в табл. 3.2 ситуации иллюстрируются следующим полным примером:

```
#include <stdio.h>

int main()
{
long g[10], *pg, i, k;

/* инициализация и вывод массива */
for (i=0;i<10;i++) { g[i]=100*(i+1)+1;
 printf("\n g[%ld]=%ld",i,g[i]); }

k=g[5]; pg=&g[5]; i=*pg++;
printf("\n было: *pg=%ld; *pg++ дает %ld, теперь\
 *pg=%ld, указатель переместился на %td позицию\
 вправо",k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=*pg--;
printf("\n было: *pg=%ld; *pg-- даёт %ld, теперь\
 *pg=%ld, указатель переместился на %td позицию\
 влев"o,k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=*++pg;
printf("\n было: *pg=%ld; *++pg даёт %ld, теперь\
 *pg=%ld, указатель переместился на %td позицию\
 вправо",k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=*--pg;
printf("\n было: *pg=%ld; *--pg даёт %ld, теперь\
 *pg=%ld, указатель переместился на %td позицию\
 влев"o,k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=++*pg;
printf("\n было: *pg=%ld; ++*pg даёт %ld, теперь\
 *pg=%ld, указатель переместился на %td позиций\
 вправо",k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
```

```
k=g[5]; pg=&g[5]; i=--*pg;
printf("\n было: *pg=%ld; --*pg дает %ld, теперь\
 *pg=%ld, указатель переместился на %td позиций\
 влев"o,k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=(*pg)++;
printf("\n было: *pg=%ld; (*pg)++ дает %ld, теперь\
 *pg=%ld, указатель переместился на %td позиций\
 вправо",k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=(*pg)--;
printf("\n было: *pg=%ld; (*pg)-- дает %ld, теперь\
 *pg=%ld, указатель переместился на %td позиций\
 влев"o,k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=(*pg)=101010101;
printf("\n было: *pg=%ld; (*pg)=101010101 дает %ld,\
 теперь *pg=%ld, указатель переместился на %td\
 позиций вправо",k,i,*pg,pg-&g[5]);

for (i=0;i<10;i++) g[i]=100*(i+1)+1;
k=g[5]; pg=&g[5]; i=(*pg)+=2020000;
printf("\n было: *pg=%ld; (*pg)+=2020000 дает %ld,\
 теперь *pg=%ld, указатель переместился на %td\
 позиций вправо\n",k,i,*pg,pg-&g[5]);

return 0;
}
```

Далее приведен простой пример эффективного применения таких сочетаний операций:

```
#include <stdio.h>

int main()
{
long g[10], *pgb=&g[0], *pge=&g[9], *pg=&g[9], i=11;

while (i--,pg>=pgb) *pg-=i*10+1;
for (i=0;i<10;i++)
```

```
  printf("\n g[%ld]=%ld",i,g[i]);
printf("\n");

/* теперь pg=pgb-1 */
i=0; while (i++,pg<pge) *++pg=100*i+1;
for (i=0;i<10;i++)
  printf("\n g[%ld]=%ld",i,g[i]);

return 0;
}
```

Следующее выражение с указателями тоже относится к допустимым в Си и даже вполне эффективно реализуемым, хотя и трудно читается:

```
*(*ak+=*--s)+=--*g
```

Последовательность действий, вероятно[1], такова: *1*) взять указатель `g` и разадресовать его; *2*) из получившегося значения вычесть единицу и записать обратно по этому указателю; *3*) взять указатель `s` и уменьшить его на единицу, а затем разадресовать его; *4*) разадресовать указатель `ak` и прибавить к результату выражение п. 3; *5*) разадресовать результат п. 4 (т.е. содержимое круглых скобок); *6*) прирастить результат п. 5 на значение, полученное в п. 2, и записать обратно.

Вычислительный выигрыш от комбинации столь многих действий в одном выражении едва ли в общем случае оправдывает повышенную вероятность допустить ошибку. Однако в практических исходных текстах фрагменты такого рода встречаются, когда идет отчаянная борьба за предельную скорость счета, поэтому необходимо уметь хотя бы их читать.

Указателями нужно оперировать по следующим причинам:

◊ код с использованием указателей получается, как правило, значительно более эффективным, чем без них, поскольку компьютер «мыслит» категориями указателей и операции с ними требуют чрезвычайно мало машинных команд;

---

[1] Точная последовательность действий зависит от алгоритма оптимизации кода в компиляторе и может отличаться от приведенной. Поэтому такие фрагменты программ называются неустойчивыми и с ними необходимо быть очень осторожными (см. § 4.9).

◊ часты ситуации, когда, не прибегая к указателям, вовсе невозможно решить поставленную задачу, например безопасно передать в функцию или вернуть из функции большой массив.

Указателям отведено важнейшее место в Си (наряду с функциями), что является одной из основных причин высокой эффективности и выразительности этого языка.

3.9. Многомерные массивы и указатели

Указатели являются предпочтительным, а иногда и единственным средством эффективной и безопасной работы с массивами размерностью более единицы. Установлено, что имя одномерного массива – это не что иное, как указатель на его начальный элемент:

```
имя_массива ≡ &имя_массива[0]
```

Почти во всех реализациях языка это остается справедливым в случае многомерных массивов[1]. Можно утверждать, что при помощи указателя на начало массива можно адресовать все его элементы:

```
тип_массива  имя_массива[[размер_массива_по_данной_раз-
мерности]]n, *указатель_на_начало_массива;
...
указатель_на_начало_массива=имя_массива[[0]]n;
```

Дело в том, что под массив гарантированно выделяется непрерывный фрагмент адресного пространства оперативной памяти, поэтому все элементы располагаются друг за другом последовательно. Но адресация в памяти устроена линейно – в виде вектора, а массив определен программистом как многомерный, следовательно, компилятор должен придерживаться некоторого регулярного способа укладки элементов. Этот способ является всеобщим стандартом в

[1] Исключение составляют в основном более старые реализации языка, использующие вспомогательные индексные массивы.

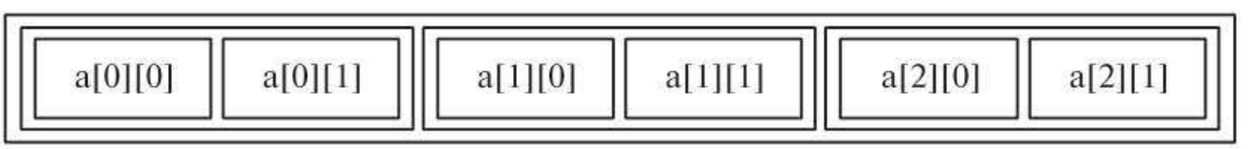
языке Си. Среди количеств элементов вдоль размерностей массива

$a[S_n][S_{n-1}]...[S_2][S_1]$

будем называть S_1 младшим размером, а S_n старшим. Запишем алгоритм размещения элементов.

1. Зафиксировать начало массива: $s_n=s_{n-1}=...=s_2=s_1=0$.
2. Выложить последовательно все элементы вдоль самой младшей размерности: $a[s_n][s_{n-1}]...[s_2][s_1]$, $s_1=0...S_1-1$.
3. $s_1=0$; перейти к более старшей размерности: $s_2=s_2+1$; Если $s_2>=S_2$, перейти к шагу 4, иначе перейти к шагу 2.
4. $s_1=0$; $s_2=0$; перейти к более старшей размерности: $s_3=s_3+1$; Если $s_3>=S_3$, перейти к шагу 5, иначе перейти к шагу 2.
5. $s_1=0$; $s_2=0$; $s_3=0$; перейти к более старшей размерности: $s_4=s_4+1$; Если $s_4>=S_4$, перейти снова к большей размерности, иначе перейти к шагу 2.
6. ...
7. $s_1=0$; $s_2=0$; ... $s_{n-1}=0$; перейти к самой старшей размерности: $s_n=s_n+1$; Если $s_n>=S_n$, весь массив пройден, иначе перейти к шагу 2.

На примере двухмерного массива графически результат можно представить так:



Такой способ размещения элементов многомерного массива в памяти называется *разверткой массива* и возможен исключительно при знании размеров массива вдоль размерностей. Отсюда следует, что при передаче массива по указателю этот указатель должны сопровождать размерности. Поскольку всякий указатель типизирован, т.е. жестко привязан к типу данных, информация о размерностях должна полностью содержаться в типе указателя. Подробнее речь об этом пойдет в гл. 3.

Остается решить вопрос об адресации отдельных элементов многомерного массива. Логично ее осуществлять при помощи вложенных циклов, что иллюстрируется полной программой:

```
#include <stdio.h>

#define I 3
#define J 4

int main()
{
long double b[I][J], a[I][J], *pa; /* два массива и указа-
тель */
int i, j;

pa=&a[0][0]; /* указатель pa указывает на начало массива a */

for (i=0; i<I; ++i)
  for (j=0; j<J; ++j)
    {
    b[i][j]=1000*i+j; /* инициализируем массив b */
    /* поэлементно копируем в массив по указателю pa */
    *(pa+J*i+j)=b[i][j];
   /* его содержимое "появилось" в массиве a, так как ука-
затель pa связан с массивом a */
    printf("\n b[%d][%d]=%Lg;     *(pa+%d*%d+%d)=%Lg;\
      a[%d][%d]=%Lg",
      i,j,b[i][j],  J,i,j,*(pa+J*i+j),  i,j,a[i][j]);
    }
return 0;
}
```

После выполнения этого фрагмента кода массивы `a` и `b` заполнены одинаковыми элементами. Следовательно, способы адресации `*(&a[0][0]+J*i+j)` и `a[i][j]` эквивалентны. Аналогичное справедливо и для бóльших размерностей.

Можно сформулировать правило: при проходе от начала многомерного массива к его концу самый правый индекс меняется быстрее всего; следующий индекс меняется в S_1 раз медленнее, третий справа еще в S_2 раз медленнее, чем второй справа, и так вплоть до самого левого старшего индекса, пробегающего весь диапазон значений всего единожды. Отсюда вытекает:

$$a[s_n][s_{n-1}]\ldots[s_2][s_1] \equiv *(\&a[0][0]\ldots[0][0] + S_{n-1}*S_{n-2}*\ldots*S_2*S_1*s_n + S_{n-2}*S_{n-3}*\ldots*S_2*S_1*s_{n-1} +$$

```
Sn-3*Sn-4*…*S2*S1*sn-2 + …
S3*S2*S1*s4 +
S2*S1*sn3 +
S1*s2 +
s1)
```

Легко видеть, что множество умножений выполняется повторно. Применение схемы Горнера дает более компактное выражение:

```
a[sn][sn-1]…[s2][s1] ≡ *(&a[0][0]…[0][0] +
s1 + S1*(s2 + S2*(s3 + S3*(s4 + S4*(s5 + …
*(sn-3 + Sn-3*(sn-2 + Sn-2*(sn-1 + Sn-1*sn))))))));
```

Ничего особенно сложного в адресной арифметике применительно к многомерным массивам нет, требуется лишь аккуратность. Действия еще более упрощаются при однородном заполнении массива, когда содержимое всякого элемента определяется лишь его адресом.

Например:

```
unsigned usi[I][J][K], i;
for (i=0; i<I*J*K; i++) *(usi+i)=i;
```

3.10. Связь скобочной индексации и адресации указателем

Для более глубокого понимания устройства многомерных массивов в Си рассмотрим, как компилятор обрабатывает запись с квадратными скобками. Выше обсуждалось, что в одномерном случае смещение вдоль единственной размерности происходит с шагом, равным размеру элемента массива `size_of(тип_элемента)`, указатель `a` имеет тип `(тип_элемента*)`:

```
тип_элемента a[I]; /* определение массива */
```

при этом `a[i] ≡ *(a+i)`.

В двухмерном случае шаг по левой и правой размерностям различается: по правой он по-прежнему совпадает с размером элемента:

```
size_of(тип_элемента),
```

а по левой – равен числу элементов вдоль правой размерности, умноженной на размер элемента `J*size_of(тип_элемента)`:

```
тип_элемента b[I][J]; /* определение массива */
```

т.е. `b[i][j]` ≡ `*(*(b+i)+j)`. Переменная `b` имеет тип указателя на массив из `J` элементов типа `тип_элемента` каждый: `тип_элемента(*b)[J]`. Без круглых скобок последнее выражение читалось бы неправильно: «`b` – это массив из `J` указателей на `тип_элемента`». Разадресовав `b` при помощи внутренней звездочки, получаем указатель `*(b+i)` на `тип_элемента`, которым оперируем, как и в случае одномерного массива.

Трехмерный массив строится аналогично:

```
тип_элемента c[I][J][K]; /* определение массива */
```

где `c[i][j][k]` ≡ `*(*(*(c+i)+j)+k)`. Тип переменной `c` читается так: «указатель на массив из `J` элементов типа массив из `K` элементов типа `тип_элемента` каждый»: `тип_элемента(*c)[J][K]`. Первая, самая вложенная разадресация осуществляет «снятие» самой левой размерности и переход к средней размерности массива. Таким образом, выражение `*(c+i)` имеет тип «указатель на массив из `K` элементов типа `тип_элемента` каждый»: `тип_элемента(*)[K]`. Выражение `*(*(c+i)+j)` типа «указатель на элемент типа `тип_элемента`» позволяет адресовать элементы исходного типа непосредственно, удалив среднюю размерность. Наконец, самая внешняя разадресация доставляет значение искомого элемента массива `c`.

Ядро компилятора, занимающееся глубокой оптимизацией кода, как правило, «понимает» лишь язык указателей, поэтому запись обращения к элементу массива посредством квадратных скобок предварительно преобразуется в эквивалентное

выражение с указателями (иногда это делает препроцессор). В самом общем случае[1] определение массива:

```
тип_элемента имя_массива{[I]}^n;
```

где имя_массива понимается так:

```
имя_массива{[I]}^n ≡
    *(*( … *(*(имя_массива+s_n)+s_n-1)+ … s_2)+s_1)
```

Понимание представленных выше преобразований имеет важное применение: можно использовать удобную индексацию при помощи [] для массивов, выделенных динамически (см. гл. 9, посвященную стандартной библиотеке Си). Для этого достаточно правильно преобразовать тип указателя.

Например:

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

/* размерности массива */
#define I 2
#define J 3
#define K 4

int main()
{
unsigned i, j, k;
double (*a)[I][J][K]; /* это структурный указатель на
массив, а не сам массив */
double *pa; /* это неструктурный указатель на массив */

/* выделение места для массива */
pa=(double*)malloc(I*J*K*sizeof(double));
for (i=0;i<I;++i)
  for (j=0;j<J;++j)
    for (k=0;k<K;++k)
      printf("\npa[%u][%u][%u]=%f",i,j,k,
        *(pa+(i*J+j)*K+k)=k+10*j+100*i);
        /* инициализация */
```

[1] Возможные спецификаторы хранения опущены, чтобы не усложнять и без того непростую конструкцию.

```
/* присваивание указателя с приведением типа */
a=(double(*)[I][J][K])pa;

for (i=0;i<I;++i)
  for (j=0;j<J;++j)
    for (k=0;k<K;++k)
      printf("\na[%u][%u][%u]=%f", i, j, k,
        (*a)[i][j][k]); /* (*a) - это массив */

free(pa); return 0; /* освобождение памяти и возврат */
}
```

3.11. Указатель и модификатор `const`

Модификатор `const` придает неизменяемый характер объекту, к которому относится.

Например:

```
const int ic=255; /* ic теперь нельзя изменить */
```

Значение константы надо определить сразу, в момент ее определения, потом будет поздно — ведь это константа. Это значение можно использовать для того, чтобы защитить некоторый объект от изменения даже путем «окольного» доступа к нему по указателю.

Например:

```
const int *pic; /* pic - это неконстантный указатель на
какую-то (пока неопределенную) константу типа int */
int const *pic1; /* полностью эквивалентная предыдущей
запись: pic1 - это также неконстантный указатель на ка-
кую-то (пока неопределенную) константу типа int */
pic=&ic; /* так как указатель неконстантный, его значение
можно поменять */
*pic=1000; /* ошибка: поменять значение константы не уда-
лось, даже опосредованно */
```

Здесь `pic` и `pic1` — указатели на константу типа `int`. В то же время существует возможность определить константный ука-

затель, навсегда привязав его к данному объекту. Сам объект при этом может быть переменной или константой.

Например:

```
int i, j;
int * const pic=&i; /* pic - это константный указатель на
переменную i типа int; переопределить его для указания на
любой другой объект уже нельзя */
pic=&j; /* ошибка: нельзя изменить значение константного
указателя */
*pic=1000; /* допустимо записать в переменную i новое
значение при помощи указателя, пусть даже и константного
*/
```

Непосредственное изменение переменной `i` при этом также не запрещается, поскольку она остается переменной. Очевидно, как и всякий константный объект, константный указатель обязательно инициализируется в момент создания (определения); после это сделать просто невозможно. Двукратное употребление модификатора `const` в разных контекстах приводит к появлению константного указателя на константу: указатель должен быть инициализирован в момент определения и впоследствии не разрешается менять ни сам указатель, ни переменную `i` с его помощью.

Например:

```
const int i=9, j=-9; /* две константы типа int */
const int * const pic=&i; /* константный указатель pic на
константу i навсегда к ней привязан */
int const * const pjc=&j; /* эквивалентная предыдущей
запись: константный указатель pjc на константу j навсегда
к ней привязан */
pic=&j; /* ошибка: поменять константный указатель нельзя
*/
*pic=1000; /* ошибка: по указателю нельзя поменять значе-
ние константы */
```

Правильно написать определение или декларацию с ключевым словом `const` можно по общему правилу: читаем *справа налево* с учетом повышенного приоритета того, что заключено в круглые скобки. Если текст получается осмысленным, описа-

ние верно. Сфера использования константных указателей и указателей на константный объект – защита от непреднамеренного изменения. К ней рекомендуется прибегать всюду, где это возможно, обеспечивая автоматическую проверку корректности программы силами компилятора и экономя усилия на этапе отладки. Такие дополнительные проверки незначительно сказываются на скорости компиляции и эффективности кода.

3.12. Спецификатор указателя `restrict`

Язык Си допускает адресацию одного и того же объекта двумя и более различными указателями в одном и том же программном блоке (например, функции).

Например:

```
long a[100], *pa=a, i=0;
…
a[i]=i;
…
*(pa+i)=i*i;
…
```

Такое положение дел называется двойным (или множественным, англ. pointer aliasing) указанием. Изменение значения по одному указателю автоматически меняет значение по другому указателю. Это может быть удобно, но заметно повышает риск создания неправильно работающей программы, так как программисту приходится постоянно отслеживать, что где меняется.

К сожалению, имеется еще одна проблема: перемежающееся использование нескольких указателей на одну и ту же область памяти (пусть и на разные ее места) не позволяет современным компиляторам генерировать исполнимый код, «выжимающий» максимум производительности современных процессоров. Дело в том, что в процессорах используется автоматическое предсказание, какой фрагмент памяти понадобится далее (англ. speculative memory access). Оно позволяет заранее прочитать эти блоки из довольно медленной оперативной

памяти и поместить их в быструю кэш-память. При записи в память тоже есть выигрыш: процессор получает «подсказку», какие блоки памяти в ближайшее время не понадобятся, и сбрасывает их содержимое в ОЗУ, освобождая вечно занятую маленькую кэш-память для актуальных данных. Если в блоке программы по отношению к некоторой области памяти гарантированно отсутствует множественное указание и на нее имеется *ровно один* указатель и этот указатель на протяжении всего своего времени жизни (англ. life time) указывает только на этот объект, и ни на какой другой, то программист может предложить компилятору включить для этого объекта новые методы оптимизации:

```
тип_данных *restrict имя_указателя;
```

Ответственность за эту просьбу целиком возлагается на программиста. Если множественное указание все же имеет место, не гарантируется не только эффективная работа программы, но даже ее правильная работа, поскольку данные в ОЗУ и в кэш-памяти могут не совпадать, а процессор об этом «не знает» и не приведет их в соответствие друг другу.

3.13. Общий способ задания типа

Объединим частные случаи деклараций и определений отдельных переменных, констант и их массивов в конструкцию самого общего вида.

Например:

```
[спецификатор_типа_хранения]
[модификатор_типа] описатель_базового_типа
[модификатор_типа_для_указателя]
имя_переменной_или_массива_1 [описание_размерности]
[, [модификатор_типа_для_указателя]
имя_переменной_или_массива_m [описание_размерности]n]m;
```

Здесь:

спецификатор_типа_хранения ≡ <auto, extern, register, static, volatile>, для локальных переменных функций по умолчанию это auto,

`модификатор_типа` ≡ `[const][<signed, unsigned>]`
`[<short, long, long long>]`
`описатель_базового_типа` ≡ один из встроенных или определенных пользователем сложных типов. К пользовательским типам, очевидно, неприменимы модификаторы типов `signed, unsigned, short, long, long long`, ибо нет возможности объяснить компилятору, как их реализовать для объекта произвольного типа. Сложные типы можно задать, используя `typedef, struct, enum` и описание указателя на функцию с определенным типом возвращаемого значения и заданным набором передаваемых параметров;
`спецификатор_типа_для_указателя` ≡ `[restrict] [const]` и некоторое количество знаков указателя `*` и парных круглых скобок для указания порядка чтения задаваемого типа,
`имя_переменной_или_массива_i` ≡ допустимое в Си имя переменной или константы;
`описание_размерности` ≡ парное число квадратных скобок – пустых или с константными выражениями внутри – присутствует только в случае описания массива.

Приведенная выше конструкция читается *справа налево* с учетом приоритета, задаваемого круглыми скобками, если они есть, а именно: сначала отыскивается и читается имя объекта, следом то, что находится в скобках самого высокого уровня вложенности, затем – меньшего и т.д., затем – все остальное.

Например:
```
int *pin[6], (*pen)[5], *(*pan)[4];
```

Первый из этих объектов читается так: «`pin` – это массив из 6 указателей на `int`», второй: «`pen` – это указатель на массив из 5 элементов типа `int`», а третий: «`pan` – это указатель на массив из 4 указателей на `int`». Как уже обсуждалось, при описании многомерного массива его индексы с увеличением адреса в памяти (приращением указателя) меняются так, что крайний правый из них меняется наиболее быстро, а крайний левый – наименее быстро.

Например:
```
double xyz[10][20][30];
```

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Можно ли, анализируя указатель, приведенный к типу `(void*)`, понять, на какой тип он указывал до преобразования к `(void*)`?
2. Можно ли при структурной инициализации многомерного массива пропустить несколько не последних элементов по данному измерению? Если можно, как это сделать? Если нельзя, почему?
3. Напишите функцию однократного обращения ко всем элементам квадратной матрицы `a[N][N]` по сворачивающейся против часовой стрелки спирали начиная с элемента `a[N-1][N-1]`.
4. Имеются два указателя: `pa` на массив из `M` элементов типа `double` и `pb` на массив из `N` элементов типа `float`. Реализуйте безопасную проверку того, непосредственно ли друг за другом эти массивы следуют в оперативной памяти.
5. Организуйте выделение памяти и присвойте указатели `pa` и `pb` на двухмерные массивы `unsigned a[M][N]` и `unsigned b[M][N]` (`M`- и `N`-четные) так, чтобы верхняя часть матрицы `M×N`, адресуемой `pa`, и нижняя часть матрицы `M×N`, адресуемой `pb`, занимали в памяти одно и то же место. Продемонстрируйте работоспособность вашего решения.
6. Объясните, как в Си реализовать массив произвольной размерности, опираясь на концепцию «массива массивов».
7. Имеется трехмерный массив `double t3[P][Q][R]`. Реализуйте его альтернативную адресацию как двухмерного `double t21[P*Q][R]` и как двухмерного `double t21[P][Q*R]`.
8. Сравните быстродействие различных способов адресации элемента в большом двухмерном массиве, усреднив его для большого числа произвольно выбранных элементов. Какие выводы отсюда следуют?

ГЛАВА 4 ОПЕРАЦИИ И ВЫРАЖЕНИЯ

4.1. Одноместные, двухместные и трехместная операции

Напомним, что всякая базовая операция в Си имеет смысл одиночного действия, не разделимого на более мелкие. Как правило, базовой операции соответствует одиночная команда процессора, выступающая минимальным неделимым квантом вычислений. Результат выполнения операции всегда представляет собой конкретное численное значение. Из операций и операндов составляются *выражения* (англ. expressions), являющиеся более крупной частицей вычислительного процесса. Одноместные операции работы с указателями `*` и `&` и операции обращения к элементу массива `[]` или полю структуры «точка» и `->` вследствие их узкой специфики и значительной сложности рассматриваются отдельно.

В зависимости от числа операндов, участвующих в операции, в языке Си операции бывают:

- одноместные (унарные, англ. unary), предусматривающие один операнд;
- двухместные (бинарные, англ. binary) с двумя операндами (не путать с понятием «двоичный», для которого в английском языке используется то же слово);
- трехместная (тернарная, англ. ternary), у которой всегда три операнда.

Результат операции не входит в число ее операндов, но может выступать как операнд вышестоящей операции. Это сделано, чтобы обеспечить гибкую возможность игнорирования или дальнейшего преобразования значения результата или его типа в рамках составного выражения при полном сохранении структуры исходного подчиненного выражения. Так кон-

струируются выражения произвольной сложности. Отличительной особенностью Си является наличие развитого набора операций. Отчасти поэтому и вследствие ограниченности набора допустимых символов некоторые операции используют один и тот же символ. Это тем не менее не ведет к конфузу, потому что из контекста всегда ясно, какая операция имеется в виду. Например, если значок операции окружен двумя операндами, ясно, что подразумевается двухместная операция. В нечастых случаях двусмысленности применяют разделение операций при помощи круглых скобок или пробелов.

С точки зрения компилятора выбор конкретной операции в выражении зависит от *типов* задействованных в ней операндов. Даже такая простая операция, как умножение, используется в одном случае для аргументов типа `int`, в другом – для аргументов типа `unsigned long long`, в третьем – для `double` и т.д. Причины такой «повышенной разборчивости» заключаются в следующем:

- ◊ для разных сочетаний типов операндов компилятор привлекает разные команды процессора (англ. instructions);
- ◊ далеко не всякое возможное сочетание типов операндов соответствует команде процессора, поэтому компилятор может прибегнуть к неявному преобразованию типа, в том числе небезопасному (см. § 2.1).

Поэтому от типов конкретных операндов зависит как выбор возможных операций над ними, так и особенности их выполнения. Например, при сложении достаточно больших целых чисел может произойти переполнение разрядной сетки компьютера с выдачей неверного результата, поскольку он не представи́м при данном числе разрядов. Однако во многих реализациях этот досадный факт не будет автоматически обнаружен в ходе выполнения программы. Иное дело – возникновение переполнения при сложении чисел с плавающей точкой. Такая ситуация обычно приводит к появлению результата специального вида: `Inf`, что гораздо проще заметить. Далеко не всякая операция применима к произвольным типам операндов. Так, указатель нельзя ни на что ни поделить, ни умножить, а для операнда действительного типа с плавающей точкой не определен побитовый сдвиг.

4.2. Арифметические операции

Операции данного вида всегда имеют своим результатом число и приведены в табл. 4.1, где приводится *модальность* (англ. modality) операций – неотрицательное целое число, сообщающее о том, насколько приоритетно вычисление этой операции по сравнению с другими. Чем модальность меньше, тем более приоритетна операция. Так, двухместные мультипликативные операции (умножение, деление, получение целого остатка от деления) имеют модальность 2 и, следовательно, выполнятся *перед* двухместными аддитивными операциями (сложение и вычитание), обладающими модальностью 3.

Таблица 4.1. Арифметические операции (стрелка показывает порядок осуществления одинаковых операций этого типа в выражении)

Запись операции	Действие	Число операндов, модальность и порядок выполнения	Значение операнда	Результат
`x+y`	Сложение `x` и `y`	Двухместная, 3, →	Произвольные числа	Число наиболее сложного типа среди типов операндов
`x-y`	Вычитание `y` из `x`			
`x*y`	Умножение `x` на `y`	Двухместная, 2, →		
`x/y`	Деление `x` на `y`		Произвольные числа, `y≠0`; в случае операции `%` – оба операнда – только целые числа	
`x%y`	Целый остаток от деления `x` на `y`			
`-x`	Число `x` с противоположным знаком	Одноместная, 1, ←	Произвольные числа, кроме типов со спецификатором `unsigned`, для которых в ряде случаев значение операции может быть неопределенным	

Окончание табл. 4.1

Запись операции	Действие	Число операндов, модальность и порядок выполнения	Значение операнда	Результат
`++x`	Увеличение (инкремент, англ. increment) значения переменной `x` на единицу	Одноместная, 1, ←, префиксная, поскольку переменную сначала меняют, потом используют	Произвольные числа в переменных	Значение `x` после приращения на единицу
`x++`		Одноместная, 0, →, суффиксная (постфиксная), поскольку сначала используют старое значение переменной, а потом ее меняют		Значение `x` до приращения на единицу
`--x`	Уменьшение (декремент, англ. decrement) значения переменной `x` на единицу	Одноместная, 1, ←, префиксная, поскольку переменную сначала меняют, потом используют	Произвольные числа в переменных	Значение `x` после убавления на единицу
`x--`		Одноместная, 0, →, суффиксная (постфиксная), поскольку сначала используют старое значение переменной, а потом ее меняют		Значение `x` до убавления на единицу

Например:

```
z=2*a++;
```

Значение переменной `a` увеличилось на единицу (и было записано обратно в `a`), а результат умножения старого значения `a` на 2 помещен в переменную `z`. Несколько примеров собраны в полную программу, демонстрирующую рассмотренные операции:

```
#include <stdio.h>

int main()
{
double a=3.1, b=2, c=-6.9, d=3;
```

```
long k=23, l=5;

printf("\n a=%g, b=%g, c=%g, d=%g",a,b,c,d);
printf("\n -a*b+c/d-5=%g",-a*b+c/d-5);
printf("\n k=%ld, l=%ld, k%%l=%ld",k,l,k%l);
d=b; b++; printf("\n значение b до инкремента:\
 %g, после инкремента b++: %g",d,b);
d=a--; printf("\n d=a-- дает d=%g, a=%g",d,a);
d=--c; printf("\n d=--c дает d=%g, c=%g",d,c);

return 0;
}
```

4.3. Логические операции

Логические операции реализуют стандартную алгебру двоичной логики высказываний. В качестве результата они могут возвращать одно из всего лишь двух возможных значений — «истина» или «ложь». Поскольку почти все современные процессоры нацелены именно на работу с числами, в стандарте языка Си принято, что *ложь* кодируется нулем, а *истина* — это любое ненулевое значение (даже отрицательное). Поэтому результатом всякой логической операции в Си является число. Доступные логические операции представлены в табл. 4.2. Как и обычно: чем меньше модальность, тем выше приоритет операции.

Таблица 4.2. Логические операции (стрелка показывает порядок осуществления одинаковых операций этого типа в выражении)

Запись операции	Действие	Число операндов, модальность и порядок выполнения	Значение операнда	Результат
`!x`	Логическое отрицание	Одноместная, 1, ←	Произвольные числа	Противоположная логическая величина: для «истины» это «ложь» и наоборот

Окончание табл. 4.2

Запись операции	Действие	Число операндов, модальность и порядок выполнения	Значение операнда	Результат

`x < y`
`x > y`
`x <= y`
`x >= y`
	Проверка на строгое или нестрогое неравенство	Двухместная
		5,
		→
			То же	«Истина», если неравенство выполняется; иначе «ложь»

`x == y` `x != y`	Проверка на строгое равенство или неравенство	Двухместная, 6, →	—»—	«Истина», если равенство (неравенство) выполняется; иначе «ложь»
`x && y`	Логическое «И» (конъюнкция)	Двухместная, 10, →	—»—	«Истина», если оба операнда ненулевые; иначе «ложь»
`x || y`	Логическое «ИЛИ» (дизъюнкция)	Двухместная, 11, →	—»—	«Истина», если хотя бы один операнд ненулевой, иначе «ложь»

Например:

```
int a=17, b=17, c=-6;
c=c || (a-b); /* a-b дает 0, но в c содержится -6, в результате в переменную c попадет ненулевое значение */
```

В большинстве систем программирования результирующее значение логического выражения, соответствующее «истине», кодируется единицей. Вероятно, в приведенном примере именно единица будет записана в `c`. Однако программист категорически не должен делать предположений на этот счет. Гарантируется лишь одно: соответствующий «истине» результат ненулевой.

4.4. Побитовые операции

Побитовые операции применяются к отдельным битам двоичного представления операнда независимо в отличие от логических операций, которые вовлекают в действие свои операн-

ды целиком как единое число. В этом смысле побитовые операции являются более «тонкими». Обозначив через $z_i=f(x_i)$ зависимость индивидуального бита z_i результирующего числа z от находящегося в той же позиции (считая с младшего разряда) бита x_i операнда x, в случае одноместной операции f можно записать результат для всего восьмибитового (однобайтного) числа:

Операнд x	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
Результат операции	$f(x_7)$	$f(x_6)$	$f(x_5)$	$f(x_4)$	$f(x_3)$	$f(x_2)$	$f(x_1)$	$f(x_0)$

Для двухместных операций влияние битов на результат таково:

Первый операнд x	x_7	x_6	...	x_1	x_0
Второй операнд y	y_7	y_6	...	y_1	y_0
Результат операции z	$f(x_7, y_7)$	$f(x_6, y_6)$	...	$f(x_1, y_1)$	$f(x_0, y_0)$

Если длины операндов различны: `sizeof(x)!=sizeof(y)`, применяется общее правило: сначала находится самый длинный из них, а затем до осуществления операции короткий операнд приводится к типу длинного, как и обычно, стандартным преобразованием типа.

Побитовые операции представлены в табл. 4.3.

В Си побитовые операции применимы только к целочисленным операндам в отличие от логических. Простые примеры использования этих операций представлены в следующей полной программе, полезной для проверки вручную полученных результатов:

```
#include <stdio.h>

int main()
{
unsigned char a=0x7A;
short b=-8;
unsigned long c=0xa, d=0x7000B0F0;

printf("\n a=0x%hhx=%hhu,  b=0x%hx=%hd,\
```

```
  c=0x%lx=%lu,  d=0x%lx=%lu",a,a,b,b,c,c,d,d);
printf("\n ~b=%hd, a&d=0x%lx, a^d=0x%lx, c|d=0x%lx",
  ~b, a&d, a^d, c|d);
printf("\n a>>2=0x%hhx=%hhu,  b>>1=0x%hx=%hd,\
  b<8=0x%hx=%hd,  d<12=0x%lx=%lu",
  a>>2,a>>2,b>>1,b>>1,b<<8,b<<8,d<<12,d<<12);

return 0;
}
```

Таблица 4.3. Побитовые операции (стрелка показывает порядок осуществления одинаковых операций этого типа в выражении)

Запись операции	Действие	Число операндов, модальность и порядок выполнения	Значение операнда	Результат
`~x`	Побитовое отрицание	Одноместная, 1, ←	Произвольный целый тип	Число, где каждый бит сменил свое значение на противоположное по сравнению с `x`
`x << y` `x >> y`	Сдвиг	Двухместная, 5, →	`x` – произвольное целое число, которое сдвигают на `y` позиций в направлении двойной стрелки, значение `y` должно быть неотрицательным целым	Число, где x_i бит операнда стал x_{i-y} битом при сдвиге влево или x_{i+y} битом при сдвиге вправо. Все биты, вышедшие за пределы разрядной сетки, теряются
`x & y`	Побитовое «И» (поразрядная конъюнкция)	Двухместная, 7, →	Произвольные целые типы	`f(`x_i`, `y_i`)` равно `1`, только если x_i`==1` и y_i`==1`, иначе `0`
`x ^ y`	Исключающее «ИЛИ» (англ. exclusive OR, сокращенно EOR, XOR)	Двухместная, 8, →	Произвольные целые типы	`f(`x_i`, `y_i`)` равно `0`, если x_i`==`y_i, иначе `1`

Окончание табл. 4.3

Запись операции	Действие	Число операндов, модальность и порядок выполнения	Значение операнда	Результат
`x | y`	побитовое «ИЛИ» (поразрядная дизъюнкция)	Двухместная, 9, →	Произвольные целые типы	`f(`x_i`, `y_i`)` равно `0`, если x_i`==0` и y_i`==0`, иначе `1`

4.5. Особенности и свойства сдвига

Ответ на вопрос, чем заполнятся позиции справа при сдвиге влево и позиции слева при сдвиге вправо, дает табл. 4.4.

Таблица 4.4. Заполнение вакантных позиций при сдвиге

Направление сдвига	Сдвиг влево <<		Сдвиг вправо >>	
Тип левого операнда	Знаковый тип	Беззнаковый тип	Знаковый тип	Беззнаковый тип
Положительное значение	Нулями			
Отрицательное значение	Нулями		Единицами	–

Отрицательное значение при сдвиге вправо остается отрицательным потому, что во всех позициях, освобождаемых при сдвиге, размещаются единицы, в том числе в знаковом бите. Эта операция называется *размножением знака* (англ. sign propagation). Сдвиг влево положительной или отрицательной величины знакового типа может привести к положительному или отрицательному результату в зависимости от значения того бита, который окажется в знаковом разряде. В операциях сдвига, доступных в языке Си, знаковый разряд ничем не отличается от других. Явление выхода ненулевых разрядов результата за пределы разрядной сетки слева называется *переполнением* (англ. overflow), а справа – *потерей точности* (иногда называемой «недополнение», англ. underflow).

Как следует из способа двоичной записи в прямом коде (см. § 1.1 и 1.2), при условии, что есть запас разрядности между старшей единицей числа и знаковым (старшим) разрядом, гарантирующий невозможность переполнения, побитовый сдвиг неотрицательного числа `x` на `y` позиций влево в точности соответствует умножению `x` на 2^y, тогда как побитовый сдвиг неотрицательного числа `x` на `y` позиций вправо эквивалентен целочисленному делению `x` на 2^y. При этом потеря точности такая же, как при обычном целочисленном делении. Вследствие простоты операция сдвига почти во всех современных процессорах выполняется за один такт и является одной из самых быстрых в отличие от умножения и деления, которые на отдельных архитектурах требуют единиц и даже десятков машинных циклов. Поэтому программисты нередко используют арифметическое действие сдвига для ускорения счета. При этом следует быть чрезвычайно осторожным с содержимым знакового разряда. Рассмотренное правило арифметического действия не распространяется на числа в дополнительном коде, т.е. отрицательные. Для них некоторый арифметический эквивалент сдвига тоже имеется, но он значительно сложнее и его использование в программистской практике неоправданно.

4.6. Операции первичного доступа

Самый высокий приоритет (нулевая модальность) в Си присвоен операциям, с помощью которых можно обратиться к подчиненному элементу данных. Краткие сведения об этих операциях сведены в табл. 4.5.

Будем различать роли пары круглых скобок в программе:

- установка последовательности вычислений (только эта роль соответствует рассматриваемой в данном параграфе операции);
- преобразование типа выражения;
- обрамление выражения или типа в операции `sizeof`;
- вызов функции и передача параметров в нее;
- обрамление проверяемого условия или управляющей группы выражений в операторах `if`, `for`, `while`, `do...while`, `switch`.

Таблица 4.5. Операции первичного доступа

Запись операции	Действие	Число операндов, модальность, порядок выполнения	Значения операндов	Результат
`(x)`	Повышение приоритета выражения `x` в составном выражении. Чем глубже уровень скобок, тем раньше вычисляется их содержимое. Гарантируется, что, каким бы низким ни был приоритет операции, заключенное в круглые скобки выражение будет вычислено ранее, чем к нему применена первая из находящихся снаружи скобок соседних операций	Одноместная, 0, →	Произвольное выражение `x`	Значение `x`
`x[y]`	Доступ к значению элемента массива по его индексу. Если массив многомерный, применяется последовательно столько раз, какова размерность массива	Двухместная, 0, →	`x` — имя соответствующего массива, `y` — неотрицательное целое — индекс элемента	`y`-й элемент массива `x` как переменная
`x->y`	Доступ к полю `y` структуры по указателю `x` на нее. Если поле `y` структуры тоже является указателем на структуру, к нему можно снова применить операцию `->`. Если поле `y` структуры является структурой, к нему можно применить операцию «точка»	Двухместная, 0, →	`x` — указатель на структуру данного типа, `y` — имя поля в структуре	Поле `y` структуры как переменная
`x.y`	Доступ к полю `y` структуры по ее имени `x`. Если поле `y` структуры тоже является структурой, к нему можно снова применить операцию «точка». Если поле `y` структуры является указателем на структуру, к нему можно применить операцию `->`	Двухместная, 0, →	`x` — имя структуры данного типа, `y` — имя поля в структуре	Поле `y` структуры `x` как переменная

Например, ниже даны выражения, где скобки меняют порядок вычисления (а также смысл выражения и его значение):

```
a=10-b==0; /* 1) tmp=10-b 2) a=(tmp==0); */
a=10-(b==0); /* 1) tmp=(b==0) 2) a=10-tmp; */
```

4.7. Операции присваивания

Операция присваивания =, как и операция проверки на равенство ==, является двухместной. Начинающие программисты их часто путают вследствие сходства записей. Как правило, в результате такой ошибки все равно получается допустимое выражение, которое компилятор не отмечает как ошибку, однако такая программа не будет правильно работать. Правый операнд операции присваивания — это вычисленное значение выражения, которое будет записано в левый операнд. Очевидно, правым операндом может быть произвольное выражение подходящего типа, а левым – только такое, куда его значение можно записать, т.е. переменная или разадресованное выражение с указателем, являющееся эквивалентом переменной. Все выражения, которые могут встречаться справа от знака присваивания, называются *правыми выражениями* (англ. R-value, сокр. от right value), а допустимые слева от операции присваивания выражения именуются *левыми выражениями* (англ. L – value, сокр. от left value).

Например, приведем недопустимые левые выражения:

```
short a, b, d;
const short c;

b+d=17; /* не ясно, какая часть числа 17 должна попасть в
b, а какая в d */
a-17=b; /* язык Си сам по себе не умеет решить даже про-
стейшее уравнение */
c=a+b; /* поменять значение константы нельзя */
&a=&b; /* переменная a имеет фиксированный адрес, поэтому
&a - константа */
```

Наиболее характерные правильные левые выражения даны ниже:

```
double d[2][4], e, *pg;
int i=0, j=1;

d[i][j]=78.9;
e=-19.24;
pg=&d[0][0];
*pg=0.06;
*(pg+j+i*4)=e - 2e-7;
```

Довольно часто в практических алгоритмах требуется выполнить вычисление с участием некоторой переменной и записать результат обратно в эту же переменную.

Например:

```
e=e+i;
```

Подобные действия современные процессоры производят очень быстро – за один такт. Чтобы записывать их лаконично, заодно давая компилятору намек на выбор простой команды, в язык Си введены комбинированные арифметические и побитовые операции с присваиванием: `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`

Каждая такая операция, хотя и состоит из двух символов, является неделимой, не позволяет ставить между ними разделители и интерпретируется следующим образом:

```
левое_значение =
левое_значение операция правое_значение
```

Например:

```
e+=i*sin(j/1000.0); /* e=e+i*sin(j/1000.0) */
d[i][j]*=e+2.0; /* d[i][j]= d[i][j]*(e+2.0) */
i>>=(j-2); /* эквивалентно i=i>>(j-2), следите, чтобы выполнялось j>=2 */
```

4.8. Цепочечное присваивание

Модальность всякой связанной с присваиванием операции равна 13 (т.е. ее приоритет весьма низок), операции выполняются справа налево, а результирующее значе-

ние — это новое значение ее левого операнда. Отсюда вытекает возможность выстраивать присваивания «паровозиком».

Например:

```
e = *(pg+i) = d[i][j] = -45.2+e;
```

Такое цепочечное присваивание вследствие обработки справа налево и очень низкого приоритета операции присваивания равноценно последовательности:

```
d[i][j] = -45.2+e;
*(pg+i) = d[i][j];
e = *(pg+i);
```

Самый лучший способ проверки для выражений с цепочечным присваиванием — разбиение такой составной конструкции на элементарные части справа налево с учетом приоритета и скобок, чтобы убедиться: запрограммировано в действительности именно то, что имелось в виду.

4.9. Точки следования

В цепочку язык разрешает выстраивать произвольные операции, в том числе комбинированные операции присваивания. Некоторые амбициозные программисты, полагающие, что отлично знают Си, иногда прибегают к такого рода конструкциям:

```
e -= *(pg+ (++i)) = d[i][j]*= i+=-2+(e*=5);  /* (*) */
```

При этом генерируется очень эффективный исполнимый код. Но далеко не всегда правильный. Дело не только в плохой читаемости исходного текста программы: очевидно, в приведенном примере следует тщательно отслеживать изменения `i`. Проблема состоит в том, что здесь не работает сама проверка — продемонстрированное выше разбиение такой конструкции на элементарные части:

```
e *= 5;                                  /* (а) */
i += -2+e;                               /* (б) */
```

```
d[i][j] *= i;                    /* (в) */
*(pg+i+1) = d[i][j];             /* (г) */
i++;                             /* (д) */
e -= *(pg+i);                    /* (е) */
```

Каждое выражение в разбиении (а) – (е) закончено точкой с запятой. Это делает его завершенным оператором: компилятор гарантирует однозначность его результата. Про исходное выражение этого сказать нельзя, поскольку в процессе вычисления (*) используются нефиксированные и неопределенные промежуточные результаты, вследствие чего возникают вопросы, на которые язык не отвечает:

- при вычислении (б) брать старое или новое значение `e` из (а)?
- в (в) и (г) слева и справа от присваивания использовать старое или новое значение `i` из (б)?
- в (г) присваивать старое или новое `d[i][j]` из (в),
- учитывать ли в (е) изменения `*(pg+i)`, `e` и `i` на предыдущих шагах? А если да, то какие?

Система приоритетов операций оставляет эти вопросы открытыми, лишь определяя последовательность действий, но не их побочные последствия. Поэтому вопросы «откуда что брать» каждый компилятор разрешает по собственному усмотрению согласно действующему алгоритму оптимизации кода, что приводит, как говорят, к *неустойчивости вычислений* (англ. instability of computations), являющейся чрезвычайно трудно-обнаружимой ошибкой. Она проявляется после перекомпиляции исходного текста другой версией или типом компилятора, а также после переноса программы на иную вычислительную платформу. Очевидный способ избежать незавидной участи ловца такого изъяна – отказаться от комбинации изменения переменной «на месте»: `++i`, `i++`, `--i`, `i--`, `i+=выражение` и тому подобных конструкций совместно с операцией присваивания. Чтобы в общем случае обеспечить отсутствие вычислительной неустойчивости в рассматриваемом смысле[1], надо использовать понятие точки следствия.

Точкой следования (англ. sequence point) называется такое фиксируемое программистом состояние вычислений, когда

[1] Одноименные проблемы, характерные для математических вычислений с плавающей точкой, имеют иную природу и здесь не рассматриваются.

все предыдущие операции гарантированно выполнены и все их результаты (включая побочные) уже проявили себя, но ни одна следующая операция еще не начата. Между точками следования ничего нельзя сказать о последовательности счета выражений, определять ее – задача блока оптимизации кода компилятора. Точками следования являются:

- завершенное точкой с запятой или запятой выражение;
- момент после вычисления первого операнда логических операций `&&` или `||` или проверяемого логического выражения трехместной операции;
- момент после завершения вычисления всех параметров, передаваемых функции, но до выполнения первой операции внутри функции;
- момент после производимой оператором `return` передачи возвращаемого функцией объекта наружу до первой операции вне тела функции.

К точкам следования не относятся, в частности:

- любые операции присваивания;
- проверяемые логические выражения в операторах цикла и в условном операторе `if`. Действительно, эти выражения могут быть составными, объединяемыми при помощи логических операций `&&` или `||`. Оптимизирующий компилятор стремится выстроить вычисления так, чтобы программа «узнала» результат как можно скорее. При этом иногда оказывается возможным не вычислять все выражение полностью. Если в этих, иногда не вычисляемых частях выражения предусматривается изменение переменных, оно может быть выполнено, а может и нет в зависимости от значения другой части составного логического выражения;
- отдельные передаваемые в функцию фактические параметры. Они могут быть выражениями, меняющими значения внешних по отношению к функции переменных. В отношении того, в какой последовательности они будут вычислены и каковы будут их побочные эффекты, нельзя строить никаких предположений.

Таким образом, список проблемных ситуаций, в которых может возникнуть неопределенность, гораздо больше, чем часто предполагают. Это позволяет предоставить компилятору максимальную свободу для оптимизации кода.

Например:

```
z[k]=k++ + a; /* непонятно, какое значение k берется слева
от присваивания, старое или новое */

if (mm++ && k--) {…}; /* mm++ или k-- может не быть вычис-
лен, если другой равен нулю */

my_func(x--,y++,x+y); /* не определен порядок вычисления
фактических параметров, вследствие чего в функцию могут
попасть различные варианты значений */
```

4.10. Трехместная операция

Единственная в своем роде трехместная операция не имеет устоявшегося названия. В литературе для нее используются также термины «условное присваивание», «присваивание с выбором», «операция выбора», «операция знак вопроса», «тернарная операция» и др. Общая форма записи такова:

```
логическое_выражение ? выражение_1 : выражение_2
```

Сначала проверяется значение `логического_выражения`. Если оно имеет значение «правда» (т.е. дает любое ненулевое целочисленное значение), вычисляется значение `выражения_1` и оно же является результирующим значением всей трехместной операции. Иначе вычисляется значение `выражения_2` и оно выступает значением трехместной операции. Эквивалентные конструкции:

```
переменная = логическое_выражение ?
выражение_1 : выражение_2
```

и

```
if (логическое_выражение)
  переменная = выражение_1;
else
переменная = выражение_2;
```

Легко убедиться, что оператор `if` предоставляет больше гибкости. Но трехместная операция, во-первых, за счет бо-

лее скромных возможностей обычно требует для своего выполнения меньше тактов процессора; во-вторых, она возвращает значение, не присваивая его какой-либо переменной, а храня его во временной ячейке быстрой кэш-памяти или в регистре процессора. Следовательно, это значение может быть гораздо эффективнее использовано в качестве операнда составного выражения.

Например, следующий фрагмент кода отыскивает минимальное из трех чисел:

```
min_xyz = (min_xyz = (x<=y ? x : y)) <= z ? min_xyz : z;
```

Сначала меньшее из `x` и `y` значение записывается временно в `min_xyz` (чтобы его не перевычислять дальше), затем оно сравнивается с `z` и вновь выбирается меньшее, которое попадает в `min_xyz` окончательно. Таким образом, внимательно следя за последовательностью вычислений, можно выстраивать целые цепочки из трехместных операций. Можно также размещать трехместную операцию в проверяемом выражении `if`, `switch`, операторов цикла и `return`.

Например:

```
while (3*a<b ? a-c : c) { … }
```

Многие программисты заключают проверяемое `логическое_выражение` в круглые скобки. Как правило, это необязательно, поскольку трехместная операция имеет очень низкий приоритет, что обеспечивает вычисление `логического_выражения` до того, как очередь дойдет до его проверки на истинность. Тем не менее практику обособления `логического_выражения` можно считать полезной. Во-первых, имеются операции с еще более низким приоритетом, которые могут появиться в проверяемом условии; во-вторых, круглые скобки улучшают читаемость конструкции. На эффективности кода это никак не сказывается. Правильность работы приведенной выше вложенной трехместной операции доказывает следующая программа, перебирающая все возможные случаи:

```
#include <stdio.h>

int main()
{
double x[]={1, 10, 200, 2000, 30000, 300000},
      y[]={2, 30, 100, 3000, 10000, 200000},
      z[]={3, 20, 300, 1000, 20000, 100000}, min_xyz;
unsigned char i;

for (i=0; i<sizeof(x)/sizeof(double); i++)
  {
  min_xyz = (min_xyz = (x[i]<=y[i] ? x[i] : y[i]))
    <= z[i] ? min_xyz : z[i];
  printf("\n min(x=%g,y=%g,z=%g)=%g",
    x[i],y[i],z[i],min_xyz);
  }
return 0;
}
```

4.11. Операция `sizeof`

При выделении памяти для структур и массивов, а также при нетривиальных преобразованиях типа требуется знать, сколько точно места занимает тот или иной объект. Ответ на этот вопрос дает значение одноместной операции `sizeof`, которую можно использовать тремя способами:

1) `sizeof(имя_объекта)`
2) `sizeof(имя_типа)`
3) `sizeof(выражение)`

Во всех случаях значение такого выражения — это размер объекта заданного типа или результата вычисления выражения, выраженный *в байтах*.

Например:

```
int a[100], b=17; double c=-123.5;
…
printf("В массиве a содержится %u элементов",
  sizeof(a)/sizeof(int));
printf("Результат вычисления выражения c/b занимает\
  байт %u",sizeof(c/b));
```

4.12. Операция явного преобразования типа

Имеющаяся в Си последовательность преобразований типов по умолчанию построена по правилу «от типа с меньшими возможностями к типу с бóльшими возможностями». При этом гарантируется, что значение не испортится из-за переполнения или потери точности. При попытке осуществить преобразование в обратную сторону компилятор, возможно, выдаст предупреждение о возможных негативных последствиях такого преобразования или даже сообщение об ошибке. Если программист четко понимает, что он хочет сделать и что данный фрагмент кода в действительности не таит угроз для работоспособности и точности работы программы, он должен уметь «умиротворить» компилятор. В этом состоит назначение двухместной операции явного преобразования (или, как говорят, *приведения*, англ. explicit type conversion, type cast) типа. Значение правого операнда «насильно» преобразуется к желаемому типу, заданному левым операндом в круглых скобках:

```
(желаемый_тип) выражение_другого_типа
```

При этом компилятор старается сохранить преобразуемое числовое значение, а не его двоичное представление.

Предположим, программист точно знает, что в переменной `a` типа `unsigned long long` в данном фрагменте кода может находиться только столь малое число, что его во всех случаях безопасно записать в переменную `b` типа `unsigned char`. Тогда правомерна запись:

```
b=(unsigned char)a;
```

Явное преобразование типа находит особенно важное применение в операциях с указателями. Поскольку всякий указатель характеризуется не только адресом, который он содержит, но и типом данного, с которым он связан, иногда бывает необходимо этот тип поменять. Так, функция выделения динамической памяти «не знает», для какой узкой цели ее вызывает программист, и всегда возвращает указатель на пустой тип: `void*`. Чтобы присвоить возвращенный ею нейтральный указатель `void*` переменной-указателю на конкретный тип и не

вызвать предупреждение компилятора, применяется явное преобразование.

Например:

```
long double *pa;
unsigned short size=100;
pa=(long double*)malloc(size*sizeof(long double));
```

В некоторых реализациях приведение `(void*)` ↔ `(тип*)` — вообще единственно допустимое преобразование указателя, поскольку существуют компьютеры, в которых указатели на разные типы хранятся по-разному. Причин прибегать к иным преобразованиям типов указателей очень мало. Если все-таки пришлось столкнуться с такой редкой ситуацией и не удается избежать преобразования `(тип_1*)` ↔ `(тип_2*)`, следует применять двухступенчатое преобразование

```
(тип_2*)((void*)указатель_на_тип_1)
```

в совокупности с тщательной проверкой на корректность работы кода на всех типах процессоров, для которых предназначена программа.

Таким образом, всякое правильное преобразование типа призвано в максимальной степени сохранить значение операнда при изменении типа его носителя.

4.13. Порядок выполнения операций

Относительная последовательность выполнения операций описывается содержимым табл. 4.6. Левая колонка содержит модальность — число от 0 до 14, характеризующее порядок выполнения операции в составном выражении: чем *меньше* модальность, тем *выше* приоритет операции.

Чтобы легче запомнить порядок применения операций с одинаковым приоритетом, в последнем столбце таблицы три класса с вычислением справа налево выделены крупной стрелкой. В программистском фольклоре предлагается сле-

дующее мнемоническое правило для запоминания этих операций: «Операции с нечетным числом операндов, а также с присваиванием вычисляются справа налево». Исключение составляют одноместные увеличение и уменьшение на единицу в суффиксной форме (операнд перед знаком операции), но их по-другому применить довольно затруднительно.

Таблица 4.6. Последовательность выполнения операций

Модальность	Общее описание	Обозначение	Пример выражений	Порядок обработки в составном выражении

0	Двухместные (2 операнда), первичное обращение (к выражению, функции, элементу массива или структуры)	`()`
		`[]`
		`->`
		`.`
			`f(u)`
			`g[t]`
			`k->x`
			`c.x`
				→

	Одноместные (1 операнд) в суффиксной форме (операнд перед знаком операции)	`++` `--`	`k++` `h--`	

1	Одноместные (1 операнд)	`(тип)`
		`!`
		`~`
		`-`
		`*`
		`&`
		`sizeof()`
			`(double)w`
			`!z`
			`~b`
			`-g`
			`*p`
			`&a`
			`sizeof(r+a)`
				←

	префиксная форма (операнд после знака операции)	`++` `--`	`++v` `--y`	

2	Двухместные (2 операнда), мультипликативные (умножение и деление)	`*`
		`/`
		`%`
			`a*d`
			`e/g`
			`j%i`
				→

3	Двухместные (2 операнда), аддитивные (сложение и вычитание)	`+` `-`	`q+z` `y-6`	→

Продолжение табл. 4.6

Модальность	Общее описание	Обозначение	Пример выражений	Порядок обработки в составном выражении
4	Двухместные (2 операнда), двоичные сдвиги	`<<` `>>`	`h<<n` `h>>n`	→

5	Двухместные (2 операнда), сравнения	`<`
		`>`
		`<=`
		`>=`
			`d<e`
			`d>e`
			`d<=e`
			`d>=e`
				→

6	Двухместные (2 операнда), проверки на [не]равенство	`==` `!=`	`p==g` `p!=g`	→

7	Двухместная (2 операнда), побитовое «И», таблица истинности:
	`0 & 0 → 0`
	`0 & 1 → 0`
	`1 & 0 → 0`
	`1 & 1 → 1`
		`&`	`v&0xFF00`
			`v&mask`
				→

8	Двухместная (2 операнда), побитовое «исключающее ИЛИ» (тест на несовпадение), таблица истинности:
	`0 ^ 0 → 0`
	`0 ^ 1 → 1`
	`1 ^ 0 → 1`
	`1 ^ 1 → 0`
		`^`	`a^b`	→

9	Двухместная (2 операнда), побитовое «ИЛИ», таблица истинности:
	`0 | 0 → 0`
	`0 | 1 → 1`
	`1 | 0 → 1`
	`1 | 1 → 1`
		`|`	`c|t`	→

Окончание табл. 4.6

Модальность	Общее описание	Обозначение	Пример выражений	Порядок обработки в составном выражении
10	Двухместная (2 операнда), логическое «И», таблица истинности: *«ложь» && «ложь» → «ложь»* *«ложь» && «правда» → «ложь»* *«правда» && «ложь» → «ложь»* *«правда» && «правда» → «правда»*	`&&`	`m&&n`	→
11	Двухместная (2 операнда), логическое «ИЛИ», таблица истинности: *«ложь» || «ложь» → «ложь»* *«ложь» || «правда» → «правда»* *«правда» || «ложь» → «правда»* *«правда» || «правда» → «правда»*	`||`	`k||m`	→

12	Единственная трехместная (3 операнда), возврат одного из двух значений в зависимости от выполнения условия	`?:`	`n=h>j ? c : s`
			
			`n=(h>j) ? c : s`
				←

13	Двухместные (2 операнда), присваивания (с арифметическими и побитовыми операциями)	`=`
		`+=`
		`-=`
		`*=`
		`/=`
		`%=`
		`&=`
		`^=`
		`|=`
		`<<=`
		`>>=`
			`y=x`
			`c+=h`
			`mn-=4`
			`ur*=9`
			`zx/=e`
			`kg%=u`
			`t&=m`
			`yt^=0xF`
			`oi|=7`
			`l<<=m`
			`e>>=k`
				←

14	Двухместная (2 операнда), операция перечисления выражений (запятая)	`,`	`a=9, g--, ++k`	→

Порядок вычисления операций в выражении крайне желательно помнить наизусть. В противном случае программист станет терзаться вопросами типа «Как понимать `if (*p&&p++)`?» или «Как компилятор будет реагировать на выражение `b=++a++` ?» и будет вынужден использовать столько круглых скобок, что исходный текст программы станет неудобочитаемым.

4.14. Сложные выражения. Тип выражения

Результаты выполнения операций могут выступать в свою очередь как операнды других операций. Максимальная вложенность таких конструкций неограниченна. Требуемая последовательность операций достигается посредством круглых скобок.

Например:

а)
```
if (x[i]==x_selected &&
    y[i]<=y_threshold) x[i]+=y[i];
```

б)
```
*(&a[0][0]+i+j*I)=-b[J-j][I-i];
```

В примере *а* два сложных выражения. В первом результат индексации в каждом из массивов сравнивается с некоторой величиной и результат сравнения участвует в логическом выражении `&&`. Во втором сложном выражении примера *а* из массива `y` извлекается `i`-й элемент и из массива `x` — тоже `i`-й элемент. К последнему прибавляется первый, и все это записывается обратно в `x[i]` (с возможным применением неявного преобразования типа `y[i]` к `x[i]`). В примере *б* вычисляется разность `J-j`, в массиве `b` берется подмассив с таким же индексом, считается `I-i`, и в получившемся подмассиве выделяется (`I-i`)-й элемент, у его копии меняется знак на противоположный. Значение справа от знака равенства готово. В массиве `a` отыскивается подмассив `a[0]`, в нем берется нулевой элемент, вычисляется указатель на него, `j` умножается на `I`, указатель `&a[0][0]` последовательно складывается с `i` и результатом `j*I`. Все это образует указатель, который разадресуется, и в это выражение со смыслом переменной записывается уже вычисленное выражение справа от знака присваивания (с возможным применением неявного преобразования типа).

При составлении сложных выражений нужно иметь в виду следующее:

- разумный компромисс между читаемостью исходного текста программы и эффективностью обычно достигается для выражений умеренной сложности (2–4-го уровня вложенности);
- такие выражения могут включать операции, меняющие значения переменных по ходу вычислений, что является распространенной причиной ошибок, особенно при повторном использовании этих переменных. Очень важно либо в точности знать приоритеты операций (что гораздо лучше), либо защищаться от незнания изрядным числом обеспечивающих нужный порядок действий круглых скобок «на всякий случай»;
- нельзя надеяться на возможности компилятора по выявлению ошибок в данной ситуации, поскольку в Си чрезвычайно велико число вариантов неправильных, но допустимых в языке выражений;
- чем сложнее выражение, тем вероятнее потребуются явные или неявные преобразования типов, дополнительно усложняющие понимание результата.

В случае одноместных операций тип результата выражения (говорят также «тип выражения») определяется операцией и типом единственного операнда. Существует множество сочетаний возможных типов операндов для двухместных и трехместной операций. Программист должен понимать тип результата всякого выражения хотя бы для того, чтобы сохранить его значение без существенных потерь информации.

4.15. Операция «запятая» и составные выражения

При помощи операции «запятая» из отдельных выражений можно составлять цепочку следующим образом:
`выражение_1 [, выражение_n]`^{n}

Например:

```
x=(x-2)*3, --k, 2*i;
```

Порядок вычисления слева направо и самый низкий приоритет выполнения влекут последовательное вычисление отдельных выражений в привычном порядке. Результирующим для составного выражения является значение последнего выражения в цепочке. От всех промежуточных выражений останутся только побочные результаты их выполнения, связанные с явным изменением переменных.

Например, в результате выполнения следующего фрагмента кода:

```
int i=10, k=10, b=2, c=-2, x, a[1000];
for (x=1000; x>=0; x--) a[i]=x;
x = (--i, 2*k, k++, k-=b*c, (a[i]=a[i+1])>=a[k] ?
a[k] : a[i+i]);
```

в переменных возникают такие значения: `i=9, k=15, a[9]=a[10]=10; x=a[18]=18`. Операция `2*k` прошла «бесследно»: ее значение нигде не было сохранено.

Очень важно отличать операцию «запятая» от запятой-разделителя:

- ◊ имен переменных и констант в определениях и декларациях;
- ◊ инициализирующих значений, присваиваемых элементам массивов или полям структур списком;
- ◊ фактических или формальных параметров при их перечислении в заголовке или прототипе функции.

В последних случаях нельзя строить никаких предположений относительно порядка вычисления значений присваиваемых выражений или фактических параметров функции либо насчет последовательности их передачи в функцию.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Выразите операцию ^ через операции & и ~.
2. Выразите операцию ^ через операции | и ~. Сравните с решением предыдущей задачи.
3. Всегда ли можно обойтись без операции «запятая»?
4. Реализуйте следующий фрагмент кода исключительно силами трехместной операции:

```
if (x<a) {if (x<b) y=1; else y=2; }
else {if (x<c) y=3; else y=4; }
```

5. В каких случаях необходимо явное приведение типа переменной?
6. Как значения двух целочисленных переменных поменять местами без привлечения временного хранилища и операции `^`? Применим ли этот способ к произвольным переменным с плавающей точкой?
7. Какие операции, кроме операции `^`, требуются для осуществления обмена заведомо маленькими значениями ($0 \le x, y \le 255$) между двумя переменными `x` и `y` типа `unsigned long`? Реализуйте и проверьте.
8. С помощью какой операции удобнее всего реализовать целочисленную беззнаковую арифметику по модулю `M`? Реализуйте функции сложения, вычитания, умножения в такой арифметике.
9. Продемонстрируйте на программном примере зависимость результата `a>>b` от знаковости целого `a`.
10. Какого рода действие производится посредством выражения `((a<<b)>>(b+c))<<c`, где `a`, `b`, `c` — целые? Зависит ли результат от знаковости `a`? В этом случае: `((a>>b)<<(b+c))>>c`?
11. При помощи двоичной маски извлеките и распечатайте экспоненту из числа типа `float`.

ГЛАВА 5 ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ

5.1. Структура как определяемый пользователем тип данных

Рассмотренные выше типы данных являются встроенными. При всем их многообразии и возможности объединять одинаковые элементы информации в массивы ограничением является то, что они довольно узкоспециализированы. В практических приложениях часто требуется в один элемент данных объединять представителей самых разных типов. Так, было бы удобно, чтобы к многомерному массиву каким-то образом «прикреплялись» его размерности для передачи в функцию как единого целого. Или еще более сложная задача: при написании базы данных по автомобилям требуется, чтобы каждый автомобиль имел свой «персональный» блок описаний, включающий тип кузова, марку, модель, мощность и литраж двигателя, вид топлива, год выпуска и т.д. Создавать общие массивы этих параметров для множества автомобилей и работать с ними крайне неудобно, поскольку отсутствует явно выраженное выделение каждого автомобиля. Требуется гибкое средство объединения разнородной информации «под одной крышей». Именно для решения такого рода проблем в языке Си программисту дается возможность вводить свои типы данных, адаптированные под конкретную задачу. Главным средством их конструирования является *структура*. В общем виде ее написание таково:

```
struct [имя_структурного_типа]
{
{тип_поля имя_поля;}n
} [список_переменных_данного_структурного_типа];
```

Все соответствующие имена подчиняются общим правилам для переменных и констант в Си. Устройство `списка_переменных_данного_структурного_типа` точно такое же, как у всякого списка вновь вводимых переменных и констант для встроенного стандартного типа, где они перечисляются через запятую. По меньшей мере должен обязательно присутствовать один из двух необязательных элементов `список_переменных_данного_структурного_типа` и `имя_структурного_типа`.

Если указано только `имя_структурного_типа`, конструкция имеет смысл чистой декларации: утверждается, что появился новый тип данных, именуемый

```
struct имя_структурного_типа
```

и сообщается, как он устроен. Особенностью названия типа является то, что оно состоит из двух слов, первым из которых обязательно выступает ключевое слово `struct`. Декларация открывает возможность в дальнейшем создавать конкретные переменные, массивы, указатели и вообще пользоваться типом точно так же, как если бы данный структурный тип был встроенный.

При отсутствии в написании структуры `имени_структурного_типа` вновь создаваемый тип остается безымянным, но зато создаются какие-то объекты (сами структуры, массивы структур, указатели на структуры и т.д.) данного типа. В таком случае это чистое определение. Впоследствии создать дополнительно некоторое число объектов этого структурного типа нельзя, поскольку у него отсутствует имя.

Наличие `списка_переменных_данного_структурного_типа` и `имени_структурного_типа` одновременно решает обе задачи: структурному типу присваивается имя и создаются связанные с ним объекты.

Пример чистой декларации дан ниже:

```
struct double_array {double *array;
  unsigned number_of_dimensions; unsigned *dimensions; };
```

Здесь определены три элемента данных (как говорят, *поля*), объединенные в структуру: *1*) указатель на начало массива; *2*) размерность массива (число измерений); *3*) вектор размеров массива вдоль каждого измерения. Начиная с места, где разме-

щена декларация, компилятор «знает» о существовании этого типа данных. Завести конкретный экземпляр такой структуры `my_array` и указатель на структуру `pointer_to_an_array` можно с помощью соответствующего определения:

```
struct double_array my_array, *pointer_to_an_array;
```

Например, объединенные декларация и определение:

```
struct double_array
{
double *array;
unsigned number_of_dimensions;
unsigned *dimensions;
} my_array, *pointer_to_an_array;
```

5.2. Доступ к полям структуры

Чтобы оперировать полями структуры, используется операция «точка», размещаемая между именем структуры и именем ее поля.

Например:

```
double a[2][3][4][5], *b;
unsigned da[4], *db, dn;
…
my_array.array=&a[0][0][0][0];
my_array.number_of_dimensions=4;
my_array.dimensions=&da[0];
da[0]=2; da[1]=3;
my_array.dimensions[2]=4; my_array.dimensions[3]=5;
```

В результате выполнения этого фрагмента программы все поля структуры `my_array` оказываются заполненными: поле `array` указывает на начало автоматического массива `a`, а поле `number_of_dimensions` содержит его размерность. Поле `dimensions` содержит указатель на автоматически выделенный массив из 4 указателей. Поскольку фактические размеры массива (2, 3, 4, 5) записываются по указателю, а `my_array.dimensions` и `&da[0]` адресуют одну и ту же область памяти, все равно, по какому из указателей инициализи-

ровать массив: `&da[0]` или `my_array.dimensions`. Можно, как обычно, взять начальный адрес объекта и присвоить его значение переменной-указателю подходящего типа:

```
pointer_to_an_array=&my_array;
```

Чтобы по указателю на структуру получить доступ к ее полям, используется операция «стрелка вправо».

Например:

```
b = pointer_to_an_array->array;
nb = pointer_to_an_array->number_of_dimensions;
db = pointer_to_an_array->dimensions;
```

Теперь вследствие цепочки присваиваний имеют место равенства:

```
b==&a[0][0][0][0], nb==4, db==&da[0].
```

Поля, доступ к которым обеспечивается операциями «точка» и «стрелка вправо», могут быть как L-выражениями, так и R-выражениями, т.е. находиться по обе стороны знака «равно» в операции присваивания.

При оптимизации кода, связанного с доступом к полям структур, следует иметь в виду вполне очевидные эквивалентные преобразования:

```
указатель_на_структуру->имя_поля ≡ (*указатель_на_струк-
туру).имя_поля
```

и

```
имя_структуры.имя_поля ≡ (&имя_структуры)->имя_поля
```

5.3. Вложенные структуры

Внутри структуры можно разместить не только поля встроенных типов, но и другие структуры при условии, что типы этих внутренних структур декларированы на момент декларации нового (как говорят, производного) структурного типа. Для примера определим тип структуры для хранения многомерного массива и его текстового описания, опираясь на

уже введенный тип структуры – носителя многомерного массива действительных чисел:

```
struct annotated_double_array
{
struct double_array da;
char *annotation;
} ada, adaa[2], *pada;
```

Первое поле вновь образованного структурного типа – это структура типа `struct double_array` целиком, а второе – указатель на строку описания. При этом в декларации без особой необходимости можно не оговаривать размерность `da` и длину описания `annotation`. Обычно это задают при создании конкретных экземпляров структуры, что увеличивает гибкость программы. В нашем примере последняя декларация заодно служит определением одиночной структуры `ada`, массива `adaa` из двух структур и указателя `pada` на структуру. Компилятор гарантированно решает вопросы, связанные с выделением памяти под все поля новоиспеченных структур `ada`, `adaa[0]` и `adaa[1]`, но не выполняет никакого выделения пространства для хранения самих тел многомерных массивов и их описаний, так как их размеры неизвестны, а тип – всего лишь указатель. Малая память под сами указатели `array`, `dimensions` и `annotation`, участвующие в новых структурах, выделена, но содержит «мусор», память по этим указателям не отпущена. Это должен сделать программист в процессе инициализации каждой из созданных структур индивидуально.

Например:

```
char t1[]="точки четырехмерного евклидова пространства";
char t2[]="распределение комаров в саду";
char t3[]="координаты объектов виртуальной реальности";
…
ada.annotation=t1;
adaa[0].annotation =&t2[0];
pada->annotation=t3;
```

Теперь с описаниями все в порядке, и ими можно пользоваться:

```
printf("%s, %s", adaa[0].annotation,
pada->annotation);
```

Чтобы инициализировать поля вложенной структуры, сначала при помощи операций «точка» и «стрелка вправо» внутри внешней структуры можно адресовать вложенную структуру, а в ней — нужные поля.

Например:

```
ada.da.array=&a[0][0][0][0];
…
adaa[0].da.number_of_dimensions=3;
…
pada->da.dimensions=
  (unsigned*)malloc(3*sizeof(unsigned));
pada->da.dimensions[0]=2;
pada->da.dimensions[1]=10;
pada->da.dimensions[2]=1000;
```

Таким образом, операции доступа к полям структуры применяются слева направо последовательно начиная с самой внешней структуры. Если используется динамическое выделение памяти, как в приведенном фрагменте кода, следует освобождать память, когда структура перестает быть нужной. Иначе выделенная, но не освобожденная память накапливается и остается заблокированной, но не используется. Это явление называется *утечкой памяти* (англ. memory leak) и потенциально способно привести к краху программы и проблемам общесистемного уровня.

5.4. Структуры и указатели

Структуры можно объединять в массивы, подобно данным встроенных типов. Наряду с обращением посредством квадратных скобок для доступа используется стандартная адресная арифметика.

Например:

```
for (i=0;i<2;++i)
```

```
for (k=0;k<4;++k)
  *(((adaa+i)->da.dimensions)+k)=i+1;
```

В то же время полями структуры могут быть массивы и указатели, в том числе на структуры других, определенных к данному моменту типов. Это очень удобно для адресации одной структуры из другой. Структуры можно связывать, отражая одно-, двух- и многосторонние информационные связи.

Если бы была возможность ввести такой тип структуры, который мог бы содержать указатели на структуры того же самого типа, это позволило бы создавать весьма сложные и эффективные хранилища данных с практически любыми связями между структурами, например перекрестными. Каждый объект мог бы нести информацию не только о себе самом, но мог бы указывать на соседние и другие каким-то образом связанные с данным однородные объекты. Такие хранилища данных описываются в математике на языке теории графов и имеют множество практических применений.

Но в Си определяемая структура или ее определяемый тип становятся доступны для использования лишь к моменту завершения определения, а до этого момента их нельзя использовать. В том случае, если внутри структуры некоторого типа требуется поместить указатель на структуру того же типа, например при создании цепочки ссылающихся друг на друга структур, в целях определения структуры разрешается пользоваться указателем на нее еще до завершения самого определения.

Например:

```
struct x {
int a;
struct x *previous;
struct x *next; } q;
```

Такая возможность существует, так как, хотя размер структуры не ясен до завершения декларации ее типа, указатель на структуру занимает постоянный размер, не зависящий от типа самой структуры. Более того, во многих системах размер и конструкция указателя (кроме указателя на символьный тип

char*, который для ускоренной работы со строками бывает устроен иначе) не зависят от того, на какой объект он указывает, но во всех случаях гарантировать это невозможно.

Для организации перекрестных ссылок полей структур различного типа друг на друга сначала надо дать короткую «декларацию существования» типов структур, чтобы дальше можно было на них ссылаться, а позже осуществить детальное определение соответствующих структур.

Например:

```
/* пустая "декларация существования" */
struct y; struct z;

struct x /* декларация типа + определение */
  {
  double m;
  struct x *link_to_another_x;
  struct y *link_to_y;
  struct z *link_to_z;
  } x1, x2[3], *px;

struct y /* декларация типа */
  {int n;
  struct x *link_to_x;
  struct y *link_to_another_y;
  struct z *link_to_z;
  };

struct y y1; /* определение */

struct z
  {
  long *p;
  struct x *link_to_x;
  struct y *link_to_y;
  struct z *link_to_another_z;
  } z1[5]; /* декларация типа + определение */
```

Данные три декларации обладают свойством трансцендентности: каждая из таких структур ссылается на два других типа. Перекрестные ссылки устанавливаются после создания конкретных экземпляров структур.

Например:

```
x1.link_to_another_x=&x2[2];
x2[2].link_to_y=&y1;
px=(struct x*)malloc(sizeof(struct x));
px->link_to_z=z1+4;
```

Можно сказать, что язык Си предоставляет практически неограниченный и очень эффективный инструментарий для создания гибких систем хранения информации.

5.5. Структуры и typedef

Обязательное добавление слова struct к имени структуры призвано напоминать программисту об оперировании объектами составного пользовательского типа. Иногда программисты ради лаконичности исходного текста предпочитают отказаться от таких напоминаний. В этом неоценима помощь директивы объявления альтернативного имени типа typedef. Новое одиночное имя структурного типа вводится для уже декларированного «длинного» имени типа в общем случае так:

```
typedef struct
имя_структурного_типа_комбинируемое_со_словом_struct
новое_эквивалентное_имя_структурного_типа;
```

Его введение можно осуществить и сразу в момент декларации нового типа:

```
typedef struct
имя_структурного_типа_комбинируемое_со_словом_struct
  {
  {тип_поля имя_поля;}n
  } новое_эквивалентное_имя_структурного_типа;
```

Теперь определение конкретных представителей этого типа выглядит компактнее:

```
новое_эквивалентное_имя_структурного_типа
  список_имен_структур;
```

Например:

```
typedef struct struct_car
{
unsigned carbody_type; /* тип кузова */
char *make_of_vehicle; /* марка */
char *model; /* модель */
float engine_power; /* мощность двигателя */
float engine_capacity; /* литраж двигателя */
char fuel_type; /* вид топлива */
unsigned short year_of_manufacture; /* год выпуска */
} car; /* это имя типа, а не экземпляра структуры */

car TagAZ, Opel, Kia, Peugeot; /* имя экземпляра структуры
*/
```

5.6. Инициализация полей структуры

Массивы как простейший составной тип данных могут быть инициализированы непосредственно в месте их создания при помощи массивов значений, перечисляемых через запятую и заключенных в фигурные скобки. Сходный способ присваивания начальных значений применим и к структурам. Важно лишь, чтобы все инициализирующие величины:

- были определены и имели осмысленные значения к моменту инициализации (старые реализации средств разработки требуют также, чтобы они являлись константами или константными выражениями);
- перечислялись точно в таком порядке, как поля в декларации структуры, которым они должны быть присвоены.

Например (для структуры из § 5.5):

```
#define universal 4
car FamilyChevy={universal, "Chevrolet",
  "Wagon", 109, 1.6, 95, 2010};
```

В результате поля структуры `FamilyChevy` получат значения согласно табл. 5.1.

Таблица 5.1. Значения полей структуры `FamilyChevy`

Имя поля	Тип	Значение
`carbody_type`	`unsigned`	4
`make_of_vehicle`	`char*`	Указатель на константную символьную строку `"Chevrolet"`
`model`	`char*`	Указатель на константную символьную строку `"Wagon"`
`engine_power`	`float`	109
`engine_capacity`	`float`	1.6
`fuel_type`	`char`	95
`year_of_manufacture`	`unsigned short`	2010

Если структура включает другую структуру, для зрительного разделения их содержимого рекомендуется прибегать к структурной инициализации с выделением вложенных объектов при помощи пар фигурных скобок.

Например:

```
struct annotated_double_array
  ada={{&a[0][0][0][0], 4, &da[0]},
    "точки четырехмерного евклидова пространства"};
```

Результаты такой инициализации представлены в табл. 5.2.

Таблица 5.2. Значения полей после инициализации

Имя поля	Тип	Значение
`da.array`	`double*`	`&a[0][0][0][0]`
`da.number_of_dimensions`	`unsigned`	4
`da.dimensions`	`unsigned*`	`&da[0]`
`annotation`	`char*`	Указатель на константную символьную строку `"точки четырехмерного евклидова пространства"`

Компилятор умеет отличить название вложенной структуры `da` от одноименного массива, содержащего размеры масси-

ва, по контексту использования. Очевидно, при такой конструкции структуры выбранный способ инициализации не в состоянии присвоить значения элементам многомерного массива `a` и вектора размерностей `da`, поэтому это необходимо сделать отдельно. Пусть эти массивы «вмонтированы» в структуру целиком с явным указанием размерностей.

Например:

```
struct double_array {double array[2][3];
  unsigned number_of_dimensions; unsigned dimensions[2];
};

struct annotated_double_array
{
struct double_array da;
char annotation[];
} ada;
```

Тогда полная инициализация была бы возможна ценой утраты гибкости.

Например:

```
struct annotated_double_array
  ada={{{{1.3, -0.9, 3.9},{8.3, -2.9, 5.4}},
  2, {2, 3}},
  "точки четырехмерного евклидова пространства"};
```

Вложенные фигурные скобки отражают структурный характер инициализации. Использование структурной инициализации структур[1] рекомендуется всегда, поскольку:

- становится легче читать и сложнее потерять счет полям;
- она позволяет более гибко осуществлять неполную инициализацию, при которой вложенный блок данных определяется не до конца.

В следующем примере последнему элементу многомерного массива `ada.da.array[1][2]` ничего не присваивается и это не сказывается на возможности инициализировать последующие поля, так как разметка при помощи фигурных скобок позволяет компилятору не потерять привязку к полям структуры.

[1] Это не тавтология, а термин.

Например:

```
struct annotated_double_array
  ada={{{{1.3, -0.9, 3.9},{8.3, -2.9}}, 2, {2, 3}},
    "точки четырехмерного евклидова пространства"};
```

Из соображений преемственности в современном Си сохранена неструктурная инициализация, когда все поля структур, в том числе вложенных, инициализируются последовательно в том порядке, в котором эти поля размещены в памяти. Бóльшая часть обсуждавшегося представлена в данном ниже полном примере, где также рассматриваются аспекты динамического выделения памяти для сложных составных структур.

Пример:

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

/* вводится новый тип struct double_array */
struct double_array {
  double *array;
  unsigned number_of_dimensions;
  unsigned *dimensions; };

/* вводится производный тип
struct annotated_double_array */
struct annotated_double_array
  {
  struct double_array da;
  char *annotation;
  };

int main()
{
double a[2][3][4][5], *b;
unsigned da[4], *db, dn;
/*  определение структуры my_array и указателя
pointer_to_an_array на структуру */
struct double_array my_array, *pointer_to_an_array;

/* инициализация структуры my_array */
```

```
for (dn=0;dn<2*3*4*5;dn++) *(&a[0][0][0][0]+dn)=0.1*dn;
my_array.array=&a[0][0][0][0];
my_array.number_of_dimensions=4;
my_array.dimensions=&da[0];
da[0]=2; da[1]=3;
my_array.dimensions[2]=4; my_array.dimensions[3]=5;

/* инициализация указателя pointer_to_an_array на струк-
туру */
pointer_to_an_array=&my_array;

/* доступ к полям структуры по указателю */
b=pointer_to_an_array->array;
dn=pointer_to_an_array->number_of_dimensions;
db=pointer_to_an_array->dimensions;

printf("\n a[0][0][0][0]=%g,...,a[1][2][3][4]=%g",
  *b,*(b+2*3*4*5-1));
printf("\n number_of_dimensions=%u",dn);
printf("\n dimensions={%u,%u,%u,%u}\n",
  db[0],db[1],db[2],db[3]);

  {
   /* инициализация и вывод автоматического экземпляра
структуры */
  struct annotated_double_array ada={
    {&a[0][0][0][0], 4, &da[0]},
    "точки четырехмерного евклидова пространства"};

  printf("\n a[0][0][0][0]=%g,...,a[1][2][3][4]=%g",
    *(ada.da.array),*(ada.da.array+2*3*4*5-1));
  printf("\n number_of_dimensions=%u",
    ada.da.number_of_dimensions);
  printf("\n dimensions={%u,%u,%u,%u}",
    ada.da.dimensions[0],ada.da.dimensions[1],
    ada.da.dimensions[2],ada.da.dimensions[3]);
  printf("\n annotation=\"%s\"\n",ada.annotation);
  }

  {
   /* работа с полностью динамически выделяемой сложной
структурой */
  struct annotated_double_array *pada;
  char t[]="распределение комаров в саду";
```

```
  if (pada=(struct annotated_double_array *)
     malloc(sizeof(struct annotated_double_array)))
    {
    if (pada->da.array=
       (double*)malloc(2*3*sizeof(double)))
      {
      unsigned i;
      for (i=0;i<2*3;++i)
        *(pada->da.array+i)=0.1*i;

      if (pada->da.dimensions=
         (unsigned*)malloc(2*sizeof(unsigned)))
        {
        pada->da.dimensions[0]=2;
        pada->da.dimensions[1]=3;
        pada->da.number_of_dimensions=2;
        if (pada->annotation=
           (char*)malloc(strlen(t)*sizeof(char)))
          {
          strcpy(pada->annotation,t);

          printf("\n a[0][0]=%g,...,a[1][2]=%g",
            *(pada->da.array),*(pada->da.array+2*3-1));
          printf("\n number_of_dimensions=%u",
            pada->da.number_of_dimensions);
          printf("\n dimensions={%u,%u}",
            pada->da.dimensions[0],
            pada->da.dimensions[1]);
          printf("\n annotation=\"%s\"\n",
            pada->annotation);
          free(pada->annotation);
          free(pada->da.dimensions);
          free(pada->da.array); free(pada);
          }
        else
          {
          printf("\n Невозможно выделить память для \
pada->annotation");
          free(pada->da.dimensions);
          free(pada->da.array); free(pada);
          return 3;
          }
        }
      else
```

```
        {
        printf("\n Невозможно выделить память для \
pada->da.dimensions");
        free(pada->da.array); free(pada);
        return 2;
        }
      }
    else
      {
      printf("\n Невозможно выделить память для \
pada->da.array");
      free(pada);
      return 1;
      }
    }
  else printf("\n Невозможно выделить память для \
pada");
  }

return 0;
}
```

5.7. Выравнивание полей структуры

Выражение

```
sizeof(имя_структурного_типа)
```

или

```
sizeof(имя_конкретной_структуры)
```

или

```
sizeof
(новое_имя_структурного_типа_с_применением_typedef)
```

дает выраженный в байтах размер структуры. Можно предположить, что этот размер равен сумме размеров всех полей структуры, но это далеко не всегда так. Дело в том, что процессору компьютера не одинаково «удобно» адресовать элементы данных, расположенные по различным адресам. Это связано с тем, что процессор «шагает» по расположенным в оперативной

памяти данным с шагом, равным его стандартной длине слова. Например, 64-разрядный процессор быстрее всего считывает или записывает данные, начальные адреса которых делятся без остатка на восемь (байт). Такие кратные размеру машинного слова адреса называются *выравненными* (англ. aligned).

Процессор в состоянии обратиться и к элементу информации, начинающемуся с другого адреса (однако адрес в байтовом выражении все равно обязательно должен быть целым). Для этого считывается элемент данных большего размера, начинающийся с выравненного адреса, и из него выделяется нужный элемент данных, что требует некоторого числа дополнительных операций. В большинстве программ происходит интенсивное обращение к ОЗУ и накладные расходы на доступ к памяти существенным образом сказываются на общем объеме вычислений. Поэтому начала элементов данных стремятся выравнивать в адресном пространстве. Так как переменные, очевидно, должны занимать непересекающиеся области в памяти, возникают «дыры» – пустые фрагменты памяти на стыках смежных элементов данных, наличием которых объясняется избыточная потребность в памяти.

Та же проблема характерна для массивов из элементов малого размера, но поскольку предполагается, что размер массива велик, «дыры» в памяти повлекли бы очень большой перерасход памяти. К счастью, массивы устроены регулярно, и при адресации их элементов компиляторы пользуются специальными трюками, позволяющими не терять память. Другое дело – структуры, в которых размеры полей различны, и компилятору «трудно приспособиться» ко всему возможному разнообразию структур. Для структур применение выравнивания полностью оправдано и неизбежно, откуда возникает обсуждаемое неравенство суммы длин полей и общего размера структуры. Современные компиляторы позволяют реализовать различные значения выравнивания или вовсе отказаться от него ради экономии памяти.

Программисту иногда бывает необходимо знать, насколько данное поле структуры отстоит от ее начала. Конечно, можно было бы взять указатели на структуру и на ее интересующее поле, привести их к типу указателя на `char` и вычесть. Но это не изящ-

но, следует рассматривать вариант, когда `sizeof(char)>1`. Здесь на помощь может прийти специализированный макрос:

```
size_t offsetof (имя_структурного_типа,
имя_поля_в_структуре_этого_типа)
```

Возвращаемое выраженное в байтах значение имеет тип `size_t` – самого длинного беззнакового целого, которое доступно данной системе, и является константой. Если `имя_структурного_типа` вводилось без `typedef` и содержит ключевое слово `struct`, его не опускают в первом параметре макроса.

Например, правомочен такой вызов:

```
offset = offsetof(struct annotated_double_array,
   annotation);
```

Для иллюстрации приведем полный пример:

```
#include <stdio.h>

typedef struct { char c[10]; char a; long double b; } st;

int main()
{
struct st_st { char a; long double b; char c[10]; };

printf("\noffset(st.c)=%zd; offset(st.a)=%zd; \
offset(st.b)=%zd; ",
  offsetof(st, c),offsetof(st, a),offsetof(st, b));

printf("\noffset(st_st.a)=%zd; offset(st_st.b)=%zd; \
offset(st_st.c)=%zd; ",
  offsetof(struct st_st, a),
  offsetof(struct st_st, b),
  offsetof(struct st_st, c));

return 0;
}
```

Отсюда видно, что для выявления выравнивания полей в структуре необязательно определять хотя бы один ее экземпляр.

5.8. Объединения (смеси, союзы) union

Нехватка оперативной памяти – одна из основных проблем при работе с крупными блоками данных. Динамическое выделение памяти работает недостаточно быстро и требует от программиста повышенного внимания. Автоматическое выделение памяти лишено гибкости динамического, в частности практически отсутствует возможность изменить размеры массивов. Частичное решение состоит в делении функции на блоки и использовании того факта, что, выходя за пределы видимости, автоматические объекты удаляются.

Например, в следующем примере массив b окажется (по крайней мере частично) на месте массива a:

```
int func(void)
{
short x[100];
…
    {
    long a[1000];
    …
    }
    {
    double b[500];
    …
    }
return 0;
}
```

При этом массив x доступен в обоих блоках. Задавая границы блоков и их вложенность, можно довольно гибко управлять *временем жизни* автоматически выделенных структур данных. Но это решение проблемы не всегда удовлетворительное, так как выделение памяти на стеке не многократно превосходит по скорости динамическое. Кроме того, такой принцип разделения функции на блоки лишь из соображений экономии памяти искусственный и не обязательно отражает логику работы программы, а также ведет к плохой читаемости программы вследствие избыточного числа фигурных скобок. Поэтому в языке Си предусмотрена возможность размещения различных эле-

ментов данных на месте друг друга без перераспределения памяти. Одна и та же область памяти поочередно может быть представлена то как один элемент данных, то как другой, то как третий и т.д. Это реализуется посредством объединений (в русскоязычной программистской литературе используется несколько вариантов перевода английского слова union – смесь, союз). Синтаксис у объединений такой же, как у структур, но в него вкладывается иной смысл, поэтому ключевое слово `struct` заменено `union`:

```
union имя_объединения
{
{тип_поля имя_поля;}^n
} список_переменных_данного_типа;
```

При этом параметр $n \geq 2$. Объединение обеспечивает хранение всех перечисленных внутри фигурных скобок полей начиная с одного и того же адреса в памяти. Очевидно, наложение в разделяемой памяти обычно ведет к перетиранию прежде использовавшегося объекта новым. В то же время при повторном использовании пула памяти старый объект в нем является чаще всего «мусором». Поэтому программисту следует обращать повышенное внимание на контекст использования объединения и корректную инициализацию блоков данных. Полный пример программы, использующей объединения традиционным образом, приведен ниже:

```
#include <stdio.h>
#include <memory.h> /* или #include <string.h>, в зависи-
мости от системы программирования, для доступности функ-
ции memcpy */

#define N 2

int main()
{
union un_type
{
double da[N];
int ia[N<<1]; /* умножаем N на 2 */
char ca[N<<3]; /* умножаем N на 8 */
```

```
struct {double sd; int si; short ss; char sc1, sc2;} st;
} un, *pun;
int i;
union un_type un1, *pun1;

printf("\nsizeof(un)=%u\n", sizeof(un));

pun=&un; /* указатель на объединение */

/* сначала используем объединение для хранения массива
типа double */
for (i=0;i<N;i++) un.da[i]=0.033*i*i;
for (i=0;i<N;i++)
  printf("\n un.da[%d]=%g\t\tpun->da[%d]=%g",
    i,un.da[i],i,pun->da[i]);
printf("\n");

/* разместим здесь же массив типа int */
for (i=0;i<2*N;i++) un.ia[i]=100*i*i;
for (i=0;i<2*N;i++)
  printf("\n un.ia[%d]=%d\t\tpun->ia[%d]=%d",
    i,un.ia[i],i,pun->ia[i]);
printf("\n");

/* теперь здесь же размещается массив char */
for (i=0;i<8*N;i++) un.ca[i]=i<<2;
for (i=0;i<8*N;i++)
  printf("\n un.ca[%d]=%d\t\tpun->ca[%d]=%d",
    i,un.ca[i],i,pun->ca[i]);
printf("\n");

/* сейчас в этом же месте находится структура */
un.st.sd=0.12345678;
un.st.si=97531;
un.st.ss=12345;
un.st.sc1=123;
un.st.sc2=124;
printf("\nun.st.sd=%g  pun->st.sd=%g\
  un.st.si=%d  pun->st.si=%d",
  un.st.sd,
  pun->st.sd,
  un.st.si,
  pun->st.si);
printf("\nun.st.ss=%d  pun->st.ss=%d  un.st.sc1=%d\
```

```
  pun->st.sc1=%d  un.st.sc2=%d  pun->st.sc2=%d\n",
  un.st.ss,pun->st.ss,
  un.st.sc1,
  pun->st.sc1,
  un.st.sc2,
  pun->st.sc2);

/* объединение целиком копируется в un1 и оттуда распечатывается как структура */
memcpy(&un1,pun,sizeof(un)); pun1=&un1;
printf("\nun1.st.sd=%g  pun1->st.sd=%g\
  un1.st.si=%d  pun1->st.si=%d",
  un1.st.sd,
  pun1->st.sd,
  un1.st.si,pun1->st.si);
printf("\nun1.st.ss=%d  pun1->st.ss=%d\
  un1.st.sc1=%d  pun1->st.sc1=%d\
  un1.st.sc2=%d  pun1->st.sc2=%d\n",
  un1.st.ss,
  pun1->st.ss,
  un1.st.sc1,
  pun1->st.sc1,
  un1.st.sc2,
  pun1->st.sc2);

return 0;
}
```

В зависимости от способа обращения объединение ведет себя попеременно как различные объекты данных. Оно предназначено для наложения в памяти конкретных элементов данных, а не для создания новых типов, поэтому не имеет смысла применение конструкции `typedef` вместе с `union`. Размер объединения равен размеру наибольшей из структур данных, на которую он рассчитан, плюс, возможно, немного «порожней» памяти для выравнивания адресов. Подробно аспекты выравнивания адресов рассмотрены в § 5.7.

Возможно также нестандартное использование `union` с целью «мгновенного» массового преобразования типа в рамках всего массива. Такое преобразование типа является необычным, рискованным, но чрезвычайно эффективным и иногда очень полезным. При стандартном преобразовании компиля-

тор в рамках возможностей типа результирующей переменной старается сохранить числовое значение исходной переменной. В случае преобразования посредством `union` это в общем случае не так, но зато гарантируется сохранение двоичного представления числа.

В качестве полного примера приведем программу, в которой используется тот факт, что для некоторых вычислительных архитектур типы `int` и `float` имеют одинаковый размер, вдвое превышающий длину `unsigned short`:

```
#include <stdio.h>
#define N 3

int main()
{
union
  {
  float float_array[N];
  int int_array[N];
  unsigned short us_array[2*N];
  } conv;
int i;

if (sizeof(int)==sizeof(float) &&
    sizeof(int)==2*sizeof(unsigned short))
  {
  printf("\nИнициализация типом float:\n");
  for (i=0;i<N;i++)
    {
    conv.float_array[i]=0.1111+i;
    printf("conv.float_array[%d]=%g  ",
      i, conv.float_array[i]);
    }
  printf("\nЧтение этого же двоичного кода как \
int:\n");
  for (i=0;i<N;i++)
    printf("conv.int_array[%d]=%d  ",
      i, conv.int_array[i]);

  printf("\nЧтение этого же двоичного кода как \
unsigned short (шестнадцатеричный вывод):\n");
  for (i=0;i<2*N;i++)
```

```
    printf("conv.us_array[%d]=%x  ",
      i, conv.us_array[i]);

  printf("\nЧтение этого же двоичного кода как \
float:\n");
  for (i=0;i<N;i++)
    printf("conv.float_array[%d]=%g  ",
      i, conv.float_array[i]);
  printf("\n");
  return 0;
  }
else
  {
  printf("\nПреобразование невозможно, так как \
sizeof(int)!=sizeof(float) или \
sizeof(int)!=2*sizeof(unsigned short).");
  return 1;
  }
}
```

Приведенная выше программа демонстрирует «трюк» по интерпретации `float` как `int` или пары `unsigned short`. Подобные приемы полезны для хранения и передачи двоичного кода одного типа внутри переменных другого типа без потерь точности. Очевидно, размер переменной исходного типа не должен превосходить размер типа-приемника для сохранения соотношения 1:1. Этот же способ годится для распечатки переменных некоторого типа в несвойственном для этого типа формате. Достоинством является «бесплатность» такого преобразования, так как для его осуществления не генерируется практически никакого машинного кода. Чтобы написать корректный код, разработчик должен тщательно следить за соответствием размеров и знать особенности архитектуры компьютера.

Объединения выступают также естественным средством плотной групповой упаковки данных из массивов с элементами малого размера в массивы с бóльшим размером единицы хранения:

Например:

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

```
#define N 3

int main()
{
union
  {
  unsigned int ui_array[N];
  unsigned char uc_array[N*
    sizeof(unsigned int)/sizeof(unsigned char)];
  } stor;
int i;

if (sizeof(unsigned int)%sizeof(char)==0)
  {
  for (i=0;
  i<N*sizeof(unsigned int)/sizeof(unsigned char); i++)
    {
    if (!(i%
       (sizeof(unsigned int)/sizeof(unsigned char))))
      printf(" ");
    stor.uc_array[i]=i; /* инициализация */
    printf(" %x", stor.uc_array[i]);
    }
  printf("\n  ");
/* распечатка упакованных блоков данных из нескольких
unsigned char в виде unsigned int */
  for (i=0; i<N; i++)
    printf("%x  ", stor.ui_array[i]);
  printf("\n\n");
  return 0;
  }
printf("\nУпаковка невозможна, так как sizeof(\
unsigned int) не кратна sizeof(unsigned char).");
return 1;
}
```

Здесь группы (например, в некоторых системах четверки) элементов `unsigned char` размещаются внутри одного элемента `unsigned int`. Такое преобразование особенно выигрышно при кратных размерах элементов массивов, поскольку в отличие от упаковки при помощи сдвигов и масок или посредством битовых полей не возникает никаких дополнительных расходов вычислительных ресурсов. Используя такие не-

стандартные «трюки», программист должен быть предельно осторожен и включать в программу обработку всех возможных случаев. Получившийся код даже в этом случае скорее всего не будет подходить для всех возможных компьютеров, т.е. не будет *переносимым* (англ. portable).

5.9. Упаковка данных малой разрядности

Встроенные целочисленные типы имеют фиксированную разрядность. В некоторых приложениях, связанных, как правило, с обработкой потоков двоичных данных или растровых изображений, часто возникает потребность оперировать целыми числами со значениями из гораздо более узкого диапазона и меньшей разрядности, чем предоставляют стандартные типы. Конечно, использование предопределенных типов допустимо, но их битовая длина используется неэкономно. Более того, для контроля диапазона их значений требуется применение битовых масок, что требует дополнительных операций. Было бы полезно наличие более коротких целочисленных типов, эффективно размещенных в памяти. В качестве решения можно использовать плотную упаковку битовых последовательностей в стандартные типы при помощи операций сдвига и побитового «И». Так, 2-битное, 3-битное, два 5-битных слова `a`, `b`, `c`, `d` и вдобавок однобитный флаг `e` можно разместить в 16-битной переменной.

Например:

```
unsigned a=3, b=1, c=15, d=1, e=1, support;
…
support=((a & 0x3)<<14) | ((b & 0x7)<<11) |
  ((c & 0x1F)<<6) | ((d & 0x1F)<<1) | (e & 0x1);
```

Распаковка получается записью операций в обратном порядке.

Например:

```
e=support & 0x1;
d=(support>>1) & 0x1F; /* 1F16=11111₂ сдвиг 1 */
c=(support>?6) & 0x1F; /* 1F16=11111₂, сдвиг 6=1+5 */
```

```
b=(support>>11) & 0x7; /* 7₁₆=111₂, сдвиг 11=1+5+5 */
a=(support>>14) & 0x3; /* 3₁₆=11₂, сдвиг 14=1+5+5+3 */
```

Диапазон значений короткоразрядных беззнаковых упакованных целых простирается от 0 до 2^n-1, где n – число битов в упакованном формате. Такой программный способ упаковки применяется нередко, но не является единственным.

5.10. Битовые поля

В Си имеется более удобная (и порой даже более быстрая)[1] альтернатива рассмотренному в § 5.9 способу – использование особой разновидности структур, называемой битовыми полями (англ. bit fields). *Битовое поле* – это непрерывная последовательность соседних битов, размещенная внутри единого целого значения. Это слово-носитель может иметь произвольный целочисленный тип, зависящий от архитектуры компьютера и устройства компилятора. Однако во многих реализациях слово-носитель является стандартным машинным словом компьютера, т.е. имеет тип `unsigned` и повлиять на это нельзя. Битовые поля последовательно упаковываются в слова-носители, обычно в направлении от младших разрядов к старшим. Но конкретное поле не может перешагивать границу целочисленного слова-носителя. Следовательно, число бит в битовом поле может быть от одного до длины слова-носителя, выраженной в битах. Воспринимать ли индивидуальное битовое поле как знаковое или беззнаковое, определяет предваряющий это поле спецификатор, который может принимать одно из трех значений – `int`, `signed` или `unsigned` (в более старых реализациях – только `unsigned`).

Например:

```
struct bf
  {
   int a:2;
```

[1] Это зависит от компилятора и системы команд процессора. В приложениях, где очень важно быстродействие, следует сначала экспериментально выяснить, что́ быстрее работает: обращение к битовым полям или их программируемый вручную эквивалент с масками и сдвигами.

```
    unsigned b:3;
    int c:5;
    int d:1;
    unsigned e:5;
    } x, f2;
```

Данная структура обеспечивает размещение данных в двух байтах (в одном слове в случае процессора разрядностью не менее 16). Если бы последнее поле `e` было задано как `unsigned d:22`, то для 32-битного слова-носителя оно размещалось бы не в первом слове-носителе, а в разрядах 0–21 второго слова, так как «перетекание» битового поля из слова в слово принципиально невозможно из соображений скорости доступа. В полях типа `signed` крайний левый бит является знаковым. Синтаксис доступа к битовым полям тождествен таковому для обычных структур.

Например:

```
x.a=-1;
f2.b=5;
m=x.a+1;
```

При этом необходимо следить за соблюдением диапазона допустимых значений, которые «умещаются» в поле. Дело в том, что при переполнении битовых полей не всегда генерируются сообщения о выходе за границы. Отсутствие контроля связано с соображениями наивысшей эффективности исполнимого кода. Значения отдельных полей подчиняются правилам стандартной модульной арифметики по модулю 2^n, где n – длина поля в битах. Если не все биты записываемого в поле значения умещаются в поле, всегда размещается младшая часть числа, а старшие биты игнорируются. Риска выйти за границы и повредить содержимое соседних битовых полей нет: взятие двоичной маски в момент загрузки битового поля входит в обязанности компилятора. В отношении стандартных преобразований типов битовые поля эквивалентны переменным стандартных целочисленных типов `int` и `unsigned`.

Например:

```
printf("x.a=%d, f2.b=%u", x.a, f2.b);
```

Ниже дана общая формализованная запись структуры, наполненной битовыми полями:

```
struct [имя_структурного_типа]
{
{тип_поля имя_поля : длина_поля_в_битах;}^n
} [список_переменных_данного_структурного_типа];
```

Из двух необязательных элементов описания – имя_структурного_типа или список_переменных_данного_структурного_типа, как и в случае структур, по меньшей мере один должен присутствовать. Если отсутствует список конкретных переменных структурного типа, получается чистая декларация нового структурного типа, а если список есть – определение. Если структурному типу дано имя, его впоследствии можно использовать, во-первых, для создания новых переменных и констант этого типа:

```
struct имя_структурного_типа
список_переменных_данного_структурного_типа;
```

Например:

```
struct bf bf3, array_of_bf[100],
  *pointer_to_bf, *array_of_pointers_to_bf[10];
```

а во-вторых, для преобразования типа указателя:

```
переменная_указатель_на_структурный_тип=
(struct имя_структурного_типа*) указатель_другого_типа;
```

Например:

```
pointer_to_bf=(struct bf*)
  malloc(sizeof(struct bf)*1000);
```

Как видно из приведенных выражений, названием типа является сочетание struct и имя_структурного_типа. Только вместе они именуют определенный программистом тип.

Являясь средством автоматической упаковки-распаковки значений нескольких переменных в одно машинное слово, битовые поля реализуют компромисс между экономией памяти,

удобством использования и высоким быстродействием. Отсюда вытекают ограничения. Во-первых, поле не может быть массивом. Это связано с потенциальной проблемой нарушения однородного плотного размещения элементов на границах слов-носителей. Во-вторых, поля не имеют адресов, поскольку их начало совсем не обязательно привязано к началу машинного слова. Вследствие этого к ним неприменима одноместная операция взятия указателя &. Одно из наиболее типичных применений битовых полей — компактное хранение битовых флагов, экономящее также труд программиста: нет необходимости описывать множество флагов и масок для их извлечения. Запрет описания битового поля как массива отчасти компенсируется возможностью определять массивы переменных структурного типа и даже их явно инициализировать.

Например:

```
struct bmt
  {
  unsigned char fa:1;
  unsigned char fb:2;
  signed char fc:4;
  } bm1={1, 2, 3}, bma[3]=
    {{1, 3, 5},{0, 2, 4},{1, 1, 7}};
```

5.11. Битовые поля и typedef

Для того чтобы исключить из названия структурного типа слово struct, как и для обычных структур, можно предварительно ввести новое лаконичное имя типа с применением typedef следующим образом:

```
typedef struct
имя_структурного_типа_комбинированное_со_словом_struct
  {
  {тип_поля имя_поля : длина_поля_в_битах;}n
  } новое_эквивалентное_имя_структурного_типа;
```

Теперь определение конкретных представителей этого типа выглядит так:

```
новое_эквивалентное_имя_структурного_типа
список_имен_структур;
```

Например:

```
typedef struct st_bmt
  {
  unsigned char fa:1;
  unsigned char fb:2;
  unsigned char fc:3;
  } t_bmt;
…
struct st_bmt bm1={1, 2, 3};
t_bmt bma[3]={{1, 3, 5},{0, 2, 4},{1, 1, 7}};
```

Здесь `bm1` — одиночная структура с битовыми полями; `bma` — массив из трех таких структур.

5.12. Перечислимый тип `enum`

В практическом программировании часто приходится нумеровать однородные объекты, например цвета, дни недели, виды продукции. Чтобы избежать повсеместного указания числовых значений и связанной с этим путаницы, в Си имеется способ однозначно сопоставить целому числу его название. В общем виде:

```
enum [имя_перечислимого_типа]
{имя_значения[=числовое_значение]
[, имя_значения[=числовое_значение]]^n}
[список_переменных_данного_перечислимого_типа];
```

Как и для структур, данное выражение универсально:

- при наличии только `имени_перечислимого_типа` это чистая декларация;
- при наличии только `списка_переменных_данного_перечислимого_типа` (как всегда, через запятую) это только определение;
- при использовании обоих — это сочетание введения нового перечислимого типа и создания объектов этого типа.

Если первое числовое_значение в фигурных скобках опущено, оно берется по умолчанию равным нулю. При отсутствии очередного числового_значения оно равно увеличенному на единицу предыдущему значению.

Например:

```
enum seasons {winter=-7, spring, summer, fall} now,
after;
enum seasons before;
now=spring;
…
after=now==fall ? winter : now+1;
before=now==winter ? fall : now-1;
```

Из примера видно, что в рамках перечислимого типа названия времен года становятся синонимами целых чисел –7, –6, –5, –4. Можно сказать, что слова winter, spring, summer и fall являются именами констант перечислимого типа, а before, now и after – именами переменных. Как имена таких констант, так и имена соответствующих переменных подчиняются общим правилам наименования объектов в Си. Природа перечислимых переменных и констант – целочисленная, соответствующая типу int, поэтому во многих реализациях языка Си допускается их использование в выражениях наряду с обычными целыми числами.

Например:

```
printf("%d",before);
```

Присваивание целочисленного значения перечислимой переменной следует осуществлять очень осторожно, поскольку может получиться недопустимое значение, например: {now=fall; now++;}, так как ни программа, ни компилятор не знают о цикличности времен года. Ситуации такого рода не всегда обнаруживаются автоматически и могут приводить к «чудесам» в программе, поскольку переменные перечислимого типа могут быть операндами любых допустимых для int операций и способны бесконтрольно принимать любые целочисленные значения, будучи изменяемыми.

5.13. Создание альтернативных имен типов и typedef

На примерах структур, битовых полей и указателей на функции можно убедиться в том, что определения и декларации могут получаться громоздкими и плохо читаемыми. Здесь приходит на помощь конструкция typedef, позволяющая существенно проще описать тип объекта. В общем виде:

```
typedef декларация_типа новое_имя_типа;
```

Например:

```
typedef struct {unsigned length; char *name;} my_string;
…
my_string s1; char c[]="строка";
s1.name=c; s1.length=strlen(c);
```

Особенно полезно применение typedef с указателями на функции. Оно может быть использовано и в левом выражении, и в правом.

Например:

```
unsigned mycmp(double a1, double a2) { return a1>=a2; }
…
/* pf - это тип указателя на функцию, принимающую int и
возвращающую unsigned */
typedef unsigned (*pf)(int);
/* присваивание с преобразованием типа */
pf myfunc=(pf)mycmp;
```

Последняя строка, написанная для сравнения без применения typedef, красноречиво свидетельствует в его пользу:

```
unsigned (*myfunc)(int) = (unsigned (*)(int))mycmp;
```

Здесь в отличие от выражения с typedef левая и правая части относительно знака присваивания устроены неодинаково, что затрудняет восприятие.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Почему можно определить объект типа «объединение», а декларировать соответствующий тип нельзя?

2. Имеется битовое поле следующей конструкции:
```
struct { unsigned a:5; unsigned b:12;
unsigned c:20; unsigned d:14; } x;
```
Можно ли его оптимизировать для экономии памяти на вашем компьютере?
3. Каким образом на Си можно реализовать преобразование типа: *1*) с сохранением значения, *2*) с сохранением двоичного представления числа?
4. Имеются два структурных типа:
```
struct st1 {unsigned L; double a[100]; char *c;};
struct st2 {unsigned long l; unsigned short *b[10];
char d[1000]; };
```
В каких случаях необходимо позаботиться о выделении памяти при создании объектов этих типов? Почему?
5. Реализуйте гибкое хранилище, в котором указатели на векторы различной длины из элементов типа `double` объединены в единый вектор. Как бы вы реализовали хранение длин отдельных векторов типа `double`?
6. Имеются два структурных типа, один из которых объявлен через другой, и указатель на первый тип, инициализированный при помощи явного преобразования типа указателем на фактическую структуру второго типа:
```
typedef struct {int a; long b; long double c;} s1;
typedef struct {s1 d; double e; unsigned f;} s2;
s2 stru2={{-123345,999999,1.23e-123},5.67e-78, 9876};
s1 *ps1=(s1*)((void*)&stru2);
```
Поэкспериментируйте с прямым доступом при помощи `ps1` к полям `a`, `b` и `c` внутренней структуры типа `s1`, а также с доступом через поле `d` объемлющей структуры `s2`. Объясните результаты.
7. Реализуйте поиск заданного элемента в кольцевом связанном списке из 4 элементов, начиная с произвольной позиции. Рассмотрите обработку ситуации обнаружения и отсутствия искомого элемента в списке.
8. Как при помощи структур реализовать вершины произвольного графа?
9. Имеются 16 битовых флагов, объединенных в битовое поле:
```
struct z {unsigned a0:1; unsigned a1:1;
unsigned a2:1; unsigned a3:1; unsigned a4:1;
unsigned a5:1; unsigned a6:1; unsigned a7:1;
unsigned a8:1; unsigned a9:1; unsigned aa:1;
unsigned ab:1; unsigned ac:1; unsigned ad:1;
unsigned ae:1; unsigned af:1;} bf;
```
Достоверно известно, что лишь один из флагов установлен в единицу. Как очень быстро выяснить, какой именно по счету в `bf`? Гарантируется, что `sizeof(unsigned)`$\geq$`16`.
10. Попытайтесь осуществить арифметические действия над переменными перечислимого типа. Поддерживает ли ваш компилятор такую возможность? В чем опасность таких действий?

ГЛАВА 6 ОПЕРАТОРЫ. УПРАВЛЕНИЕ ХОДОМ ВЫЧИСЛЕНИЙ

6.1. Понятие оператора

Всякая нетривиальная программа состоит:

- ◊ из деклараций, утверждающих, что некоторый элемент программы (например, переменная, функция) существует;
- ◊ определений, выделяющих память под элементы исполнимого кода или данных программы;
- ◊ блоков и функций, разграничивающих отдельные фрагменты программы и описывающих их взаимодействие;
- ◊ выражений, дающих согласно заданному правилу для данной совокупности переменных и констант (операндов) конкретный числовой результат;
- ◊ возможно, директив препроцессора, упрощающих программисту работу с исходным текстом программы;
- ◊ операторов, которые осуществляют всю работу в программе – отвечают за последовательность вычислений и обработки данных и совместно с участвующими в них выражениями образуют скелет алгоритма.

Простейшим оператором является оператор присваивания:

```
a=b+2.0*c;
```

Он отличается от операции присваивания. В приведенном выражении имеются три операции. Сложение и умножение не меняют значений своих аргументов, а присваивание заносит в переменную `a` новое значение. Соответствующая операция `a=b+2.0*c`, будучи дополненной на конце точкой с запятой, становится оператором присваивания, выполняющим простое действие – изменение значения `a`. Известно «золотое правило» написания операторов: если последним символом оператора не является закрывающая оператор фигурная скобка, то на конце оператора обязательно ставится точка с запя-

той. Забвение замыкающей оператор точки с запятой – одна из наиболее распространенных ошибок начинающих программистов.

6.2. Составной оператор

Предположим ненадолго, что мы не используем трехместную операцию выбора `?:` Из операторов присваивания с любыми допустимыми операциями в выражениях в правой части от знака присваивания можно составить простой код, реализующий линейную (последовательную) модель вычислений, где операции и операторы следуют друг за другом линейно в порядке, не зависящем от значений переменных.

Например:

```
…
x+=d*sin(k*i)+bias; y=x*pow(m,16.2);
a[0]=a[1]+3*a[3]+5*a[5];
…
```

Подобно тому как выражения можно объединять в составное выражение посредством операции «запятая», операторы можно группировать в *составной оператор* (англ. composite statement), заключая их последовательность в фигурные скобки.

Например:

```
{
x+=d*sin(k*i)+bias; y=x*pow(m,16.2);
a[0]=a[1]+3*a[3]+5*a[5];
}
```

Теперь для внешней программы данный фрагмент кода является одним оператором, хоть и достаточно сложным. Произвольное число составных операторов можно снова объединить фигурными скобками в еще больший составной оператор.

Например:

```
{
  {
  x+=d*sin(k*i)+bias; y=x*pow(m,16.2);
```

```
    a[0]=a[1]+3*a[3]+5*a[5];
    }
  i++;
    {
    z=a[0]+x*(a[1]+x*(a[2]+x*(a[3]+x*(a[4]+x*a[5]))));
    z=y>z ? y : z;
    }
  mu=k*i;
  }
```

Хотя здесь снова введена условная операция, которая превращает код в нелинейный, это не меняет возможности объединения в составной оператор. Более того, составной оператор может включать средства ветвления вычислений в зависимости от промежуточных результатов и многое другое. Современный стандарт Си рассматривает всякий составной оператор как блок, следовательно, сразу после открывающейся фигурной скобки можно поместить определения констант и переменных, область видимости которых ограничена данным составным оператором (см. § 2.5).

6.3. Пустой оператор

Форма некоторых операторов предусматривает наличие подчиненного оператора, выполняемого при некоторых условиях. В некоторых случаях этот подчиненный оператор не нужен. Чтобы не нарушать правила языка, его заменяют на *пустой оператор* в виде пустого блока `{ }`, который ничего не делает.

6.4. Оператор `if`

Простейшая форма: только если значение логического выражения не равно нулю, выполняется `оператор`:

```
if (выражение) оператор
```

Форма с альтернативой: если значение проверяемого логического `выражения` не равно нулю, выполняется только `оператор_1`, иначе – только `оператор_2`:

```
if (выражение) оператор_1 else оператор_2
```

Общая форма:

```
if (выражение) оператор_1
[ else оператор_2 ]
```

В качестве любого из подчиненных операторов `оператор_1` и `оператор_2` может снова выступать любой, в том числе другой оператор `if`. Это способно создать множество вариантов выполнения вычислений (так называемое ветвление вычислительного процесса). Легко догадаться, как поступит компилятор, обнаружив зарезервированное слово `else`. При отсутствии ограничивающих составные операторы фигурных скобок `else` всегда приписывается последнему встретившемуся оператору `if`, не закрытому другим `else`.

Например:

```
if (a>=b)
   if (a>=c)
      x=a;
   else
      x=c; /* относится к if (a>=c) */
```

С другой стороны:

```
if (a>=b)
   {
   if (a>=c)
      x=a;
   }
else
   x=b; /* относится к if (a>=b) */
```

6.5. Вложенные операторы `if`

Последовательную обработку многих вариантов — так называемый *цепочечный* `if` — покажем на примере из 8 взаимно исключающих ветвей:

```
if (выражение_1) оператор_1
else if (выражение_2) оператор_2
else if (выражение_3) оператор_3
```

```
else if (выражение_4) оператор_4
else if (выражение_5) оператор_5
else if (выражение_6) оператор_6
else if (выражение_7) оператор_7
else оператор_8
```

При этом `оператор_2` будет выполнен, только если `выражение_1` дает нуль, а `выражение_2` – нет. Аналогично: `оператор_3` будет выполнен, только если `выражение_1==0`, `выражение_2==0`, а `выражение_3!=0` и т.д.

При отсутствии `else` сформируется совсем другая последовательность вычислений

```
if (выражение_1) оператор_1
if (выражение_2) оператор_2
if (выражение_3) оператор_3
if (выражение_4) оператор_4
if (выражение_5) оператор_5
if (выражение_6) оператор_6
if (выражение_7) оператор_7
```

В этом случае вложенный `if` распадается на независимые, не исключающие друг друга операторы `if`.

Древесная обработка многих вариантов показана далее на примере дерева полного перебора 16 вариантов.

```
if (выражение_0)
    if (выражение_0_0)
        if (выражение_0_0_0)
            if (выражение_0_0_0_0) оператор_0_0_0_0
            else оператор_0_0_0_1
        else
            if (выражение_0_0_1_0) оператор_0_0_1_0
            else оператор_0_0_1_1
    else
        if (выражение_0_1_0)
            if (выражение_0_1_0_0) оператор_0_1_0_0
            else оператор_0_1_0_1
        else
            if (выражение_0_1_1_0) оператор_0_1_1_0
            else оператор_0_1_1_1
else
    if (выражение_1_0)
        if (выражение_1_0_0)
            if (выражение_1_0_0_0) оператор_1_0_0_0
```

```
        else оператор_1_0_0_1
    else
        if (выражение_1_0_1_0) оператор_1_0_1_0
        else оператор_1_0_1_1
else
    if (выражение_1_1_0)
        if (выражение_1_1_0_0) оператор_1_1_0_0
        else оператор_1_1_0_1
    else
        if (выражение_1_1_1_0) оператор_1_1_1_0
        else оператор_1_1_1_1
```

Все проверяемые логические выражения занумерованы для наглядности в двоичном коде. Глубина закраски прямоугольников зрительно подчеркивает глубину вложенности, в данном примере максимально равную четырем. Отсюда видно, что выбрать одну возможность из 2^n можно, совершив не более n проверок. Подобная структура называется *полным двоичным деревом решений*.

6.6. Упрощение организации ветвления

Если это допустимо (возможность проверки `выражения_2` не зависит от выполнения `выражения_1`, а возможность проверки `выражения_3` не зависит от выполнения `выражения_2` и т.д.), рекомендуется объединять много `if` в один.

Например:

```
if (выражение_1) if (выражение_2) if (выражение_3) оператор
```

заменяется на

```
if (выражение_1 && выражение_2 && выражение_3) оператор
```

Одиночный `if` выполняется гораздо более эффективно потому, что в современных вычислительных системах даже весьма значительные последовательные операции «дешевле» одного ветвления вычислительного процесса.

Вычисление самого выражения тоже нередко удается упростить с помощью правил де Моргана, известных из алгебры логики.

В обозначениях языка Си:

```
(!выражение_1 && !выражение_2) ==
!(выражение_1 || выражение_2)
```

и

```
(!выражение_1 || !выражение_2) ==
!(выражение_1 && выражение_2)
```

Это экономит одну операцию, а в случае более сложных выражений выигрыш может быть значительнее.

6.7. Оператор switch

Цепочечный `if` можно записать более изящно:

```
switch (целочисленное_выражение)
  {
  case целочисленное_константное_выражение_1:
    оператор_1;
    [ break; ]
  case целочисленное_константное_выражение_2:
    оператор_2;
    [ break; ]
  case целочисленное_константное_выражение_3:
    оператор_3;
    [ break; ]
    …
  [default:
    оператор_по_умолчанию;]
 }
```

Здесь значение `целочисленного_выражения` поочередно (сверху вниз) сравнивается с `целочисленными_константными_выражениями` до первого совпадения. При совпадении выполняется соответствующий оператор и все последующие операторы до первого оператора `break`, который вызывает выход из конструкции `switch`. Совпадающие значения двух и более `целочисленных_константных_выражений` не допускаются, о чем компилятор сообщает как об ошибке.

В том случае, если значение целочисленного_выражения не совпало ни с одним из целочисленных_константных_выражений, управление передается в секцию default, которая в switch должна быть последней. Находящиеся там операторы осуществляют обработку всех нестандартных ситуаций, например могут сообщать о недопустимом значении целочисленного_выражения. Хотя секция default не является строго обязательной, ее использование рекомендуется как средство дополнительного контроля за ходом вычислений; в некоторых системах программирования ее отсутствие отражается в виде предупреждения во время компиляции. Если секция default отсутствует, происходит выход из switch и выполняется следующий за ним оператор.

Достоинствами оператора if являются:

- способность более гибко управлять ходом вычислений;
- предоставление большего выбора в отношении проверяемых выражений для выполнения той или другой ветви вычислительного процесса: они могут быть разными в каждом if и сравнение не ограничивается константами.

К достоинствам switch относятся:

- switch при помощи «контролируемого падения вниз до первого break» позволяет более изящно программировать выполнение одних и тех же операторов в разных случаях;
- switch обычно влечет создание более эффективного кода, чем аналогичный вложенный if, поскольку целочисленное_выражение всего одно и компилятор строит внутреннюю вспомогательную структуру данных для ускоренного выяснения (обычно методом половинного деления), в какую секцию перейдет управление.

6.8. Оператор безусловного перехода goto

При помощи конструкций с if можно в зависимости от значения выражения включать или исключать последовательности операторов по ходу программы. Иногда возникает потребность многократно использовать некоторый участок исходного кода программы, подставляя в него разные значения

переменных. Лучший способ это сделать – использовать функции или макроопределения. Однако есть немало алгоритмов, описываемых математической абстракцией, называемой конечным автоматом, где существует лучшее программистское решение.

Конечный автомат характеризуется наличием конечного числа состояний, в одном из которых он может находиться во всякий момент времени. При этом возникновение некоторых событий (например, результат анализа выражений) переводит автомат из одного состояния в другое. Таким образом, автомат в зависимости от промежуточных результатов вычислений переходит из состояния в состояние. Среди множества состояний выделяют два класса особенных – начальные, соответствующие тому или иному набору входных данных, и конечные, отвечающие различным исходам обработки данных. Пример простого конечного автомата приведен на рис. 6.1. Начальное и конечное состояния автомата обозначены соответственно буквами Н и К в кружкáх.

Р и с. 6.1. Схема состояний простого конечного автомата

На математическом аппарате конечных автоматов очень часто базируется встраиваемое в различные системы программное обеспечение, предназначенное для оптимизации ре-

жимов работы двигателей внутреннего сгорания, управления автоматическими дверями и камерами наблюдения, обеспечения функционирования практически любой сложной бытовой аппаратуры, промышленных, медицинских, транспортных, оборонных и других систем.

Места в коде программы, соответствующие отдельным состояниям автомата, удобно выделять при помощи так называемых меток. Метка размещается, начиная с самой левой позиции листинга программы, и завершается символом двоеточия. Всякая метка должна иметь собственное уникальное имя, которое может содержать те же символы, что и имя переменной в Си. Это имя помещается между началом строки и двоеточием, например:

```
unique_name_of_the_label:
```

Метка именует часть кода программы, следующую непосредственно за ней. Переходя по данной метке, программа попадает в начало этого фрагмента кода. Переход из другого места программы осуществляется при помощи оператора `goto`, общая форма записи которого очень проста:

```
goto имя_метки_куда_перейти;
```

Двоеточие не относится к имени метки и в операторе `goto` отсутствует. Этот оператор называют оператором *безусловного перехода*, так как его роль – передача управления в фиксированное наперед заданное место программы. Очевидно, что если оператор `goto` используется сам по себе, следующий за ним фрагмент кода программы становится недоступным, если туда нельзя попасть каким-то иным образом. Единственным способом обратиться к этому коду служит какой-то другой оператор `goto`.

Например:

```
goto label1;
label2:
a=b[0]+mu;
…
label1:
…
goto label2:
```

В подобных переходах нет смысла, поскольку это ухудшает понятность исходного текста, повышает шанс допустить ошибку и заставляет выполнять ничем не оправданные переходы. Всегда можно «развернуть» текст программы так, чтобы избежать `goto`. Использование этого оператора имеет смысл лишь при его парном применении с оператором `if`. Именно их сочетание выступает одним из наиболее адекватных средств программирования алгоритмов работы конечных автоматов. При этом необходимо соблюдать следующие понятные правила:

- посредством `goto` категорически нельзя попадать внутрь цикла, оператора `switch`, функции и вообще любого программного блока, ограниченного фигурными скобками, поскольку при таком входе утрачивается контекст использования этого программного фрагмента, не определено его поведение и программа «не видит» расположенных в начале блока определений переменных и констант;
- не следует вводить метку, на которую не осуществляется переход;
- в программе не должно быть операторов `goto` вне `if`;
- необходимо избегать «очень далеких» переходов, которые трудно отследить. Такие фрагменты кода нужно локализовать и оформить их как отдельные блоки или функции;
- следует четко выделять точки входа и выхода фрагментов кода с применением переходов (состояния автомата), определить и минимизировать набор переменных, исчерпывающим образом описывающих каждое состояние автомата;
- нужно внимательно проверять сохранение контекста переменных при переходах, чтобы в них не оказались неправильные значения и не возникли неинициализированные переменные;
- количество операторов `goto` должно быть минимальным, так как этот оператор «опасен» в том смысле, что способен чрезвычайно запутать текст программы. Поэтому многие специалисты в области информатики считают вообще недопустимым использование `goto`. Эти взгляды мне кажутся слишком категоричными, поскольку я неоднократно был свидетелем того, как при помощи минимального использования `goto` удавалось крайне запутанные реализации сложных алгоритмов сделать более эффективными и легкими для восприятия. Очевидно, общая рекомендация такова: следует использовать `goto` только при значительном улучшении читаемости программы.

6.9. Цикл for

Цикл for (англ. для) служит удобным средством выполнения некоторой последовательности операторов (называемой телом цикла) фиксированное число раз, однако его возможности этим не исчерпываются. Его общая запись такова:

```
for (выражение_инициализации_цикла;
логическое_выражение_являющееся_условием_продолжения_цикла;
выражение_перехода_к_следующему_проходу_цикла)
  оператор_тела_цикла
```

Принцип действия цикла for удобно пояснить на простом примере:

```
int k, a[K], c=1, alpha=3;
…
for (k=0; k<K; k++) a[k]=(c*=alpha);
…
```

Точка с запятой относится к подчиненному оператору_тела_цикла, а не к оператору for. Конструкция for берет последовательно значения k от 0 до K-1 и присваивает всем соответствующим a[k] последовательно изменяемые согласно геометрической прогрессии значения c.

При выполнении цикла важна последовательность операций:

1) в самом начале (как говорят, при входе в цикл) выполняется выражение_инициализации_цикла;

2) осуществляется проверка логического_выражения_являющегося_условием_продолжения_цикла; если оно дает значение «истина», происходит однократное выполнение оператора_тела_цикла, иначе происходит выход из цикла;

3) вычисляется выражение_перехода_к_следующему_проходу_цикла;

4) производится проверка логического_выражения_являющегося_условием_продолжения_цикла. Если оно и на этот раз дает значение «истина», вновь один раз выполняется оператор_тела_цикла, иначе происходит выход из цикла;

5) так циклически продолжается до тех пор, пока `логическое_выражение_являющееся_условием_продолжения_цикла` принимает значение «истина» (как говорят, выполнено условие цикла). При первом невыполнении условия цикла сразу осуществляется выход из цикла, в результате которого выполняется следующий за телом цикла оператор.

Поскольку цикл `for` проверяет условие до первого выполнения тела цикла (т.е. до первого прохода цикла), его относят к циклам *с проверкой в начале*. При этом гарантируется, что при невыполнении проверки в момент входа в цикл его тело не будет выполнено ни разу.

Выражения внутри круглых скобок – `выражение_инициализации_цикла`, `логическое_выражение_являющееся_условием_продолжения_цикла` и `выражение_перехода_к_следующему_проходу_цикла` – могут быть не только простыми, как в приведенном примере, но и составными (с перечислением отдельных выражений через запятую) или даже пустыми. В случае составного выражения действия, как и обычно, выполняются последовательно слева направо (см. § 4.15) и значение составного выражения совпадает со значением последнего выражения в цепочке. Таким образом, нет особого смысла делать составным `логическое_выражение_являющееся_условием_продолжения_цикла`. Однако в него можно включить все выражения, которые нужно выполнить перед каждым проходом цикла. В то же время отсутствующее `логическое_выражение_являющееся_условием_продолжения_цикла` трактуется как константа, имеющая значение «истина». При отсутствии возможности выхода из тела цикла такой цикл делается бесконечным. Механизм, позволяющий покинуть тело цикла, рассматривается в § 6.11.

Тело цикла состоит из одного оператора, который может быть простым, как в приведенном выше примере, и составным. Как и всегда, операторы, входящие в составной, объединяются при помощи фигурных скобок.

Например, цикл `for` с составными выражениями и составным оператором в теле цикла запишется так:

```
for (i=0, k=K; i<k; i++, k--) { tmp=a[i]; a[i]=a[k];
a[k]=tmp; }
```

Циклы for, как и прочие, можно делать вложенными, т.е. вставлять один цикл внутрь другого.

Например:

```
for (m=0; m<M; m++) for (n=0; n<M; n++) a[m][n]=a[n][m];
```

При этом внутренний цикл является телом внешнего и должен отвечать всем требованиям к телу цикла.

Во-первых, невозможность оказаться в теле цикла, минуя оператор цикла. Это означает, что передавать в цикл управление при помощи goto является одной из грубейших ошибок, которая приводит к непредсказуемым результатам. Многие современные компиляторы обнаруживают эту ошибку.

Во-вторых, непересечение циклов. Это означает, что вложенный цикл должен полностью размещаться внутри тела объемлющего цикла, т.е. внутренний цикл не должен включать в себя операторы за пределами внешнего цикла. Хотя средства языка позволяют создавать такие циклы, доказано, что нет ни малейшей необходимости в этом, а отлаживать такие программы чрезвычайно затруднительно.

В-третьих, отсутствие модификации переменных внешнего цикла во внутреннем. Строго говоря, стандарт языка Си этого не запрещает, но такие циклы также очень трудно проверять при отладке. В частности, изменение переменной цикла for внутри тела цикла является плохим стилем программирования уже потому, что результат такого изменения зависит от реализации компилятора и в общем случае непредсказуем.

Многие программисты предпочитают для наглядности всегда заключать тело цикла в фигурные скобки (что не запрещается правилами языка Си и не приводит к каким-либо неприятностям), а скобки размещают с отступом от левого края, отражающим степень вложенности цикла.

Например:

```
for(i=0;i<i_max;++i)
  {
  for(j=0;j<J;++j)
    {
```

```
    tmp=2*pi*i*j;
    for(k=0;k<K;k++)
      {
      t[i][j][k]=a*sin(tmp*(k-j));
      }
    }
  }
```

Достоинство такой записи — в ее улучшенной читаемости, поэтому она особенно полезна при большой вложенности крупных циклов. Тогда по размеру отступа можно сразу определить, к какой вложенности цикла относится данный фрагмент исходного текста. При этом одна из наиболее распространенных ошибок — непарные скобки, что вызывает сообщение об ошибке во время компиляции. Предельная вложенность циклов в программе ограничена только ресурсами компьютера и приемлемым временем счета. Легко видеть, что число выполнений тела самого внутреннего вложенного цикла равно произведению числа выполнений всех объемлющих циклов. Следовательно, оно очень быстро возрастает с увеличением числа их проходов.

6.10. Цикл `while` и `do...while`

Цикл `while` (англ. до тех пор, пока), подобно циклу `for`, производит проверку условия продолжения цикла перед каждым выполнением тела и в общем виде записывается так:

```
while (логическое_выражение_условия_продолжения_цикла)
оператор_тела_цикла
```

Логика у него проще, чем у цикла `for`. При входе в цикл осуществляется проверка `логического_выражения_условия_продолжения_цикла` и в случае его истинности единожды выполняется `оператор_тела_цикла`. Этот процесс повторяется до тех пор, пока условие продолжения цикла выполнено. После этого управление получает оператор, следующий за телом цикла.

Форма оператора цикла `while` не предусматривает ни инициализации, ни отдельно оформленного перехода к следующе-

му проходу (который называют *итерацией*, англ. iteration). Легко убедиться, что такая форма записи не является ограничением: инициализацию можно осуществить до входа в цикл, а переход к новой итерации ввести в тело цикла. Подобно оператору for, тело оператора while может содержать произвольное число вложенных циклов любого типа.

Например:

```
row=0;
while (found_elements<=n && row<=row_max)
  {
  found_elements=0;
  for (column=0; column<=c_max; column++)
    if (m[row][column]>=threshold) found_elements++;
  row++;
  }
```

Внешний цикл while остановится, как только в очередной строке матрицы m будет найдено более n элементов, превышающих число threshold, либо все строки матрицы будут исчерпаны.

Цикл do...while (англ. делай до тех пор, пока) отличается от уже рассмотренных тем, что проверку условия продолжения осуществляет в конце, после выполнения тела цикла:

```
do оператор_тела_цикла
while (логическое_выражение_условия_продолжения_цикла);
```

Например:

```
r=0;
do
  {
  for (c=0, Norm[r]=0.0; c<RowWidth; c++)
    Norm[r]+=x[r][c]*x[r][c];
  r++;
  }
while (r<ColumnHeight);
```

Характерной ошибкой при написании цикла do...while является пропуск точки с запятой после проверяемого логического выражения.

6.11. Оператор break

Оператор break (англ. прервать) может быть размещен внутри операторов for, do...while, while, switch и прерывает в текущем месте выполнение именно той управляющей структуры, в которой встретился, передавая управление идущему следом за этой структурой коду, например внешнему циклу. Недопустимо выходить из if...else или из функции при помощи break. В приведенном ниже примере с помощью одного break будет осуществлен выход из внутреннего цикла while во внешний цикл for, а с помощью другого — из цикла for.

Пример:

```
for (k=0, cnt=0, acc=0; k<K; k++)
    {
    l=0;
    while (l<=k)
        {
        acc+=a[k][l];
        if (a[k][l]<0) break;
        l++; acc+=b[l]; cnt++;
        }
    /* здесь окажется программа при срабатывании
    break в цикле while */
    acc+=b[k]; cnt++;
    if (b[k]<0) { acc/=alpha; break; }
    acc*=alpha;
    }
/* здесь окажется программа при срабатывании break в цик-
ле for */
```

В одном цикле, как и в одном операторе switch, может быть несколько операторов break. Их называют *дополнительными точками выхода* из цикла. Не стоит использовать в циклах оператор break вне пары с if, так как часть тела цикла становится заведомо недоступной, а также вставлять break в самое начало или самый конец тела цикла, поскольку этот код всегда можно более наглядно переписать без использования break путем изменения условия в операторе цикла или переработки самого цикла.

Основное назначение сочетания if и break — выход из середины тела цикла по результатам дополнительных проверок.

Если имеется возможность избежать применения break в цикле, не слишком усложняя его конструкцию, эту возможность целесообразно реализовать, поскольку использование сочетания if и break вводит дополнительное часто повторяемое ветвление, а ветвления вычислений большинство существующих архитектур процессоров выполняет медленно. Как правило, сложное выражение условия продолжения цикла обрабатывается заведомо быстрее отдельной простой проверки в if.

6.12. Оператор continue

Оператор continue (англ. продолжить) может быть размещен только внутри тела оператора цикла for, do...while или while и действует иначе, чем оператор break. При выполнении continue происходит переход к концу области, охваченной *текущим* циклом (например, к закрывающей тело цикла фигурной скобке). Вслед за этим предпринимается попытка проверить условие продолжения цикла и, если оно удовлетворяется, цикл выполняется далее. Можно сказать, continue позволяет «перепрыгнуть» через оставшуюся после него часть тела цикла на данной итерации, не выходя из цикла вообще.

Например:

```
for (m=1; m<100; ++m)
  {
  b[m]-=b[m-1];
  if (m==5) continue;
  b[m]*=3; a[m]=sin(b[m]+a[m]);
  }
```

В данном примере элемент массива b[5] не будет умножен на 3 и a[5] не получит нового значения.

В теле цикла допустимо иметь несколько операторов continue и комбинировать их с break. Очевидно, использовать continue в отрыве от if бессмысленно, поскольку тогда идущая после него часть тела цикла не будет выполнена никогда. Все сказанное в отношении сочетания if и break в равной степени относится к связке if и continue.

6.13. Эквивалентные преобразования циклов

Покажем, что одного любого способа задания цикла в принципе достаточно для записи произвольного алгоритма. Для этого выразим каждую форму записи цикла через две другие в самом общем виде.

1. Цикл:

```
while (логическое_выражение_условия_продолжения_цикла)
оператор_тела_цикла
```

имеет эквивалентные записи

◊ через `for`:

```
for
  ( ; логическое_выражение_условия_продолжения_цикла; )
    оператор_тела_цикла
```

◊ через `do...while`:

```
if (логическое_выражение_условия_продолжения_цикла)
  {
  do оператор_тела_цикла
  while
    (логическое_выражение_условия_продолжения_цикла);
  }
```

◊ посредством комбинации `if` и `goto`:

```
label_of_loop_entry:
if (логическое_выражение_условия_продолжения_цикла)
  {
  оператор_тела_цикла
  goto label_of_loop_entry;
  }
```

2. Цикл:

```
do оператор_тела_цикла
while (логическое_выражение_условия_продолжения_цикла);
```

имеет эквивалентные записи

◊ через `for`:

```
for (first_run=1, выражение_инициализации_цикла;
```

```
first_run ||
логическое_выражение_условия_продолжения_цикла;
first_run=0,
выражение_перехода_к_следующему_проходу_цикла)
  оператор_тела_цикла
```

◊ через `while`:

```
first_run=1;
while
  (first_run ||
   логическое_выражение_условия_продолжения_цикла)
  {
  оператор_тела_цикла
  first_run=0;
  }
```

◊ при помощи комбинации `if` и `goto`:

```
label_of_loop_entry:
оператор_тела_цикла
if (логическое_выражение_условия_продолжения_цикла)
  goto label_of_loop_entry;
```

3. Цикл:

```
for (выражение_инициализации_цикла;
логическое_выражение_условия_продолжения_цикла;
выражение_перехода_к_следующему_проходу_цикла)
  оператор_тела_цикла
```

имеет эквивалентные записи:

◊ через `while`:

```
выражение_инициализации_цикла;
while (логическое_выражение_условия_продолжения_цикла)
  {
  оператор_тела_цикла
  выражение_перехода_к_следующему_проходу_цикла;
  }
```

◊ через `do...while`:

```
выражение_инициализации_цикла;
if (логическое_выражение_условия_продолжения_цикла)
```

```
{
do
  {
  оператор_тела_цикла
  выражение_перехода_к_следующему_проходу_цикла;
  }
while
  (логическое_выражение_условия_продолжения_цикла);
}
```

◊ с привлечением сочетания `if` и `goto`:

```
выражение_инициализации_цикла;
label_of_loop_entry:
if (логическое_выражение_условия_продолжения_цикла)
  {
  оператор_тела_цикла
  выражение_перехода_к_следующему_проходу_цикла;
  goto label_of_loop_entry;
  }
```

Естествен вопрос: зачем в Си предусмотрены три разновидности циклов, если каждая из них может быть получена при помощи любой другой, а также операторов `if` и `goto`? Язык предоставляет эту избыточность для того, чтобы у программиста был выбор наиболее изящной реализации цикла в каждом конкретном случае. При этом конструирование цикла при помощи `if` и `goto` всегда выглядит более неуклюже и приводит к худшему коду программы, чем стандартные циклы, не говоря об удобстве отладки.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Можно ли реализовать полноценную конструкцию `if(...) {...} else {...}` при помощи одного оператора `switch`? Если нет, почему? Если да, как?
2. Реализуйте вложенный оператор `switch`. Куда осуществится переход при срабатывании `break` во внутреннем `switch`? Почему?
3. Можно ли в данном фрагменте кода
   ```
   for (i=0;i<I;i++) { a[i]+=b[i];
   if (a[i]>threshold) continue; a[i]-=c[i]; }
   ```
   обойтись без `continue`? Если да, как? Если нет, почему?

4. Можно ли в данном фрагменте кода
```
for (i=0; i<I; ++i) { a[i]*=b[i];
if (fabs(a[i])<eps) break; c[i]/=a[i]; }
```
обойтись без `break`? Если да, как? Если нет, почему?
5. Имеется ли возможность более оптимально запрограммировать следующий фрагмент программы?
```
for(i=0;i<I;i++)
for(j=0;j<J;++j) a[i][j]*=b[i]*c[j];
for(k=0;k<=I-1;++k) d[k]=0.5*e[k+1];
```
Если да, как? Если нет, почему?
6. Внутри условия оператора `while` реализуйте весь заголовок следующего оператора `for(i=0; i<=99; i++)`.
7. Реализуйте функцию циклической свертки вектора длиной `K` с вектором длиной `L`.
8. При помощи циклов организуйте эффективное последовательное считывание элементов массива `float x[K][L]` по столбцам справа налево (внешний цикл) и по строкам снизу вверх (внутренний цикл).
9. Как рационально реализовать цикл с единственной проверкой условия продолжения в середине тела цикла?
10. Куда относится каждый `break`:
```
switch(a)
{
case 1:
for(i=1;i<=I;i++) if (b[i]>0) break;
case 2:
for(i=1;i<=I;i++) if (c[i]>0) break;
}
```
Как вся эта конструкция работает?
11. Что будет, если в предположении об определенности и допустимости значений всех переменных написать так:
```
for(i=1;i<=I;i++)
for(j=1;i<=J;j++) if (b[i][j]>0) { break; break; }
```
Поставьте эксперимент и объясните его результат.
12. Известно, что оператор `if` довольно медленно работает, в некоторых реализациях даже медленнее, чем простой оператор `for`. Следует проверить, используя один-единственный `if`, что в массиве `unsigned a[N]` есть хотя бы одно число, превосходящее $t = 2^n-1$. Меньшее разрядности `unsigned` натуральное число `n` задано. Годится ли аналогичный подход для проверки того, что все элементы массива `a` больше `t`? Аргументируйте свой ответ.

ГЛАВА 7 ФУНКЦИИ И ИХ ПАРАМЕТРЫ

7.1. Функции и процедурное программирование

Язык Си относят к процедурным модульным языкам программирования. Это означает, что для удобства программиста и пользователя, а также для повышения гибкости и зачастую эффективности программу разбивают на независимые блоки, каждый из которых выполняет определенную четко заданную функцию. Имеется возможность привязки данных и методов их обработки друг к другу. Всякая функция, кроме выполняемого действия, характеризуется набором передаваемых параметров и типом единственного возвращаемого параметра. Совокупность выполняемой функцией задачи и набора параметров функции называется интерфейсом функции (программным интерфейсом, англ. programming interface). Подобно кирпичикам конструктора Lego, из функций собирают программу. Очевидно, функции-кирпичики должны хорошо сопрягаться друг с другом. Представим себе, что было бы, если бы программу писали единым фрагментом кода.

Во-первых, ее исходный текст, часто достигающий десятков–сотен тысяч строк, был бы совершенно необозримым и непонимаемым. Очевидно, что в таком тексте чрезвычайно высока вероятность допустить ошибку, а шансов ее отыскать почти нет.

Во-вторых, очень часто один и тот же модуль желательно вставить в различные места программы. При монолитной конструкции программы его пришлось бы вставлять с помощью механического копирования текста. В результате код программы становится столь огромным, что может не помещаться в оперативной памяти компьютера, не говоря о данных!

В-третьих, в монолитной программе продолжают занимать место все переменные и массивы, которые давно стали не

нужны по мере продвижения вычислений, что отрицательно сказывается на эффективности программы. Проследить изменения данных от исходной информации к результату становится практически нереально.

В-четвертых, монолитные программы все равно необходимо как-то соединять друг с другом для решения сложных задач; кроме того, они должны взаимодействовать с операционной системой. Через файлы это делать очень неэффективно, а способы соединения программных модулей друг с другом относятся к области процедурного программирования. Таким образом, современный программист неизбежно обращается к концепции процедурного программирования.

В языке Си процедурный принцип построения программ реализуется посредством функций. Функции в Си имеют немало общего с функциями в математическом смысле этого слова, но между ними есть существенные отличия:

- функции в Си реализуют не абстрактную связь параметров, а ее конкретную алгоритмическую реализацию, следовательно, обычно не обладают столь же высокой гибкостью;
- функции в Си могут не принимать и/или не возвращать никакого значения. Например, такова функция печати заранее заданного фиксированного сообщения;
- программные функции могут пользоваться глобальными (отсутствующими в списке передаваемых параметров) переменными и константами, таким образом реализуя неявные связи и имея внешний контекст своего действия;
- функции в Си способны автоматически осуществлять повторный вызов себя и других функций, что дает эффективное средство программирования ряда алгоритмов.

7.2. Определения и декларации функций

Любая функция характеризуется следующими элементами программного интерфейса:

- имя функции, которое должно отвечать таким же требованиям, как и имя переменной. Во многих реализациях системы программирования Си имя функции, начинающееся с симво-

ла подчеркивания, предусматривает некоторые особенные свойства функции, влияющие на процесс сборки программы;

- ◊ список передаваемых в функцию параметров состоит из перечисляемых через запятую пар *тип параметра – имя параметра*. Как и всякая другая переменная, каждый передаваемый параметр должен иметь конкретный тип. Число параметров в списке принципиально не ограничено. Имеется возможность создавать функции с переменным числом параметров или вовсе без входных параметров;
- ◊ тип возвращаемого значения также должен быть оговорен. Если программист задумал создать функцию, ничего не возвращающую, в качестве возвращаемого типа он указывает *пустой тип* `void` (англ. пустышка, несуществующая величина).

Подобно тому, как для переменных и констант различают определение и объявление (декларацию), так же поступают для функций. Определение функции, исчерпывающим образом описывающее ее устройство и способ действия, обязательно включает *тело функции* (англ. function body, function code), а ее объявление, именуемое также *прототипом* (англ. prototype, declaration), – нет. Объявление используется для сообщения компилятору о наличии такой функции далее в данном файле или в каком-то другом файле проекта, чтобы избежать сообщения об ошибке при попытке вызвать функцию. Прототип сообщает о типах входных параметров функции и ее возвращаемого значения и в общем случае выглядит так:

```
[спецификатор] тип_возвращаемого_значения имя_функции
(список_передаваемых_параметров_прототипа);
```

Определение функции отличается от прототипа заменой точки с запятой после закрывающейся круглой скобки на тело самой функции, размещенное внутри фигурных скобок:

```
[спецификатор] тип_возвращаемого_значения имя_функции
(список_передаваемых_параметров_определения)
{ код_тела_функции }
```

Наличие фигурных скобок отражает тот факт, что тело функции является программным блоком и обладает всеми его свойствами. Необязательный спецификатор указывает особенности этой функции и может принимать значения: `extern`, `static`, `inline`. Для одной и той же функции число передаваемых пара-

метров и их типы должны в точности совпадать в списке_передаваемых_параметров_прототипа и списке_передаваемых_параметров_определения. Список передаваемых параметров определения имеет следующий общий формат:

```
[[[register] тип имя_параметра,]^n
[register] тип имя_параметра[,…]]
```

Многоточие в конце списка_передаваемых_параметров ставится только в том случае, если функция допускает переменное число параметров. Необязательный спецификатор хранения register сообщает компилятору о желании программиста передать данный параметр в функцию через регистр процессора, а не через стек. Эта просьба необязательно будет выполнена компилятором, поэтому нельзя строить никаких предположений об истинном механизме попадания данного параметра в функцию.

В целом вид списка передаваемых параметров определения функции и ее прототипа одинаков. Но поскольку прототип не содержит кода функции (ее тела), то не важно, как в нем называются переменные, имеют значение только их типы и порядок следования. Поэтому существует альтернативная, более короткая форма записи списка передаваемых параметров прототипа:

```
[[[register] тип,]^n [register] тип[,…]]
```

Особый случай – функция, вовсе не имеющая входных параметров. В ее прототипе и определении внутри круглых скобок размещают единственное слово void, именно оно разъясняет компилятору отсутствие параметров у функции. Такова, например, следующая функция, силами препроцессора и стандартной библиотеки печатающая название файла и номер строки, где находится printf() в ее теле:

```
void where_am_i(void){
printf("\nline %d in file %s", __LINE__, __FILE__);
return; }
```

Во многих системах разработки для совместимости с написанными ранее исходными текстами программ поддерживается старый способ оформления прототипа и определения таких функций – просто пара пустых круглых скобок, например: void where_am_i(). Пользоваться им при написании новых программ настоятельно не рекомендуется, поскольку он от-

ключает проверку соответствия параметров функции и делает незаметными ошибки программиста. При современном способе оформления функции: `void where_am_i(void)`, напротив, компилятор в состоянии контролировать вызов функции, сверяя ФоП в ее прототипе и определении и ФаП. При вызове функции ключевое слово `void` не указывается.

В качестве полного примера приведем определение функции `get_ptr_max`, получающей в качестве первого параметра указатель на массив `a` типа `double`, а в качестве второго — число `n` (типа `unsigned`) элементов в этом массиве и возвращающей указатель на максимальный элемент в массиве `a`. Далее в функции `main` формируется массив `b` и для него вызывается данная функция.

Пример:

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

double* get_ptr_max(double *a, unsigned n)
{
double *a_max;
if (a==0 || n==0)
  {
  /* нулевой указатель или пустой массив */
  return (double*)0;
  }
a_max = &a[0];
for (n--; n>0; n--) if (*(a+n) > *a_max) a_max = a+n;
return a_max;
}


int main()
{
double b[100], *b_ptr_max;
int i, b_length=10;

for (i=0; i<b_length; ++i)
  {
  b[i] = exp(-0.2*i) * sin(0.1*i);
  printf("\n  b[%d] = %g",i,b[i]);
  }
```

```
b_ptr_max = get_ptr_max(&b[0], b_length);
printf("\n\n  наибольший элемент в массиве -\
это b[%d] = %g\n", b_ptr_max-b, *b_ptr_max);
return 0;
}
```

Приведенный пример иллюстрирует несколько важных фактов относительно использования функций. О существовании функции программист должен сообщить компилятору до первого ее использования. Иначе компилятор, разбирая строку, где функция вызывается, не сможет понять, что это такое и правильно ли это написано:

```
b_max = get_ptr_max(&b[0], b_length);
```

Компилятору необходимы сведения не только о названии функции, но также о типах ее параметров и возвращаемого значения, чтобы он мог проверить, верно ли функцию вызывают. Имеются три возможности уведомить компилятор о существовании функции:

1) привести определение функции до первого ее вызова, как это сделано в примере;

2) заранее привести прототип функции, а саму функцию определить дальше в том же файле;

3) привести прототип функции перед первым обращением к ней, а саму функцию определить в другом файле. В этом случае отсутствие функции в данном файле компилятору требуется объяснить, разместив в начале прототипа спецификатор `extern`.

7.3. Оператор возврата из функции `return`

Когда работа функции завершена, необходимо покинуть ее тело и вернуться в то место, откуда ее вызвали, а также передать туда возвращаемое значение, если оно предусматривается. Компилятору нужно дать об этом знать. Только простейшие функции, вроде функции `main` в нашем примере, имеют элементарную (как говорят, последовательную) структуру вычислений и точка выхода совпадает с закрывающейся фигурной скобкой. Функция `get_ptr_max` оснащена двумя точками выхода:

аварийной

```
if (a==0 || n==0) return (double*)0;
```

и штатной:

```
return a_max;
```

Оператор `return` (англ. возврат) указывает, в каком месте кода функции из нее необходимо выйти и какое конкретное возвращаемое значение вернуть в место вызова. Его синтаксис в общем виде таков:

```
return [выражение];
```

Выражение может в принципе иметь произвольный тип (например, целый, плавающий, указатель). Важно лишь, чтобы этот тип соответствовал типу возвращаемого функцией значения, указанному в определении и прототипе, или мог быть автоматически к нему преобразован. Если функция не возвращает никакого значения (определена как `void`), параметр `выражение` опускают:

```
return;
```

Программисты часто заключают выражение, дающее возвращаемое значение, в круглые скобки. Это необязательно, но многие авторы программ так делают в силу традиции и для более четкого визуального выделения выражения.

Поскольку возвращаемое функцией значение всего одно, важно понимать, какого типа оно может быть и что через него можно передать в вызывающую функцию. Следует выполнять простое правило: возвращаемый функцией тип может быть в принципе произвольным, кроме массива и функции. Тем не менее можно вернуть указатель на массив и указатель на функцию. При выборе типа возвращаемого значения следует задуматься о том, что передача большой структуры данных требует значительных усилий процессора и дополнительной оперативной памяти для стека и поэтому может существенно затруднять выполнение программы. Более того, по-прежнему используется немало компиляторов, не допускающих возвращаемое значение иного вида, чем скаляр простого типа. Из дальнейшего обсуждения станет ясно, что даже это не является сколько-нибудь серьезным ограничением объема и характера передаваемой из функции информации.

7.4. Формальные и фактические параметры функции

Из примера § 7.2 видно, что массив внутри функции имеет имя `a`, в то время как при вызове функции в нее передается массив `b`. Это не ошибка. Имена передаваемых в функцию параметров не связаны с именами этих же параметров внутри функции. Рассмотрим, как устанавливается соответствие.

Параметры, описываемые в определении функции как получаемые ею, называются *формальными параметрами* (ФоП, англ. formal argument), поскольку при написании функции им пока еще не соответствуют конкретные числовые значения и эти параметры носят как бы абстрактный характер. Это всего лишь имена автоматически формируемых внутри переменных, в которые в момент вызова функции будут загружены входные данные.

Каждый раз в момент вызова функции ей передаются вполне конкретные числа, и всякий раз они могут быть разными. Соответствующие переменные, константы и выражения называются *фактическими параметрами* (ФаП, англ. actual argument).

Каждому ФоП должен в точности соответствовать один ФаП, чем и устанавливается их взаимное соответствие при вызове функции. При этом ФоП и ФаП должны соответствовать друг другу не только количеством, но и типом. Например, если в нашем примере передать в функцию `get_ptr_max` в качестве первого параметра указатель не на `double`, а на `float`, данные будут переданы неправильно, так как длина и формат этих типов различны. В то же время второй параметр `n` в определении функции `get_ptr_max` описан как `unsigned`, а передаваемый параметр `b_length` имеет тип `int`. Это не является проблемой, если только не пытаться передать в функцию отрицательное `n`. Для параметров обычных функций (исключением являются функции с переменным числом параметров) действуют те же правила преобразования типов, что и для обычных переменных.

7.5. Функции и видимость переменных и констант

Внутри функции, как и внутри всякого программного блока, ограниченного фигурными скобками, можно определять любые переменные и константы. Они видны только внутри тела этой функции, что логично, так как тело функции является блоком. Внутренние переменные и константы недоступны даже из вложенных функций, вызываемых из данной. Кроме них изнутри функции доступны глобальные переменные и константы, определенные или декларированные до тела этой функции. Однако передача данных в функцию через глобальные объекты, хотя и очень эффективна, чревата трудно обнаружимыми ошибками и конфликтами с другими функциями.

Использование глобальных переменных является плохим стилем программирования, поскольку создает сложности при поиске ошибок и препятствует написанию корректно работающих многопроцессорных приложений. Внутри функции можно создавать блоки, ограниченные фигурными скобками, и в начале блока описывать свои внутренние переменные и константы. Их видимость ограничена блоком.

7.6. Обмен данными при вызове функции

Основной и единственно рекомендованный способ передачи данных в функцию – через параметры ее вызова. Важно понимать, как эти данные попадают в функцию, что позволит писать корректно работающие и безопасные программы, не имеющие неприятных «побочных эффектов». Вспомним, как размещаются в памяти переменные и константы. Если они относятся к самому распространенному автоматическому классу хранения, действующему по умолчанию, место их размещения – вершина стека. Пусть функции не передается ни одного параметра:

```
double random(void)
{
```

```
double gaussian_random_number=1;
int i;
for (i=100; i>0; --i) gaussian_random_number *= rand();
return gaussian_random_number;
}
```

Тогда при ее вызове указатель стека установлен на его вершину, а именно: на первый свободный байт стека. В стеке уже находятся все переменные и константы, определенные к этому моменту времени в вызывающей функции и внешних по отношению к ней функциях. После открывающейся фигурной скобки в функции `random` определена переменная `gaussian_random_number`. Компилятор выделяет для нее место, передвигая указатель стека на `sizeof(double)` байт далее, и записывает на это место единицу в формате `double`, чтобы осуществить инициализацию. Следом идет определение переменной `i`, и компилятор поступает аналогично – перемещает указатель стека вперед еще на `sizeof(int)` байт. Но поскольку программист не произвел инициализацию `i` , до момента входа в цикл `for` в переменной `i` остается «мусор» – те данные, которые случайно оказались в этой области памяти после включения компьютера или предыдущих действий с памятью. Теперь место для переменных выделено, и компилятор генерирует код самой функции. Встретив оператор `return`, компилятор должен удалить далее ненужные внутренние переменные покидаемой функции (в нашем примере это `gaussian_random_number` и `i`), для чего указатель стека возвращается в исходное положение на момент вызова функции, т.е. снова указывает на начало переменной `gaussian_random_number`. Она затирается выкладываемым на стек возвращаемым функцией значением. В нашем случае это значение и есть переменная `gaussian_random_number`, поэтому компилятор не выполняет лишней бесполезной работы (и программист экономит немного тактов работы процессора). Вызывающая функция, фрагмент которой приведен ниже

```
…
mu=a*random()-b;
…
if ((tmp=abs(random())) > u) u + = tmp;
…
```

получает со стека возвращаемое функцией `random()` значение и «мысленно» вставляет его в то место выражения, где был написан вызов функции. Если программист не интересуется возвращаемым функцией значением, он пишет в вызывающей функции

```
...
random();
...
```

и это значение пропадает. Компилятор может даже не помещать его в переменную или регистр, а сразу выбрасывать, сдвигая указатель стека на его размер назад.

Рассмотрим более сложный случай – функцию с двумя параметрами.

Например:

```
double random_enhanced(unsigned n, double norm)
{
double gaussian_random_number=1;
int i;
for (i=n; i>0; --i) gaussian_random_number *= rand();
return gaussian_random_number*norm;
}
```

В вызывающей функции на момент вызова должны быть посчитаны значения передаваемых параметров.

Например:

```
...
if ((tmp=abs(random_enhanced(k*N,mu-delta))) > u) u + = tmp;
...
```

Функция `random_enhanced` в качестве первого фактического параметра получает готовое значение выражения `k*N`, а в качестве второго – предварительно посчитанное значение разности `mu-delta`. Оба ФаП размещаются друг за другом на стеке[1]

[1] Здесь и далее для простоты рассматривается традиционный для Си способ размещения ФаП на стеке, начиная с самого левого параметра. Однако язык Си этим способом не ограничивается и поддерживает также обратный порядок размещения параметров – справа налево, характерный для языка Pascal. Поэтому на практике, не предпринимая специальных мер, относящихся к области системного программирования и выходящих за рамки данного пособия, нельзя предполагать тот или иной способ пересылки ФаП в функцию.

до передачи управления в тело функции. В момент входа в функцию на стеке следом за передаваемыми параметрами выделяется место под внутренние переменные функции `gaussian_random_number` и `i`, первая из которых инициализируется. По завершении работы функции вычисляется значение выражения `gaussian_random_number*norm`, которое размещается на месте первого передаваемого в функцию параметра (в нашем случае `n`). Таким образом, начиная с момента возврата функцией значения все ее внутренние переменные (как переданные в функцию через аппарат ФоП–ФаП, так и определенные в теле функции) теряются и на их месте может располагаться другая информация.

С этим связана одна из наиболее характерных для начинающих программистов тяжелых ошибок, рассмотренная ниже.

Например:

```
double* copy_array(double *a, int length_a)
{
double b[length_a];
for (length_a--;length_a>=0; length_a--) b[i]=a[i];
return &b[0]; /* ошибка */
}
```

Во-первых, далеко не все реализации языка позволяют задавать в определении массива его размер с помощью неконстантного выражения, поскольку это является нововведением языка. Во-вторых, функция возвращает указатель на массив, уже уничтоженный на момент выхода из нее, и именно данная ошибка – одна из наиболее «коварных». Последствия могут быть любыми: массив может быть случайно еще не затерт выкладываемыми на стек последующими данными, поэтому программа будет казаться работающей; данные в `b` могут быть уже испорчены новыми переменными и константами, размещенными на стеке.

Есть два способа вернуть корректный указатель из функции:

- ◊ место в памяти, на которое он указывает, выделено до входа в функцию;
- ◊ внутри функции применяется динамическое выделение памяти, которое не затрагивает стек, а выделяет фрагменты памяти в так называемой куче.

Если входных параметров может быть несколько, то выходной параметр бывает максимум один в отличие, например, от системы программирования Matlab. Это ограничение введено с целью повысить эффективность программ. Предположим, что можно было бы реализовать возвращение нескольких параметров в вызывающую функцию, как в описанном методе передачи входных параметров. На момент выполнения оператора `return` могли бы вычисляться все возвращаемые выражения и размещаться последовательно в стеке. Рассмотрим три возможных варианта.

Первый вариант. Если возвращаемые параметры выложить в вершину стека после всех внутренних переменных и констант функции, то при удалении внутренних переменных и констант эта область стека станет недоступной из вызывающей функции, поскольку стек не допускает «дыр» в адресном пространстве, либо вызывающая функция в отличие от вызываемой должна заниматься обработкой стека при возвращении в нее, а для этого она должна «знать» особенности реализации конкретной вызываемой функции, что сложно и непрактично.

Второй вариант. Если возвращаемые значения записать в стек поверх внутренних переменных функции, это сильно усложнит генерацию кода функции и снизит ее эффективность, так как возвращаемые значения могут вычисляться с использованием внутренних переменных, которые нельзя преждевременно испортить, пока все возвращаемые значения не будут вычислены. Возникает вопрос: где и как их временно сохранять?

Третий вариант. Возвращаемыми значениями придется затереть область передаваемых в функцию параметров. Но нет гарантии, что для всякой функции набор передаваемых параметров занимает достаточно места. Кроме того, для получения возвращаемых значений могут потребоваться передаваемые в функцию значения и возникает та же проблема, что и при размещении возвращаемых значений на месте внутренних автоматических переменных и констант.

До сих пор не найден изящный и эффективный способ организации возвращения функцией множества параметров через единственный стек. Но в этом нет серьезной необходимости.

В заключение напомним, что можно попросить компилятор о передаче небольшой, помещающейся в одиночный регистр порции данных в функцию через регистр процессора, указав для данного ФоП функции спецификатор хранения `register`. Если компилятор последует этой рекомендации, все приведенные выше операции со стеком по отношению к этому параметру выполняться не будут, что позволяет уменьшить время вызова функции и является неплохим методом оптимизации.

7.7. Передача в функцию и получение из функции данных

Из нескольких способов организации обмена большими блоками данных между функциями самый простой – определить массив или структуру как глобальную в рамках программы или файла. Высокая эффективность этого метода обусловлена отсутствием манипуляций со стеком, но имеет очень неприятную обратную сторону. Программы, использующие такую передачу, сложно писать, отлаживать и очень трудно объединять в большие прикладные пакеты. Как показывает практический опыт, выигрыш в рамках отдельных функций ведет к неэффективности и ненадежности большой программы в целом, поскольку программистам приходится прикладывать много усилий для объединения таких функций. Данная практика обоснованно считается неприемлемой, и к ней рекомендуется прибегать только при крайней необходимости.

Рассмотренный в § 7.6 механизм обмена данными через стек в момент выхода из функции удаляет все размещенные на стеке входные параметры. Это удобно для программиста, так как можно не беспокоиться о том, что функция случайно повредит переменные из внешней программы. Такое повреждение просто невозможно, поскольку внешние переменные и константы (кроме глобальных) в функции не видны, а передаются лишь их копии, изменение которых внутри вызываемой функции не сказывается на значениях их оригиналов в вызы-

вающей функции. Важным следствием является то, что, если не прибегать к глобальным переменным, кроме единственного возвращаемого значения, из функции во внешнюю программу передать ничего нельзя.

Возникает вопрос, затрагивает ли это ограничение возможность писать сложные функции, интенсивно обменивающиеся данными. Наличие в языке Си указателей позволяет решить эту проблему просто и изящно. Чтобы это продемонстрировать, рассмотрим две характерные ситуации.

1. В функцию требуется передать массив или структуру, которую внутри этой функции следует изменить. Для этого достаточно передать в функцию указатель на начало[1] этого массива в совокупности с данными о его размере или указатель на структуру. Хотя изменения копии указателя внутри функции не приведут к его изменениям вовне, меняя содержимое памяти, адресуемой этим указателем, можно менять содержание переданного объекта, поскольку из вызывающей и вызываемой функции адресуется одна и та же память.

Например:

```
#include <stdio.h>
#define N 10 /* задание константного размера массива */

void reverse_order(float *a, int n)
{
float tmp; int n1;
if (a)
  { for (n--, n1=0; n>n1; n--, n1++)
    { tmp=a[n]; a[n]=a[n1]; a[n1]=tmp; }}
return;
}

int main()
{
float b[N]; int i;
for (i=0; i<N; ++i) b[i]=((float)i)/100.0;
/* передача указателя на начало массива и его длины в
функцию */
reverse_order(&b[0], N);
```

[1] Или на любое другое фиксированное место массива, например, на его последний элемент.

```
/* вывод результата изменения порядка элементов в массиве
на обратный */
for (i=0; i<N; ++i) printf ("\n b[%d]=%g",i,b[i]);
}
```

2. Если требуется передать в функцию структуру или массив и «обезопасить» его от изменений внутри функции, следует писать в прототипе и определении функции перед именем типа соответствующего данного модификатор `const`.

Например:

```
void print_array(const float *a, int n);
```

Конечно, модификатор не изменит автоматически алгоритм функции для защиты массива от вмешательства. В случае попытки внутри функции поменять что-либо в объекте, передаваемом таким способом, компилятор сообщит об ошибке. Зная, что вызываемая функция меняет значение передаваемого объекта в силу конструкции алгоритма, и желая сохранить в вызывающей функции исходные значения, в ней делают копию объекта и передают ее по указателю, а после выхода из функции эту копию можно уничтожить.

Например:

```
#include <stdio.h>

/* функция сортирует массив в месте его хранения */
void bubble_sort(long long *a, int n)
{
int k; char swaps;

/* применен трюк с обменом пары целочисленных переменных
значениями без вспомогательной переменной, основанный на
свойстве исключающего ИЛИ */
do
  {
  for (swaps=0, k=1; k<n; ++k)
    if (a[k]<a[k-1])
      { a[k-1]^=a[k]; a[k]^=a[k-1]; a[k-1]^=a[k];
      swaps=1; }
  }
```

```
while (swaps);
return;
}

const long long L=10;

int main()
{
long long x[L], x_copy[L], l;
/* массив и его копия для сортировки заполняются числами
*/
for (l=0;l<L;l++) x[l]=x_copy[l]=-10*l;
/* сортировка применяется к копии массива */
bubble_sort(x_copy,l);/* указатель на начало и длина */
printf("\n");
for (l=0;l<L;l++)
  printf("\n x[%lld]=%lld; x_copy[%lld]=%lld",
    l,x[l],l,x_copy[l]);
}
```

Таким образом, стандартный способ передачи в функцию больших объектов состоит в использовании указателя. Аналогично решается задача возвращения из функции крупных структур и массивов. Если функция ограничивается изменением переданной в нее по указателю структуры данных, проблема решается автоматически – этот указатель даже не нужно возвращать, так как он уже имеется в вызывающей программе. При необходимости вернуть несколько указателей, не переданных в функцию извне, а вычисленных внутри функции, прибегают к одному из трех корректных и безопасных решений:

1. В функцию передаются указатели на указатели (так называемые двойные указатели). Поскольку содержимое адресуемой указателем памяти можно модифицировать, по этим указателям внутри функции записываются значения возвращаемых указателей. Определим функцию, возвращающую таким образом указатели на наименьший и наибольший элементы в массиве.

Например:

```
#include <stdio.h>

int find_min_and_max_1(unsigned sz, double *ar,
double **p_min, double **p_max)
```

```
{
unsigned s;

if (ar==0) return -1; /* передан нулевой указатель */
if (sz<1) return 1; /* массив пустой */
for (s=1, *p_min=*p_max=ar; s<sz; ++s)
  {
  if (*(ar+s)<**p_min) *p_min=ar+s;
  if (*(ar+s)>**p_max) *p_max=ar+s;
  }
return 0; /* нормальное завершение */
}

int main()
{
double vector[]={0.1, 10, -5.6, 13.2, 20.1, 200.09,
-59.7, 2.7}, *pointer_to_min, *pointer_to_max;
int rv;

if (rv=
    find_min_and_max_1(sizeof(vector)/sizeof(double),
      vector, &pointer_to_min, &pointer_to_max))
  printf("\n код ошибки %d",rv);
else /* нормальное завершение функции */
  printf("\n min(vector)=%lg max(vector)=%lg",
     *pointer_to_min, *pointer_to_max);

return 0;
}
```

Здесь `p_min` и `p_max` – указатели на указатели на `double`; `*p_min` и `*p_max` – указатели на позиции в векторе, в которых находятся соответственно минимальный и максимальный элементы; `**p_min` и `**p_max` – сами минимальный и максимальный элементы непосредственно в памяти вектора (а не их копии).

2. Несколько изменяемых указателей логично объединить в структуру, единственный указатель на которую передается.

Например, переделанная таким образом предыдущая программа:

```
#include <stdio.h>

typedef struct {double *pmin; double *pmax; }
```

```
   st_extrema;

int find_min_and_max_2(unsigned sz,
  double *ar, st_extrema *p_extrema)
{
unsigned s;

if (ar==0) return -1; /* передан нулевой указатель */
if (sz<1) return 1; /* массив пустой */
for (s=1, p_extrema->pmin=p_extrema->pmax=ar; s<sz; ++s)
  {
  if (*(ar+s)<*p_extrema->pmin) p_extrema->pmin=ar+s;
  if (*(ar+s)>*p_extrema->pmax) p_extrema->pmax=ar+s;
  }
return 0; /* нормальное завершение */
}

int main()
{
double vector[]={0.1, 10, -5.6, 13.2, 20.1, 200.09,
-59.7, 2.7};
st_extrema extrema;
int rv;

if (rv=find_min_and_max_2(
  sizeof(vector)/sizeof(double), vector, &extrema))
  printf("\n error with code %d",rv);
else
  printf("\n min(vector)=%lg max(vector)=%lg",
  *(extrema.pmin), *(extrema.pmax));

return 0;
}
```

3. Задачу выделения памяти можно делегировать самой вызываемой функции.

Например, версия программы поиска указателей на минимальный и максимальный элементы вектора:

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

typedef struct {double *pmin; double *pmax; }
```

```
  st_extrema;

st_extrema* find_min_and_max_3(unsigned sz, double *ar)
{
st_extrema *p_extrema;
unsigned s;

if (ar==0 || sz<1) return 0; /* передан нулевой указатель
или массив пустой */
if (!(p_extrema=
  (st_extrema*)malloc(sizeof(st_extrema))))
  return 0;/* не удалось динамически выделить память */

for (s=1, p_extrema->pmin=p_extrema->pmax=ar; s<sz; ++s)
  {
  if (*(ar+s)<*p_extrema->pmin) p_extrema->pmin=ar+s;
  if (*(ar+s)>*p_extrema->pmax) p_extrema->pmax=ar+s;
  }
return p_extrema; /* нормальное завершение */
}

int main()
{
double vector[]={0.1, 10, -5.6, 13.2, 20.1, 200.09,
-59.7, 2.7};
st_extrema *p_extrema;
int rv;

if (!(p_extrema=find_min_and_max_3(
  sizeof(vector)/sizeof(double), vector)))
  printf("\n error with code %d",rv);
else
  {
  printf("\n min(vector)=%lg max(vector)=%lg",
  *(p_extrema->pmin), *(p_extrema->pmax));
  free(p_extrema); /* необходимо освободить память */
  }

return 0;
}
```

Последняя версия требует повышенной внимательности программиста. С одной стороны, вне зависимости от возмож-

ного хода вычислений необходимо освободить взятую из кучи память ровно столько раз, сколько она была выделена. Отсутствие освобождения динамической памяти приводит к ее неоправданной консервации и накоплению в программе, называемых *утечкой памяти* (англ. memory leakage), которая может обусловливать постепенное замедление работы операционной системы и в конечном счете крах программы. Подобную ошибку сразу заметить крайне затруднительно. С другой стороны, более чем однократное освобождение одного и того же блока (как говорят, *пула*) динамически выделенной памяти или попытка освободить выделенную на стеке автоматическую память так, как будто она была взята из кучи, вероятнее всего приведет к мгновенному аварийному завершению программы. Кроме того, постоянное выделение и освобождение даже небольших фрагментов оперативной памяти крайне трудоемко для операционной системы и может существенно замедлять выполнение программы. Динамическое выделение памяти оправданно лишь тогда, когда необходимо получить большой фрагмент памяти, размер которого заранее предсказать нельзя.

7.8. Передача многомерных массивов в функцию

В Си намеренно не предусмотрена возможность переслать в функцию массив «как есть» по значению, выложив его целиком на стеке, а можно передать лишь указатель на него. Однако это ограничение не затрагивает практические возможности языка, поскольку выгрузка больших массивов на стек связана с копированием содержимого значительных областей памяти, очень трудоемка и, значит, непрактична. В процессе автоматического выделения массива обычно явно указываются все его размерности.

Например:

```
long double s[N][d+2*a][k*l+2];
```

Очевидно, такой массив уже не может быть описан единственным указателем на его начало, поскольку утрачивается ин-

формация о его размерностях. Поэтому передача многомерного массива в функцию не сводится к передаче указателя. Вследствие этого внутри такой функции даже при явной передаче в нее размерностей уже не работает самая удобная для программиста скобочная запись адресации нужного элемента.

Например, такая:

```
s[i][j][k]*=factor;
```

Это происходит потому, что скобочная запись скрытым образом использует для достижения нужного элемента массива информацию обо всех размерах массива, не попадающую внутрь функции.

Известны два классических способа адресовать элементы выделенных снаружи многомерных массивов внутри функции.

1. Сделать определение массива глобальным. Адресация при помощи квадратных скобок доступна, но перед программистом встают проблемы, связанные с организацией корректного одновременного доступа к массиву из различных функций в параллельных программах и защитой массива от случайного изменения, поскольку описание его как константного запрещает всем функциям его менять. Поэтому такое решение в общем случае рекомендовать нельзя.
2. Лучше всего использовать тот факт, что многомерный массив занимает сплошной фрагмент памяти. Это дает возможность мысленно «развернуть» его в длинный вектор и адресовать его элементы как элементы вектора.

Например:

```
#include <stdio.h>

#define I 2
#define J 3
#define K 4
#define L 5

char fill_array(double *ar,
  unsigned n1, unsigned n2, unsigned n3, unsigned n4)
{
unsigned i1, i2, i3, i4;
```

```
if (ar==0 || n1==0 || n2==0 || n3==0 || n4==0)
  return 1;/* нулевой указатель на массив или индекс */
for (i1=0; i1<n1; i1++)
  for (i2=0; i2<n2; i2++)
    for (i3=0; i3<n3; i3++)
      for (i4=0; i4<n4; i4++)
         *(ar+i4+n4*(i3+n3*(i2+n2*i1)))=
i4+10*i3+100*i2+1000*i1;
return 0; /* нормальное завершение */
}

int main()
{
double x[I][J][K][L];
unsigned i, j, k, l;

if (!fill_array(&x[0][0][0][0], I, J, K, L))
  {
  for (i=0; i<I; i++)
    for (j=0; j<J; j++)
      for (k=0; k<K; k++)
        for (l=0; l<L; l++)
          printf ("x[%u][%u][%u][%u]=%lg; ",
            i, j, k, l, x[i][j][k][l]);
  }
else
  printf ("\n Неверный вызов \"fill_array\"\n");
return 0;
}
```

Чтобы наглядно продемонстрировать правильность работы программы, здесь элементы массива инициализируются числами, получающимися сцеплением вместе (конкатенацией) цифр индексов соответствующих элементов. Для компактности списка передаваемых параметров и лучшего оформления данных параметры функции `ar`, `n1`, `n2`, `n3` и `n4` нередко объединяют в структуру.

Например, с полным сохранением смысла передачи в функцию указателя на массив, в заголовке функции вместо:

```
char fill_array(double *ar,
 unsigned n1, unsigned n2, unsigned n3, unsigned n4)
```

можно писать:

```
char fill_array(double ar[],
  unsigned n1, unsigned n2, unsigned n3, unsigned n4)
```

(т.е. в списке формальных параметров `double *ar` и `double ar[]` эквивалентны), а в функции `main` вместо

```
if (!fill_array(&x[0][0][0][0], I, J, K, L))
```

допускается упрощенная запись:

```
if (!fill_array((double*)x, I, J, K, L))
```

Последняя запись справедлива, поскольку имя массива — это указатель на его первый элемент и требуется лишь преобразовать тип указателя. Тем не менее полная форма `&x[0][0][0][0]` представляется более предпочтительной, так как напоминает о конструкции массива.

7.9. Способы адресации многомерного массива в функции

Приведенный в § 7.8 способ адресации элемента массива изнутри функции `fill_array`

```
*(ar+i4+n4*(i3+n3*(i2+n2*i1)))= значение;
```

напоминает схему Горнера экономного вычисления многочлена, что является следствием развернутой, менее вычислительно рациональной записи:

```
*(ar+i4+n4*i3+n4*n3*i2+n4*n3*n2*i1)= значение;
```

Такой метод корректен, эффективен и наиболее распространен, но его нельзя назвать наглядным и визуально приятным для программиста. Существует несколько малоизвестных способов внутри функции по-прежнему использовать адресацию внешних массивов при помощи квадратных скобок, два из которых приведены ниже.

Первый способ требует значительного локального выделения памяти внутри функции `fill_array`, но он глубоко вскрывает природу «массива массивов», иллюстрирует технику применения вспомогательных индексов и осуществляет наиболее быстрый доступ к элементам, правда, ценой трудоемкого подготовительного этапа. Если бы в функции имелся массив указателей `arp[n1][n2][n3]` на начала всех одномерных мас-

сивов размером n4, это помогло бы решить задачу. Действительно:

```
(arp[i1][i2][i3])[i4]
```

это и есть искомый элемент

```
x[i1][i2][i3][i4]
```

В силу того что операции индексации [] выполняются слева направо, (arp[i1][i2][i3])[i4] эквивалентно arp[i1][i2][i3][i4]. Трехмерный массив указателей arp создается вполне очевидным способом на основе указателя ar на начало массива и размерностей n2, n3, n4:

```
double *arp[n1][n2][n3];
for (i1=0;i1<n1;i1++)
  for (i2=0;i2<n2;i2++)
    for (i3=0;i3<n3;i3++)
      arp[i1][i2][i3]=ar+i1*n2*n3*n4+i2*n3*n4+i3*n4;
```

Теперь большой массив arp заполнен указателями на векторы, соответствующие крайней правой размерности массива x, и можно записать:

```
arp[i1][i2][i3][i4] = значение;
```

или

```
переменная = arp[i1][i2][i3][i4];
```

Например, полный исходный текст такой реализации функции fill_array:

```
char fill_array(double *ar,
  unsigned n1, unsigned n2, unsigned n3, unsigned n4)
{
double *arp[n1][n2][n3]; /* вспомогательный массив */
unsigned i1, i2, i3, i4;

if (ar==0 || n1==0 || n2==0 || n3==0 || n4==0)
  return 1;/* нулевой указатель на массив или индекс */

/* заполнение вспомогательного массива указателей */
for (i1=0;i1<n1;i1++)
  for (i2=0;i2<n2;i2++)
    for (i3=0;i3<n3;i3++)
      arp[i1][i2][i3]=ar+i1*n2*n3*n4+i2*n3*n4+i3*n4;
```

```
/* скобочная индексация посредством arp */
for (i1=0; i1<n1; i1++)
  for (i2=0; i2<n2; i2++)
    for (i3=0; i3<n3; i3++)
      for (i4=0; i4<n4; i4++)
        arp[i1][i2][i3][i4]=i4+10*i3+100*i2+1000*i1;
return 0; /* нормальное завершение */
}
```

Второй способ организации скобочной индексации сложнее понять, но проще программировать. Он обеспечивает довольно высокую эффективность адресации, не требует значительной вспомогательной памяти и сложного подготовительного этапа. Прежде чем его рассмотреть, найдем причину, почему без дополнительных усилий внутри функции нельзя в нашем примере писать: `ar[n1][n2][n3][n4]=значение;`

Внешний массив, указатель на начало которого передан в функцию `fill_array`, определен в функции `main` так:

```
double x[I][J][K][L];
```

Это определение следует читать рекуррентно, как обычно, справа налево:

1) `x[I][J][K][L]` — четырехмерный массив из элементов типа `double`;
2) `x[I][J][K]` — трехмерный массив из элементов типа `double[L]`, каждый из которых имеет размер `sizeof(double[L])=sizeof(double)*L;`
3) `x[I][J]` — двухмерный массив из элементов типа `double[K][L]`, каждый элемент размером `sizeof(double[K][L])=sizeof(double)*K*L;`
4) `x[I]` — одномерный массив из элементов типа `double[J][K][L]`, каждый элемент размером `sizeof(double[J][K][L])=sizeof(double)*J*K*L`.

Таким образом, навигация по самой левой размерности `I` использует скрытое знание размеров `J`, `K` и `L`, перемещение вдоль `J` требует «спрятанного» в тип данных знания размеров `K` и `L`, адресация вдоль `K` — всего лишь размера `L`, а смещение вдоль самой правой размерности не требует знания каких-либо размеров,

кроме того, что каждый элемент занимает `sizeof(double)` байт.

Информация относительно размеров массива вдоль размерностей программистом явно передана в функцию `fill_array`, остается только вручную выполнить соответствующее преобразование типа указателя `ar`, введя один вспомогательный указатель:

```
double (*ari)[n2][n3][n4];
ari=(double (*)[n2][n3][n4])ar;
```

Преобразование типа читается по общим правилам: «`ar` преобразуется к типу указателя на массив из `n2` штук массивов из `n3` массивов по `n4` элементов типа `double` в каждом», и его последствия практически не влияют на быстродействие программы, поскольку затрагивают преимущественно момент ее компиляции. Преобразование указателя «растолковывает» компилятору то, что он «знал» в функции `main`, но «забыл» при входе в другую функцию. Указатель соответствующего типа нельзя сразу передать в функцию `fill_array` по следующей причине. Тип переменной `ari` содержит конкретные числовые данные о размерах массива. Тип параметра обычной функции не может быть изменяемым, поскольку это противоречит идее фиксированного прототипа функции. Если бы функция была написана так, что принимала параметр типа

```
double (*)[n2][n3][n4],
```

то `n2`, `n3`, `n4` должны быть константами, а это привязало бы функцию к конкретным значениям размеров передаваемого массива, лишив ее гибкости. Трюк с преобразованием типа указателя делает то же, но не имеет таких ограничений. Выражения типа

```
ari[i1][i2][i3][i4] = значение;
```

или

```
переменная = ari[i1][i2][i3][i4];
```

теперь доступны и корректны в новом варианте функции `fill_array`, полный текст которого приведен ниже:

```
char fill_array(double *ar,
  unsigned n1, unsigned n2, unsigned n3, unsigned n4)
{
```

```
double (*ari)[n2][n3][n4]; /* определение */
ari=(double (*)[n2][n3][n4])ar; /* инициализация ar */
unsigned i1, i2, i3, i4;

if (ar==0 || n1==0 || n2==0 || n3==0 || n4==0)
  return 1;/* нулевой указатель на массив или индекс */

/* скобочная индексация посредством ari */
for (i1=0; i1<n1; i1++)
  for (i2=0; i2<n2; i2++)
    for (i3=0; i3<n3; i3++)
      for (i4=0; i4<n4; i4++)
        ari[i1][i2][i3][i4]=i4+10*i3+100*i2+1000*i1;
return 0; /* нормальное завершение */
}
```

7.10. Функции с переменным числом параметров

Иногда требуется функция очень гибкая в том смысле, чтобы она была способна корректно принимать и обрабатывать переменное число входных параметров (англ. variable arguments, variadic function). Наиболее известным примером таких функций является `printf` из стандартной библиотеки. У нее есть лишь один обязательный параметр – строка формата (кстати, вопреки распространенным представлениям она не обязана быть строковой константой). Важно, что эта форматная строка указывает не только как оформлять при печати выражения, но также, сколько и каких параметров (печатаемых величин) далее следует в качестве необязательных параметров. Их столько, сколько одиночных символов процента встретится в форматной строке. Это число может быть произвольным, и до момента передачи функции ФаП оно неизвестно, поэтому должен существовать механизм сообщения функции числа параметров, передаваемых ей каждый конкретный раз, и типов этих параметров. Это необходимо знать, чтобы правильно извлечь дополнительные параметров из стека. Использование глобальных переменных для данной цели технически возможно, но крайне нежелательно, поскольку такие функции сложно отла-

живать, сопровождать и встраивать в программу. Поэтому следует осуществлять передачу такой информации посредством одного из обязательных параметров функции.

При написании функции с переменным числом параметров можно условно выделить четыре шага описания и работы с необязательными параметрами, которые ее отличают от обычной функции.

1. Один или несколько обязательных параметров должны содержать точную информацию о числе необязательных параметров, поскольку не существует иного стандартного способа сообщить это функции. Кроме того, должна присутствовать информация о типах параметров, которая может быть явной или неявной. Так, в библиотечной функции `printf` это выполняется посредством явного указания форматов после символа `%`. Иногда требуется дополнительное преобразование типов к «более стандартным» типам, о котором пойдет речь ниже. Перед определением в файле первой из функций с переменным числом параметров в него включается соответствующий заголовочный (англ. header, в литературе называемый также включаемым, англ. included) файл

```
#include <stdarg.h>
```

Факт, что данная функция поддерживает переменное число параметров, программист отмечает указанием запятой и трех точек после имени последнего обязательного параметра в списке перед закрывающей круглой скобкой. Это делается и в прототипе, и в определении функции.

```
[спецификатор] тип_возвращаемого_значения имя_функции
    (список_обязательных_передаваемых_параметров, …)
```

2. В функции определяется переменная – указатель на очередной необязательный параметр в списке ФоП. Ее тип определен в `stdarg.h`.

```
va_list имя_указателя_на_необязательные_параметры;
```

Традиционно этот указатель называют `arg_ptr`, хотя допустимы и другие имена. Для краткости будем его далее назы-

вать arg_ptr. По выполняемой роли этот указатель сопоставляется указателю стека.

3. Указатель arg_ptr устанавливается на первый необязательный параметр в списке посредством макроса.

```
va_start(имя_указателя_на_необязательные_параметры,
имя_последнего_обязательного_параметра);
```

Во втором параметре макроса указывается имя последнего обязательного параметра в списке (после которого в определении и прототипе функции стояли запятая и многоточие). В результате выполнения макроса указатель стека устанавливается сразу за концом этого параметра, поскольку компилятор способен легко вычислить это место, зная число и размер обязательных параметров. Макрос va_start ничего не возвращает.

4. Из стека извлекается значение очередного необязательного параметра, тип которого задан вторым параметром макроса va_arg. Чтение производится по адресу, заданному указателем arg_ptr.

```
имя_переменной=va_arg(
имя_указателя_на_необязательные_параметры,
тип_считываемого_параметра);
```

Сразу после считывания параметра происходит приращение значения указателя arg_ptr на длину использованного параметра функции (длина типа определяется вторым параметром макроса). Таким образом, arg_ptr перемещается по стеку дальше, указывая на следующий параметр вызываемой функции, если он есть. Макрос va_arg используется в общей сложности столько раз, сколько необходимо для извлечения всех нужных программисту параметров вызываемой функции. Ни в прототипе функции, ни в ее определении невозможно указать типы необязательных параметров. Это предусмотрено специально, чтобы программист мог сделать эти типы переменными, задаваемыми в момент вызова функции. Важно лишь, чтобы во второй параметр макроса va_arg попадали правильные значения типа. Механизм реализации программист выбирает по своему усмотрению.

5. Требования выбрать со стека все необязательные параметры, переданные функции, строго говоря, нет, можно удовлетвориться несколькими первыми параметрами, а остальные игнорировать. По завершении считывания всех нужных необязательных параметров необходимо закрыть список параметров:

```
va_end(имя_указателя_на_необязательные_параметры);
```

При этом указатель стека возвращается в корректное исходное положение: он указывает на начало первого обязательного параметра функции, что гарантирует правильный выход из функции. Если макрос `va_end` забыт, возможна ошибка при передаче в точке вызова функцией возвращаемого значения в программу и даже повреждение стека.

Ниже приведен полный пример определения и вызова функции с переменным числом параметров, позволяющей перечитывать параметры несколько раз:

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

/* Шаг 1. В данной демонстрационной функции число необязательных параметров передается явно через "mandatory_argument2", а их типы для простоты фиксированы */
int variadic_function(
  int mandatory_argument1, /* число проходов считывания необязательных параметров (это можно делать сколько угодно раз */
  unsigned mandatory_argument2, /* число необязательных параметров */
  char the_last_mandatory_argument,...)
{
double optional_argument1_double;
float optional_argument2_float;
char optional_argument3_char;
char* optional_argument4_char_ptr;

/* Шаг 2. Определение переменной-указателя на необязательный параметр в списке */
va_list arg_ptr;

while (mandatory_argument1--) /* счетчик проходов считывания необязательных параметров */
```

```
  {
  if (mandatory_argument2)
    {
    /* Шаг 3.В нашем случае последний обязательный
    параметр называется "the_last_mandatory_argument"*/
    va_start(arg_ptr, the_last_mandatory_argument);

    /* Шаг 4. Первый необязательный параметр имеет тип
    double */
    optional_argument1_double=va_arg(arg_ptr, double);

    printf("\nпервый необязательный аргумент имеет тип\
 double. Его значение=%g",optional_argument1_double);

    if (mandatory_argument2>=2)
      {
      optional_argument2_float=va_arg(arg_ptr,double);
/* ВАЖНО: в качестве второго параметра макроса нельзя пи-
сать "float", поскольку при вызове функции происходит не-
явное стандартное преобразование к "double". В данном
примере из соображений наглядности не реализован механизм
явной передачи типа необязательного параметра в функцию.
Мы просто предполагаем, что первый и второй необязатель-
ные параметры имеют тип "double", а третий "int" */
      printf("\nвторой необязательный аргумент имеет\
 тип float вне функции и double внутри функции. Его\
 значение=%g", optional_argument2_float);
      }

    if (mandatory_argument2>=3)
      {
      optional_argument3_char=va_arg(arg_ptr, int);
      printf("\nтретий необязательный аргумент имеет\
 тип char при вызове функции и int внутри функции\
  вследствие неявного преобразования. Его значение=%d",
optional_argument3_char);
      }

    if (mandatory_argument2>=4)
      {
      optional_argument4_char_ptr=
        (char*)va_arg(arg_ptr, char*);
      printf("\nчетвертый необязательный аргумент\
 имеет тип char*. Это строка=\"%s\"",
```

```
      optional_argument4_char_ptr);
      }

    if (mandatory_argument2>4)
      printf("\nфункция вызвана более чем с 4\
 параметрами; только первые 4 аргумента будут\
 присвоены внутренним переменным");

    /* Шаг 5. Закрытие указателя на параметры */
    va_end(arg_ptr);

    printf("\nпосле того как все параметры прочитаны,\
 можно реинициализировать механизм и прочесть все\
 необязательные параметры еще столько раз, сколько\
 требуется");
    }
  else
    printf("\nни одного необязательного параметра не\
 передано");
  }

return 0;
}

int main()
{
int mandatory_argument1=2;
unsigned mandatory_argument2=4;
char the_last_mandatory_argument=123,
optional_argument4_char_ptr[]="Я - строка,\
 переданная из функции main";

variadic_function(
  mandatory_argument1,
  mandatory_argument2,
  the_last_mandatory_argument,
  123.456789,
  1234.56789,
  12,
  optional_argument4_char_ptr);

return 0;
}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Есть ли ошибка в следующем фрагменте кода:
```
int *f1(int*a, short *s)
{ int *tmp, tmp=a+s; return tmp; }
```
Если ошибки нет, аргументируйте свой ответ. Если ошибка есть, исправьте ее.
2. Имеется ли ошибка в данной функции:
```
int *f2(int*a, short s)
{int tmp; tmp=*(a+s); return &tmp;}
```
Если ошибки нет, аргументируйте свой ответ. Если ошибка есть, исправьте ее.
3. Как быстро убедиться в том, что значение, передаваемое в большую функцию по указателю, там случайно не меняется?
4. Реализуйте функцию, эффективно вычисляющую все возможные скалярные произведения четырех векторов: `ab`, `ac`, `ad`, `bc`, `bd`, `cd`.
5. Вы – автор функции, в которую передана прямоугольная матрица, требуемая внутри функции только в транспонированном виде. Обязательно ли ее явно транспонировать при помощи двух вложенных циклов? Почему?
6. Запрограммируйте функцию доступа к элементам массива `long double a[K][L][M]` с контролем выхода за границы диапазона индексов. Функция должна давать возможность читать и менять значения элементов массива `a`.
7. Запрограммируйте передачу двух указателей на `double` и двух беззнаковых целых типа `unsigned` из функции посредством возврата структуры.
8. Можно ли извне функции проверить, не вернула ли она указатель на внутреннюю автоматическую переменную? Если нет, то почему? Если да, то как?

ГЛАВА 8 ФУНКЦИИ: ВАЖНЕЙШИЕ ЧАСТНЫЕ СЛУЧАИ

8.1. Главная функция `main`

При наличии в программе нескольких функций должно существовать средство указать, какую из них операционная система должна запустить при старте программы. В языке Си эта функция должна быть единственной и носить предопределенное имя `main` (англ. главный). Поскольку эту *главную* функцию вызывает операционная система, которая не обязана «знать» особенностей ее запуска, то именно `main` должна отвечать ряду стандартных требований. В о - п е р в ы х, она не может иметь произвольное возвращаемое значение; стандарт предписывает возвращаемое значение типа `int`[1]. В о - в т о р ы х, передаваемое в систему конкретное число в соответствии с имеющимся соглашением свидетельствует об успешности и причине завершения программы: нулевое значение символизирует успешное завершение, а ненулевое кодирует тот или иной вид ошибки или проблемы. Большинство операционных систем вполне терпимо относится к отсутствию возвращаемого значения:

```
void main() { … return; }
```

хотя не рекомендуется отклоняться от стандартного `int`. Сложнее обстоит дело с ФоП-ФаП аппаратом главной функции программы `main`. У нее могут быть два или три передаваемых значения или не быть их вовсе. В общем случае:

```
   int main ([int argc, char *argv[][, char
*argp[]]])
   { … return возвращаемое_значение; }
```

Первые два из этих трех параметров сообщают в функцию параметры *командной строки*. Даже если графический интер-

---

[1] Предыдущие реализации систем программирования разрешают и другие типы возвращаемого значения, но этой рискованной практики стоит избегать.

фейс операционной системы показывает пользователю программу в виде пиктограммы (иконки), в действительности запуск программы происходит из командной строки. Это можно представить себе как наличие системного приглашения, в котором можно набрать имя программы и, возможно, через пробелы перечислить в текстовом виде параметры, передаваемые главной функции программы. Их может быть произвольное число. Нажатие клавиши Ввод приводит к запуску программы с этими параметрами. Увидеть командную строку и запустить из нее какую-либо установленную программу можно в Microsoft Windows посредством меню Пуск→Выполнить, а в операционных системах Unix и Linux открыв окно терминала.

Первый целочисленный параметр `argc` сообщает общее число параметров командной строки при запуске, включая само имя исполнимого файла программы. Параметр `argv` является массивом из ровно `argc` указателей, и каждый из них адресует константную символьную строку, размер которой можно определить, как обычно, – по положению концевого нулевого символа `'\0'`, например с помощью функции `strlen` стандартной библиотеки Си. Третий необязательный параметр `argp` устроен аналогично `argv`. Каждая символьная строка `argp` содержит одну системную переменную, определяющую окружение запущенной программы. *Переменные окружения* (англ. environment variables) сообщают программе некоторые сведения о конфигурации аппаратной части компьютера, операционной системы и программной среды, например число процессорных ядер, доступных для использования, директории, в которых следует искать файлы определенного типа (системные утилиты, библиотеки и т.д.), и многое другое. Поскольку аналог параметра `argc` для массива `argp` отсутствует, число строк определяется по следующему признаку: последний элемент `argp` содержит нулевой указатель, следовательно, перед использованием всякого указателя из вектора `argp` его необходимо проверить на нулевое значение.

Например, главная функция выводит на экран все передаваемые в нее параметры:

```
#include <stdio.h>

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

```
{
int k;

for (k=0;k<argc;k++)
  printf("\nпараметр командной строки номер\
%d это \"%s\"",k+1,argv[k]);

k=1;
while (*argp)
  {  printf ("\nпеременная окружения номер\
%d такова \"%s\"",k,*argp); argp++; ++k; }
printf("\n");
return 0;
}
```

Если эту программу откомпилировать в исполнимый файл с именем `command_line_parametrs.exe`, при вызове ее при помощи командной строки вида:

```
D:\>command_line_parametrs.exe  parm1  16  'a  string
parameter'
```

параметры `argc, argv` и `argp` приобретут в памяти примерный вид:

```
argc=4

argv[0]="command_line_parametrs.exe" (строка завершена '\0')
argv[1]="parm1" (строка завершена '\0')
argv[2]="16" (строка завершена '\0')
argv[3]="a string parameter" (строка завершена '\0')
argv[4]=NULL (присутствует в некоторых реализациях, но необязательно, поскольку число элементов массива задается значением
argc)
argp[0]="!C:=C:\Program Files\Far" (строка завершена '\0')
argp[1]="ALLUSERSPROFILE=C:\Documents and Settings\All
Users" (строка завершена '\0')
argp[2]="PATH=/cygdrive/c/Program Files/Far" (строка завершена '\0')
argp[3]=NULL (обязательно у последнего элемента массива, так как
нет другого способа определить их число).
```

Адресуемую `argv` и `argp` память ни в коем случае не следует освобождать — этим занимается операционная система.

8.2. Рекурсивные функции

Существует довольно широкий и важный класс алгоритмов, реализация которых требует использования рекурсии. Рекурсия тесно связана с рекуррентностью в математике. Рекуррентным называется выражение окончательного результата, достижимого, например, на N-м шаге, через промежуточные результаты на предыдущих $(N-i)$-х шагах, $i \geq 1$. Если подобная зависимость может быть прослежена вплоть до исходных данных, это позволяет автоматически пошагово продвигаться вперед:

- сначала от искомого решения к входным данным (*прямой ход рекурсии*);
- затем от входных данных к результату (*обратный ход рекурсии*), при помощи рекурсии.

Для того чтобы запрограммировать рекурсию, выполняют следующие шаги:

1) выделяют набор переменных, исчерпывающе характеризующих состояние алгоритма на любом шаге;
2) получают зависимость состояния алгоритма на $(i+1)$-м шаге от предыдущих шагов (для простой рекурсии достаточно проследить связь с i-м шагом) и реализуют ее в виде рекурсивной функции (или нескольких функций в более сложных случаях);
3) реализуют правила останова алгоритма, гарантирующие конечное число шагов (как говорят, конечную *глубину рекурсии*);
4) вызывают рекурсивную функцию на исходных данных с целью получить требуемый результат.

Все остальное компьютер осуществит автоматически.

Например, следующая функция проверяет, является ли заданное число факториалом некоторого натурального числа:

```
#include <stdio.h>

int is_it_a_factorial(
  unsigned long x, unsigned long divisor)
{
unsigned int quotient=x%divisor;
if (quotient)
  return 0; /* x не является факториалом */
```

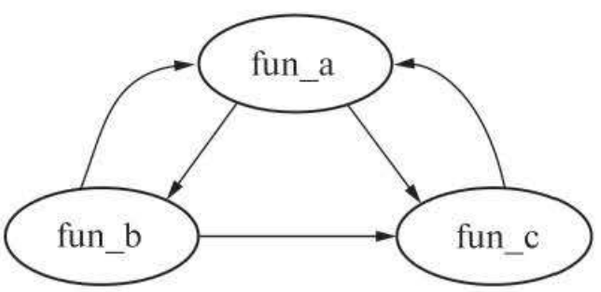
```
if ((quotient=x/divisor)==1L)
  return 1; /* x является факториалом */
/* решение не готово: рекурсивно вызываем функцию */
return (is_it_a_factorial(quotient,++divisor));
}

int main()
{
unsigned long z=2*3*4*5*6*7*8*9*10*11*12;
printf("\nz=%ld   это%s   факториал   некоторого   чис-
ла\n",z,is_it_a_factorial(z,1) ? "" : " не");

return 0;
}
```

При первичном вызове функции `is_it_a_factorial` проверяемое число передается в качестве параметра `quotient`. При всяком вызове выясняется, делится ли оно нацело на `divisor`, и при возможности принимается окончательное решение, которое через возвращаемое значение передается «наверх». Если на данном шаге это невозможно, функция идет «вглубь», вызывая саму себя. При этом обязательно уменьшается `quotient` и увеличивается `divisor`. Это гарантирует, что процесс завершится за конечное число шагов, а знание способа вычисления факториала определяет потребное число вызовов.

Возможности рекурсии не ограничиваются вызовом функции из себя. В рекурсию может быть вовлечено сколь угодно сложное множество функций, главное при этом — соблюдать разумную достаточность, гарантировать непременное завершение процесса и не запутаться. Так, функция `fun_a` может вызывать функции `fun_b` и `fun_c`. В свою очередь функция `fun_b` обращается к `fun_a` и `fun_c`, а `fun_c` — только к `fun_a`, что можно изобразить в виде *графа вызовов*:



Рекурсия бывает полезна в классических рекурсивных алгоритмах, при вычислениях с применением конечных автоматов.

8.3. Спецификатор хранения функции extern

Многие программные проекты столь масштабны, что неоправданно сосредоточивать в одном файле все необходимые декларации и определения констант, переменных и функций. Это и неудобно, потому что в такой монолитной программе затруднительно найти нужное место. Кроме того, над программным проектом обычно трудятся многие специалисты и им сложно постоянно объединять (как говорят, *синхронизировать*) результаты своей работы. Поэтому проект разбивают на отдельные файлы, обычно по принципу «одна функция—один файл» или «группа тесно связанных функций—один файл». Чтобы, обрабатывая какой-то исходный файл, компилятор узнал о существовании вызываемой функции, размещенной в другом файле, где-либо вне функций до первого вызова этой внешней функции размещают ее прототип, предварив его спецификатором `extern` (внешний, от англ. external):

```
extern прототип_внешней_функции
```

Как всегда, `прототип_внешней_функции` завершается точкой с запятой и должен в точности соответствовать определению этой функции по числу и типу передаваемых параметров и типу возвращаемого функцией значения. Чтобы не повторять эту конструкцию в каждом файле, где используется эта внешняя функция, данную декларацию выносят в заголовочный файл, имеющий обычно расширение `.h`, текст которого включают при помощи директивы `#include` препроцессора. Назначение спецификатора хранения `extern` для переменных, констант и функций одинаково. Его цель – «успокоить» компилятор, заверив его, что такая переменная, константа или функция действительно существует и определена в другом файле.

8.4. Спецификатор `static` для функций

Если не предпринимать дополнительных мер, функция, находящаяся в одном файле, «видна» из другого и ее можно оттуда вызывать. Это иногда нежелательно, например если два файла содержат одноименные вспомогательные функции, не вызываемые «снаружи» этих файлов. Компилятор по умолчанию пытается сделать эти функции доступными извне, чтобы к ним можно было обратиться при использовании в другом модуле прототипа со спецификатором `extern`. В свою очередь сборщик при компоновке программы для получения исполнимого файла обнаружит два определения функции (даже не важно, совпадают они или нет) и выдаст ошибку — «двойное определение функции». Но нет необходимости переименовывать одну из функций — достаточно перед прототипом и определением одной или обеих функций поместить спецификатор `static`:

```
static прототип_функции
```

Теперь гарантируется, что функции застрахованы от возможных обращений извне файла. Важно отметить, что действие `static` по отношению к функциям и данным различно: функции становятся локальными, а данные — локальными и «вечными». Но есть и общее: их видимость становится местной.

8.5. Встраиваемые функции и спецификатор `inline`

В случае вызова обычной функции все фактические параметры выкладываются на стеке в порядке слева направо, как они перечислены в списке формальных параметров, или в обратном порядке. Указатель стека на момент вызова указывает на начало первого (последнего) из параметров. При выходе из функции указатель стека перемещается в исходное положение (т.е. вновь указывает на начало первого (последнего) переданного в функцию фактического параметра) и ровно в это место на стек помещается возвращаемое значение, перетирая начало списка параметров функции. Вызывающая функ-

ция, зная, где размещается возвращенное значение, забирает его и использует. В том случае, если приходится передавать много параметров в очень короткую и простую функцию, затраты процессорного времени на работу со стеком могут превзойти полезную вычислительную работу в функции. Опыт программирования свидетельствует, что так происходит со многими маленькими функциями. Для уменьшения накладных расходов по работе со стеком в Си введен спецификатор `inline`. Если `inline` указан в описании прототипа функции и в определении функции перед типом ее возвращаемого значения, эта функция будет вызываться по-другому. Ее откомпилированный код будет автоматически встроен непосредственно в каждое место вызова. Можно сказать, что код *встраиваемой функции* автоматически подставляется в программу столько раз, сколько есть вызовов функции. Имеются следующие требования к применению `inline`:

1) недопустимо указывать спецификатор `inline` перед рекурсивной функцией, так как это могло бы привести к непомерному разрастанию кода; кроме того, как правило, наперед неизвестна глубина рекурсии;

2) если в прототипе функции встречается `inline`, в этом же модуле должно быть дано определение функции. Иначе компилятор «не будет знать», откуда брать тело функции, чтобы вставить в место вызова. Указатель на функцию, определенную в другом файле, для встраивания недостаточен, поскольку требуется полностью само тело функции;

3) следствие из требования 2: если функция определена как `inline`, она нигде в этом модуле не может быть объявлена как `extern`.

8.6. Назначение и общий синтаксис указателя на функцию

Одной из наиболее мощных возможностей языка Си является вызов функций по указателю. Действительно, если при помощи указателя можно обратиться к блоку данных, то целесообразно реализовать механизм и обращения по

указателю к блоку исполнимого кода – к функции. Определение и декларация указателя на функцию мало отличаются от прототипа функции, поскольку в обоих случаях необходимо описать:

- как называется функция или указатель на нее;
- параметры каких типов принимает;
- значение какого типа возвращает.

Пусть прототип функции записывается в самом общем виде:

```
[спецификатор] тип_возвращаемого_значения
имя_функции(список_передаваемых_параметров_прототипа);
```

Тогда определение и декларация указателя на функцию записываются путем добавления звездочки перед именем указателя, стоящим на месте имени функции, и заключения их в пару круглых скобок.

```
[спецификатор] тип_возвращаемого_значения
(*имя_указателя_на_функцию)
(список_передаваемых_параметров_прототипа);
```

Многими системами программирования поддерживается старый способ записи, в котором `список_передаваемых_параметров_прототипа` можно опустить:

```
[спецификатор] тип_возвращаемого_значения
(*имя_указателя_на_функцию)();
```

Не рекомендуется пользоваться старой записью, поскольку она не позволяет компилятору автоматически проконтролировать соответствие типов параметров вызова и предотвратить связанные с этим ошибки. Определение и декларация указателя на функцию различаются только спецификатором: в данном случае наличие спецификатора `extern` относится не к функции, а к указателю, сообщая, что именно указатель определен в другом файле. Таким образом, в определении слова `extern` быть не может, а в декларации оно имеется, что сближает указатель на функцию и прочие переменные и константы. В качестве примера определим `f` как указатель на функцию, принимающую два параметра – типа `unsigned` и типа `char*` и ничего не возвращающую:

```
void (*f)(unsigned, char*);
```

Если бы выражение `*f` не было заключено в придающие высокий приоритет фигурные скобки:

```
void *f(unsigned, char*);
```

определение читалось бы справа налево по общим правилам как «`f` — это функция, принимающая два параметра — типа `unsigned` и типа `char*` и возвращающая указатель на `void`». Таким образом, получился бы прототип другой функции, а не указатель на желаемую. Указатель на функцию присвоить переменной соответствующего типа очень просто:

```
имя_указателя_на_функцию=&имя_функции_подходящего_типа;
```

```
void print_string(unsigned n, char *s)
{
printf("Строка номер %u такова: \"%s\"\n",n,s);
return;
}

f=&print_string;
```

Чтобы вызвать функцию по указателю, достаточно его разадресовать и обратиться к результату разадресации как к функции, передав ей параметры.

Например:

```
[переменная_принимающая_возвращаемое_значение_функции=]
(*имя_указателя_на_функцию)
(список_передаваемых_в_функцию_параметров)
```

Так,

```
(*f)(12,"Попугай не сядет на кактус.");
```

приведет к такому же результату, как и

```
print_string(12,"Попугай не сядет на кактус.");
```

Подобно прочим указателям, указатель на функцию способен выступать в качестве параметра функции. В прототипе и определении такой функции полностью указывается тип передаваемого указателя.

8.7. Указатели на функции в массиве и структуре

Указатели на функции одинакового типа можно объединять в массив. Определение или декларация такого многомерного массива размерностью n в общем виде:

```
[спецификатор] тип_возвращаемого_значения
(*имя_массива_указателей_на_функции{[размер_массива_n]}^n)
(список_передаваемых_параметров_прототипа);
```

Так, определение вектора из 6 указателей функции такого же типа, что и f из § 8.6, записывается следующим образом:

```
void (*fa[6])(unsigned,char*);
```

Присваивание указателей элементам массива таково:

```
имя_массива_указателей_на_функции{[индекс_n]}^n =
&имя_функции_подходящего_типа;
```

Вызов функции происходит следующим образом:

```
[переменная_принимающая_возвращаемое_значение_функции=]
(*имя_массива_указателей_на_функции{[индекс_n]}^n)
(список_передаваемых_в_функцию_параметров)
```

Например:

```
fa[3]=&print_string;
(*fa[3])(100,"Футбол - древняя игра");
```

Массив указателей на функции – удобное и эффективное средство автоматизированного выбора нужной функции в зависимости от ситуации, определяющей индекс. Так, меню программ, как правило, обрабатывается указанным образом: все пункты пронумерованы и при выборе конкретного пункта вызывается функция с соответствующим номером, выполняющая требуемое действие.

Одиночный указатель на функцию или массив указателей на функцию может быть и полем структуры. Для этого в декларацию и определение структуры помещают рассмотренное выше определение указателя на функцию или массива указателей.

Например, указатель на функцию print_string из § 8.6 можно разместить в структуре struct_d особняком (поле b) или в массиве (поле c):

```
/* декларация типа struct struct_d и определение перемен-
ной d */
struct struct_d {
int a;
void (*b)(unsigned,char*);
void (*c[6])(unsigned,char*);
} d;
/* инициализация указателей */
d.b=&print_string; d.c[3]=&print_string;

/* вызов функций */
(*d.b)(1000,"Утонула ложка в банке земляничного \
варенья. (С) Радионяня");
(*d.c[3])(1100,"А в выходные опять ненастье.");
```

Важнейшим условием успеха в применении массивов функций является умение правильно прочесть написанное, возможно, весьма заковыристое. Например, допустима такая декларация типа:

```
double (*const [])(long, int*, ...)
```

Это вектор, состоящий из неопределенного числа константных указателей на функции, каждая из которых принимает два обязательных параметра: первый типа long, а второй — указатель на int, а также неопределенное число других параметров и возвращает double.

8.8. Преобразование типа указателя на функцию

Как и для всякого указателя, преобразование типа можно осуществить для указателя на функцию. Такое преобразование должно быть явным, чтобы компилятор мог точно проверить намерение программиста и предупредить о возможных ошибках. Если функцию вызовут неправильно (т.е. с недопустимым числом и типом параметров), она по меньшей мере неправильно отработает и скорее всего разрушит стек, что

означает крах программы. Если указатель на функцию был определен так:

```
тип_возвращаемого_значения_истинный
(*имя_указателя_на_функцию_для_истинного_типа)
(список_передаваемых_параметров_прототипа_истинный);
```

а его требуется присвоить указателю на функцию такого типа:

```
тип_возвращаемого_значения_преобразованный
(*имя_указателя_на_функцию_для_преобразованного_типа)
(список_передаваемых_параметров_прототипа_преобразованный);
```

пишут:

```
имя_указателя_на_функцию_для_преобразованного_типа=
(тип_возвращаемого_значения_преобразованный(*)
(список_передаваемых_параметров_прототипа_преобразованный))
имя_указателя_на_функцию_для_истинного_типа;
```

Данное преобразование типа является частным случаем обычного явного преобразования:

```
переменная_нового_типа=(новый_тип)переменная_старого_типа;
```

Такое сложное преобразование нужно, поскольку однотипные, унифицированные указатели можно поместить в массив, а потом, извлекая, восстанавливать их тип.

Например:

```
#include <stdio.h>

void function0(void)
{
printf("\nСпасибо, что меня вызвали. Ваша function0.");
return;
}

char function1(float a)
{
printf("\nСпасибо, что меня вызвали. Ваша function1.");
return ((a>=0.0) ? 1 : -1);
}

float function2(float b)
{
printf("\nСпасибо, что меня вызвали. Ваша function2.");
return ((b>0.0) ? -b : b);
}
```

```
double function3(float c, unsigned char d)
{
double tmp=1.0;
while (d) { tmp*=c; --d; }
printf("\nСпасибо, что меня вызвали. Ваша function3.");
return tmp;
}

int main()
{
void (*array_of_pointers_to_functions[])(void)=
{function0, /* тип функции точно соответствует */
(void (*)(void))function1, /* преобразование типа */
(void (*)(void))(&function2), /* преобразование типа */
(void (*)(void))0}; /* нулевой указатель "правильного типа" */
float f_ret;
signed char c_ret;

array_of_pointers_to_functions[3]=
(void(*)(void))(&function3);

(*array_of_pointers_to_functions[0])();
printf(" Я ничего не возвращаю, но и не беру ничего");

c_ret=(*(char(*)(float))
array_of_pointers_to_functions[1])(4);
printf(" Я вернула %d",c_ret);

f_ret=(*( (float(*)(float))
(*(array_of_pointers_to_functions+2)) ) )(-6);
printf(" Я вернула %g",f_ret);

printf("Я вернула %g",(*( (float(*)(float, unsigned char))
 (*(array_of_pointers_to_functions+3)) ) )(3.0,4));

return 0;
}
```

При внимательном чтении приведенного выше полного примера возникает вопрос: в функции `main` при инициализации `array_of_pointers_to_functions` написано:

`(void (*)(void))function1` .

Но по своей природе `function1` является функцией, а не указателем на функцию. Тем не менее программа работает. Дело в том, что в предыдущих версиях языка Си функция и указатель на функцию – одно и то же. Компилятор сам «подыгрывает» программисту, прощая такую «мелочь», как отсутствие взятия указателя или его разадресации. Это нечеткое поведение из соображений совместимости пока сохранилось и в новом стандарте языка Си, но не рекомендуется прибегать к такой манере программирования, поскольку легко запутаться в типах.

Некоторые компиляторы вообще не требуют приведения типа функции для присваивания указателю или последующего корректного ее вызова, поскольку вовсе не проверяют ее тип: все остается на совести программиста, в том числе проверка числа и типов параметров и сопряженные опасности.

Залог правильного написания конструкций с функцией – это умение прочесть получившееся выражение. Если оно осмысленно читается, значит, почти наверняка правильно написано. Правила чтения принципиально не отличаются от правил чтения констант и переменных и в развернутом виде таковы:

1) найти имя указателя или массива указателей на функцию. Оно, очевидно, отсутствует в выражении явного преобразования типа, и его место в этом случае можно отыскать по пустой или обнимающей звездочку паре круглых скобок слева от списка параметров функции;
2) если имя вместе со стоящей слева от него звездочкой заключено в круглые скобки, то это указатель;
3) если справа от имени имеется по крайней мере одна пара квадратных скобок, то это массив соответствующей размерности;
4) правее должна быть пара круглых скобок, заключающих список параметров (он читается слева направо, как обычно у функции);
5) левее имени функции (возможно, заключенного в круглые скобки) располагается тип возвращаемого значения;
6) левее возможны модификаторы;
7) самыми левыми могут быть спецификаторы хранения.

8.9. Указатель на функцию и typedef

Указатель на функцию является, пожалуй, наиболее сложным из возможных в Си типов, и понятно желание иметь способ записывать его проще. Это возможно с применением определения альтернативного имени типа typedef. Общая форма создания нового типа одиночного указателя на функцию такова:

```
typedef тип_возвращаемого_значения
(*короткое_имя_типа_указателя_на_функцию)
(список_передаваемых_параметров_прототипа);
```

Короткое имя для типа массива указателей на функцию создается аналогично:

```
typedef тип_возвращаемого_значения
(*короткое_имя_типа_массива_указателей_на_функции
{[размер_массива_n]}^n)
(список_передаваемых_параметров_прототипа);
```

К сожалению, в последнем случае приходится фиксировать размеры вдоль размерностей, поэтому такая конструкция на практике применяется нечасто. Гораздо удобнее задавать размеры массива не в месте создания типа, а далее, при определении конкретных переменных и констант.

```
[спецификатор] короткое_имя_типа_указателя_на_функцию
<имя_конкретного_указателя,
имя_конкретного_массива_указателей{[размер_массива_n]}^n>^m;
```

Здесь m – общее число определяемых объектов (имеется выбор, переменная или массив); n – размерность массива. Описанные возможности иллюстрируются простым завершенным примером.

Пример:

```
#include <stdio.h>

typedef struct mystruct_s { int a,b; } mystruct_t;

/* вводятся два новых типа указателей на функцию: они раз-
ные: первый для создания массива-хранилища указателей,
а второй соответствует функциям get_a и get_b */
typedef int (*fptr_storage)(void*);
typedef int (*fptr_indeed)(mystruct_t* x);
```

```
/* определяются две конкретные функции */
int get_a(mystruct_t* x) { return x->a; }
int get_b(mystruct_t* x) { return x->b; }

int main()
{
mystruct_t y={98, 99};

/* определяется и инициализируется массив указателей на
функции, при этом требуется явно привести тип указателей
на функцию с fptr_indeed к фиктивному fptr_storage */
fptr_storage
f[2]={(fptr_storage)(&get_a),(fptr_storage)(&get_b)};

/* функции вызываются непосредственно из массива; чтобы в
них правильно передавались параметры, необходимо выпол-
нить явное обратное преобразование типа указателя на
функцию с fptr_storage к fptr_indeed */
printf("f[0] gets %d; f[1] gets %d\n",
(*((fptr_indeed)f[0]))(&y),(*((fptr_indeed)f[1]))(&y));

return 0;
}
```

Мощность языка Си позволяет и передавать в функцию указатель на функцию, и возвращать его из функции. Однако в силу правил синтаксиса языка Си последнее напрямую сделать в общем случае не удается. При анализе прототипа такой функции или ее определения компилятор скорее всего не сможет различить тип указателя на функцию как возвращаемое значение и собственно тип описываемой функции, поскольку они имеют сходное написание. Здесь на выручку приходит `typedef`. С его помощью вводится тип указателя на нужную функцию, который затем подставляется на место возвращаемого значения другой функции. Такой подход иллюстрируется следующим полным примером.

Пример:

```
#include <stdio.h>

typedef double(*get_double_const)(char);
```

```
double get_e(char verbose)
{
if (verbose) printf("\n   вызвали \"get_e\"");
return 2.718281828;
}

double get_pi(char verbose)
{
if (verbose) printf("\n   вызвали \"get_pi\"");
return 3.141592654;
}

get_double_const f_selector(double(*f1)(char),
  double(*f2)(char),unsigned selector,char verbose)
{
if (verbose) printf("\n  вызвали \"f_selector\" с \
selector=%u",selector);
if (selector==0) return f1;
if (selector==1) return f2;
return 0;
}

int main()
{
double(*f)(char), ret_val;
unsigned selector; char verbose=1;

for (selector=0;selector<=1;selector++)
  {
  printf("\nтеперь selector=%u",selector);
  f=f_selector(&get_e,&get_pi,selector,verbose);
  ret_val=f(verbose);
  printf(", возвращаемое значение=%g",ret_val);
  }

return 0;
}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Имеется ли ошибка в такой декларации:
   `extern (int* (c*))(double, const double *);`

2. Можно ли внутри тела функции силами языка Си выяснить значение указателя на нее саму, если он в нее не передан?
3. Реализуйте счетчик уровня рекурсии рекурсивной функции. Неизбежно ли использование переменной со спецификатором `static` либо глобальной переменной?
4. Допустимо ли в вопросе 3 использовать счетчик со спецификатором `static` или глобальную переменную в условиях многопоточной программы, когда функция может быть вызвана одновременно в разных потоках выполнения (нитях) программы? Почему?
5. Можно ли сделать так, чтобы внутри некоторой функции была возможность вызывать друг за другом одноименные функции – внешнюю (`extern`) и строго локальную для данного файла (`static`)? Если да, то как? Если нет, почему?
6. Реализуйте функцию, в зависимости от истинности логического выражения возвращающую указатель на одну из двух функций с одинаковым прототипом. Обязательно ли оба этих указателя передавать в функцию-селектор?
7. Имеются 20 функций, каждая из которых подпадает под один из трех типов: `(int*) f1 (double, int); (double*) f2 (int*, int); (int) f3 (double*, int*);`
   Как организовать хранилище вектора указателей на эти функции с сохранением информации о способе вызова каждой из них?
8. Имеются ли ошибки в следующей декларации:
```
((unsigned *a)(*b)(unsigned*c))(*d),
((unsigned *e)(*f)(unsigned *g),
 (unsigned *h)(*i)(unsigned *j));
```
   Если имеются, исправьте их. Что означает декларация?

ГЛАВА 9 СТАНДАРТНАЯ БИБЛИОТЕКА ФУНКЦИЙ

9.1. Назначение и организация стандартной библиотеки

Строго говоря, стандартная библиотека не входит в стандарт ядра языка Си. Однако без подобной библиотеки программы лишены многих полезных возможностей, например инструментов ввода-вывода. Программист имеет возможность отказаться от применения стандартной библиотеки и использовать более мощные средства узкого назначения, предоставляемые той или иной операционной системой или средой разработки. Тем не менее знание возможностей стандартной библиотеки чрезвычайно полезно, поскольку она распространена повсеместно и дает ключ к пониманию других, более специализированных библиотек. Традиционно выделяют следующие довольно замкнутые группы наиболее употребительных функций:

- работа с динамически выделяемой памятью;
- операции со строками текста;
- файлы и поддержка базовых устройств ввода-вывода;
- математические функции.

Современные реализации стандартной библиотеки этим функционалом не ограничиваются. Достаточно упомянуть:

- средства автоматизации проверки логики программ и улучшенной диагностики ошибок;
- поддержку комплексных чисел и операций над ними;
- функции работы с датой и временем;
- расширенную поддержку различных кодировок символов, включая Unicode;
- работу с сигналами, средствами многопоточности и синхронизации, управления процессами и обеспечения атомарности (неразрывности) исполнения;
- управление выравниванием в сложных объектах.

Последние вопросы, с одной стороны, очень обширны, а с другой – весьма специфичны, поэтому в данном пособии не рассматриваются.

Чтобы вызвать какую-либо функцию стандартной библиотеки, следует в исходный текст программы включить заголовочный файл с ее прототипом.

Например:

```
#include <math.h>
…
double x, y,z, k;
…
y=sin(x)*exp(-k*z);
```

Заголовочные файлы также содержат полезные макроопределения и константы, к сожалению, нередко зависящие от реализации средств разработки. Лучший способ с ними ознакомиться – заглянуть в соответствующий `.h` файл.

Включения заголовочного файла в исходный текст часто недостаточно для успешного подключения библиотеки, а для получения работоспособного продукта к исполнимому объектному коду создаваемого программного модуля требуется пристыковать исполнимый код библиотек. Для этого даются соответствующие указания сборщику. Детальное рассмотрение способов связывания модулей выходит за рамки языка Си и данного пособия. Отметим, что в интегрированных графических средах разработки программного обеспечения подключение библиотек производится через меню, а при использовании командной строки – посредством указания соответствующего ключа при вызове сборщика.

Для задания размера произвольного объекта в стандартной библиотеке используется искусственно введенный тип `size_t`, обычно совпадающий с наибольшим возможным беззнаковым целым в системе и зависящий от разрядности процессора. Его нельзя подменять типами `unsigned long` или даже `unsigned long long`, так как это может привести к некорректной работе программ.

9.2. Динамическое распределение памяти и операции с областями памяти

Это семейство функций позволяет распоряжаться памятью из кучи (англ. heap) и тем самым очень гибко управлять временем жизни переменных и констант: место выделяется в тот самый момент, когда этого хочет программист. Наиболее важные из этих функций даны в табл. 9.1.

Таблица 9.1. Семейство функций

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
`void *calloc( size_t n, size_t size);` `stdlib.h`	Выделяет для массива `n` ячеек памяти размером `size` байтов каждая. Память выделяется монолитным фрагментом	Указатель на начальный адрес памяти или **0**, если не удалось выделить память	См. `malloc`, `realloc`, `free`
`void *malloc( size_t size);` `stdlib.h`	Выделяет `size` байтов памяти. Память выделяется слитно	То же	См. `calloc`, `realloc`, `free`
`void *realloc( void *ptr, size_t size);` `stdlib.h`	Изменяет размер выделенной по указателю `ptr` памяти до `size` байтов	—»—	См. `calloc`, `malloc`, `free`
`void free( void *ptr);` `stdlib.h`	Освобождает память, выделенную по указателю `ptr`	Нет	См. `calloc`, `malloc`, `realloc`
`void *memset( void *dest, int val, size_t size);` `string.h`	Присваивает значение `val` первым `size` байтам (или `char`) памяти по указателю `dest`	Значение `dest`	См. `memcmp`, `memcpy`, `memmove`

Окончание табл. 9.1

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`void *memcpy(`
`void *dest,`
`const void *src,`
`size_t size);`
 	
`string.h`
	Копирует `size` байтов (или `char`) из области по указателю `src` в область по указателю `dest`. Если области памяти `src` и `dest` перекрываются, не гарантируется корректная работа	Значение `dest`	См.
			`memmove`,
			`memcmp`,
			`memset`

`void *memmove(`
`void *dest,`
`const void *src,`
`size_t size);`
 	
`string.h`
	Копирует `size` байтов (или `char`) из области по указателю `src` в область по указателю `dest`. Даже если области памяти `src` и `dest` перекрываются, гарантируется корректная работа в отличие от `memcpy`	Значение `dest`	См.
			`memcpy`,
			`memmove`,
			`memcmp`,
			`memset`

`int memcmp(`
`const void *buf1,`
`const void *buf2,`
`size_t size);`
 	
`string.h`
	Лексикографически сравнивает первые `size` байтов (или `char`) областей памяти по указателям `buf1` и `buf2` и возвращает результат сравнения	**<0**, если первые $n-1$ байт (или `char`) областей совпадают, $(n-1)$-й байт (или `char`) в `buf1` меньше $(n-1)$-го байта (или `char`) в `buf2`, $1 < n <$ `size`, нумерация ведется, как всегда, с **0**
		
		>0, если первые $n-1$ байт (или `char`) областей совпадают, а $(n-1)$-й байт (или `char`) в `buf1` больше $(n-1)$-го байта (или `char`) в `buf2`, $1 < n <$ `size`
		
		0, если первые `size` байт (или `char`) областей памяти идентичны
			См.
			`memcpy`,
			`memmove`,
			`memset`

Обычный цикл работы с областью динамической памяти включает ее выделение, использование и освобождение. Изменение размера выделенного фрагмента памяти используется нечасто, потому что оно в большинстве систем работает довольно медленно. Вообще постоянное выделение и утилизация динамических блоков данных являются одними из наиболее существенных факторов, замедляющих работу программы. Полный пример традиционного способа работы приведен ниже.

Пример:

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

int main()
{
struct my_struct {double real; double imag;} *a;
unsigned a_length=100, i;

if (a=(struct my_struct*)
  malloc(sizeof(struct my_struct)*a_length))
  {
  for (i=0;i<a_length;++i)
    { a[i].real=-10.0*i; (*(a+i)).imag=2.0/(i+1); }
  for (i=0;i<a_length;++i)
    printf("\n a[%d].real=%g; a[%d].imag=%g;",
      i, (a+i)->real, i, (a+i)->imag);
  free(a);
  }
else printf("\n Ошибка: невозможно выделить память.");
return 0;
}
```

9.3. Работа с текстовыми строками

Данная группа функций (табл. 9.2) призвана существенно упростить манипуляцию как константными, так и изменяемыми строками текста в программе. Все функции отвечают общей договоренности, согласно которой завершающим символом всякой текстовой строки является нулевой символ `'\0'`. Проверка выхода за отведенную область памяти

при копировании или дописывании в конец строки не осуществляется; контроль его отсутствия — забота программиста.

Таблица 9.2. Функции работы с текстовыми строками

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
`size_t sprintf( char *buffer, const char *format_string [, argument]n);` `stdio.h`	Действует аналогично `printf`, но осуществляет вывод в область памяти. Согласно строке формата `format_string`, выполняет форматные преобразования для данных `[, argument]n` и выводит получающуюся последовательность символов в символьный массив по указателю `buffer`	Число выведенных в `buffer` символов, не считая завершающего нулевого символа `'\0'`	См. `fprintf`, `printf`, `sscanf`
`size_t strlen( const char *string);` `string.h`	Возвращает длину строки в байтах (или `char` в зависимости от реализации), адрес которой определяется значением параметра `string`, не считая завершающего нулевого символа `'\0'`	Длина строки	
`void *memchr( const void *buf, char c, size_t cnt);` `string.h`	Ищет первое местонахождение символа `c` среди первых `cnt` символов области памяти по указателю `buf`	При нахождении `c` указатель на его расположение, иначе `NULL`	См. `strchr`, `strrchr`, `strpbrk`
`char *strpbrk( const char *string1, const char *string2);` `string.h`	В строке, адрес которой задан `string1`, находит первое местоположение любого символа из набора символов, содержащихся в строке, адрес которой задан `string2`. Завершающий символ `'\0'` не включается в поиск	Указатель на первое местоположение любого символа из `string2` в `string1` или `NULL`, если нет общих символов	См. `memchr`, `strchr`, `strnchr`

Продолжение табл. 9.2

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
`char *strchr( const char *string, int sim);` `string.h`	Находит первое вхождение символа `sim` в строке, адрес начала которой задан `string`. Если символ `sim` нулевой `'\0'`, поиск завершается в конце строки	Указатель на первое местонахождение символа, имеющего код `sim`, в строке по указателю `string`, `NULL`, если символ в строке отсутствует	См. `memchr`, `strpbrk`, `strrchr`
`char *strrchr( const char *string, int sim);` `string.h`	Находит последнее вхождение символа `sim` в строке, адрес начала которой задан `string`. Если символ `sim` нулевой `'\0'`, поиск завершается в конце строки	Указатель на последнее местонахождение символа, имеющего код `sim`, в строке по указателю `string`, `NULL`, если символ в строке отсутствует	См. `memchr`, `strpbrk`, `strchr`
`char *strstr( const char *string1, const char *string2);` `string.h`	Находит первое вхождение строки `string2` в строке `string1`	Указатель на местонахождение `string2` в строке `string1`, `NULL`, если `string2` в строке `string1` отсутствует	
`char *strcat( char *string1, const char *string2);` `string.h`	Дописывает строку по указателю `string2` в конец строки по указателю `string1`, помещая в конец строки-результата символ `'\0'`	Указатель `string1` на сцепленную строку	См. `strncat`
`char *strncat( char *string1, const char *string2, size_t n);` `string.h`	Дописывает не более `n` первых символов (менее `n` символов, если строка короче) из строки, по указателю `string2` в конец строки по указателю `string1`, устанавливая в конце результата `'\0'`. Если `n` больше длины строки `string2`, то длина строки `string2` используется вместо `n`	Указатель на сцепленную строку `string1`	См. `strcat`

Продолжение табл. 9.2

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`int strcmp(`
`const char *string1,`
`const char *string2);`
 	
`string.h`
	Лексикографически сравнивает строки по указателям `string1` и `string2`. Это означает, что для каждой пары символов, находящихся в одной и той же позиции от начала строк, сравниваются их коды до первого несовпадения	**<0,** если `string1` меньше `string2`;
		0, если `string1` идентична `string2`;
		>0, если `string1` больше `string2`
			См.
			`strncmp`,
			`stricmp`,
			`strnicmp`

`int stricmp(`
`const char *string1,`
`const char *string2);`
 	
`string.h`
	Лексикографически сравнивает строки по указателям `string1` и `string2`, считая заглавные и строчные буквы эквивалентными	**<0,** если `string1` меньше `string2` ;
		0, если `string1` идентична `string2`;
		>0, если `string1` больше `string2`
			См.
			`strcmp`,
			`strncmp`,
			`strnicmp`

`int strncmp(`
`const char *string1,`
`const char *string2,`
`size_t n);`
`string.h`
	Лексикографически сравнивает не более `n` первых символов в строках по указателям `string1` и `string2`	**<0,** если `string1` меньше `string2`;
		0, если `string1` идентична `string2`;
		>0, если `string1` больше `string2`
			См.
			`strcmp`,
			`stricmp`
			`strnicmp`

`int strnicmp(`
`const char *string1,`
`const char *string2,`
`size_t n);`
`string.h`
	Лексикографически сравнивает не более `n` первых символов в строках по указателям `string1` и `string2`, считая заглавные и строчные буквы эквивалентными	**<0,** если `string1` меньше `string2`;
		0, если `string1` идентична `string2`;
		>0, если `string1` больше `string2`
			См.
			`strcmp`,
			`stricmp`,
			`strncmp`

`char strcpy(`
`char *string1,`
`const char *string2);`
`string.h`
	Копирует строку по указателю `string2`, включая завершающий нулевой символ, по указателю `string1`	Указатель `string1` на строку-результат	См.
			`strncpy`

Продолжение табл. 9.2

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`char *strncpy(`
`char *string1,`
`const char *string2,`
`size_t n);`
 	
`string.h`
	Копирует ровно `n` символов (но не более, чем содержится в строке `string2`) строки по указателю `string2` в строку по указателю `string1`	Указатель `string1` на строку-результат. Если `n` меньше длины строки `string2`, символ `'\0'` не добавляется в новую строку. Если `n` больше длины строки `string2`, то в конец строки-результата добавляется символ `'\0'`	См. `strcpy`

`size_t strspn(`
`const char *string1,`
`const char *string2);`
 	
`string.h`
	Дает максимальную длину начальной подстроки строки `string1`, состоящей только из символов строки `string2`. Завершающий нулевой символ `'\0'` не учитывается	Индекс первого символа в строке по указателю `string1`, который не принадлежит набору символов, содержащихся в строке по указателю `string2`. Если `string1` начинается с символа не из `string2`, то возвращается **0**	См. `strcspn`

`size_t strcspn(`
`const char *string1,`
`const char *string2);`
 	
`string.h`
	Дает максимальную длину начальной подстроки строки `string1`, не содержащей ни одного символа из строки `string2`. Завершающий нулевой символ `'\0'` не учитывается	Индекс первого символа в строке по указателю `string1`, который принадлежит набору символов, содержащихся в строке по указателю `string2`. Если `string1` начинается с символа из `string2`, то возвращается **0**	См. `strspn`

Окончание табл. 9.2

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`char *strdup(const char *string);`
 	
`string.h`
	Получает область памяти (вызывая `malloc`) и копирует в эту область строку по указателю `string`	Указатель на область памяти, в которую скопирована строка, а в случае невозможности выделить память `NULL`	

Например: в результате исполнения следующего фрагмента программы

```
char s1[]="Мама мыла рамы", s2[]="Маша мыла Мурку.",
s3[1000];
int n=4;
sprintf(s3,"%s %d раза в год. ", s1, n);
strcat(s3,s2);
```

в буфере `s3` получается строка:

```
"Мама мыла рамы 4 раза в год. Маша мыла Мурку."
```

9.4. Общие принципы работы с устройствами ввода-вывода

По отношению к программе все внешние устройства обмена информацией делятся на следующие:

- ◊ *устройства ввода*, осуществляющие преобразование получаемой из операционной системы информации к внутреннему представлению в оперативной памяти программы. Стандартная библиотека гарантирует только поддержку клавиатуры;
- ◊ *устройства вывода*, производящие вывод информации вовне. В рамках стандартной библиотеки доступны терминал (простой текстовый дисплей или его аналог) и принтер, выбранный в системе по умолчанию;
- ◊ *устройства ввода-вывода* (двунаправленные), реализующие двухсторонний обмен информацией между программой и

операционной системой. Стандартная библиотека поддерживает обмен с файлами на произвольном носителе и прочими устройствами, представимыми в виде стандартных потоков.

Поток (англ. stream, pipe) – абстрактный программный механизм, позволяющий задавать способ обмена, назначать задействованные устройства, передавать и/или принимать данные и осуществлять контроль обмена информацией. Поток может находиться в каждый момент времени в одном из нескольких состояний:

- закрыт – обмен данными невозможен. Это его исходное и конечное состояния;
- открыт (инициализирован) на прием (чтение), передачу (запись) и ожидает следующего действия;
- обмен информацией;
- состояние ошибки, которая может иметь программно устранимый или неустранимый характер.

Понятие потока позволяет единообразно работать с устройствами, по-разному организованными.

Различают буферизованные и небуферизованные потоки.

Буферизованные потоки характеризуются тем, что для промежуточного хранения принимаемых или передаваемых данных используется область оперативной памяти компьютера, называемая *буфером*. Буферизация позволяет более комфортно для программы и операционной системы осуществлять обмен информацией с минимумом ожидания:

- при передаче данные укладываются в буфер и отправляющая их программа не ждет их фактического отправления. Она считает, что они уже отосланы, и может заниматься следующими вычислениями. Операционная система по мере возможности отправляет буферизованную информацию по назначению;
- при приеме в буфере накапливается некоторый объем данных, которые программа может забирать в «удобный» для нее момент времени поодиночке или блоком (*пакетом*), не ожидая каждый раз момента их прибытия от операционной системы.

Буферизация может осуществляться программно операционной системой или аппаратно контроллерами интерфейсов компьютера.

9.5. Файлы и потоки

Потоком называется обобщение понятия файла, позволяющее абстрактно и единообразно рассматривать процесс практически произвольных последовательных передачи и приема данных между программой и внешней средой. Понятия файла и потока выступают как родственные друг другу.

Стандартные способы работы с файлами предусматривают такую последовательность действий:

1) открытие файла. Для этого необходимо указать путь и имя открываемого файла, а также режим открытия, определяющий, можно ли осуществлять запись в файл или только чтение, как поступать при отсутствии или наличии файла с уже имеющимися данными и каким образом будет происходить дальнейшая работа. Результатом успешного открытия файла является получение указателя на открытый поток. При этом говорят, что данный поток связан с данным файлом;

2) [Перепозиционирование на нужный фрагмент файла]^{m} – необязательное действие, которое может перемежаться с шагом 3;

3) [Чтение и/или запись в файл]^{n}. Это действие чередуется с шагом 2;

4) закрытие файла.

9.6. Функции работы с файлами

Различают двоичный и текстовый режимы работы с файлами. В *двоичном режиме* информация в файле воспринимается как двоичная и точно соответствует ее представлению в памяти компьютера. Такие файлы человек не может читать без предварительной перекодировки в текстовый вид, но данные хранятся в них компактно. Они обрабатываются обычно функциями бесформатного доступа `fwrite` и `fread`, оперирующими данными в двоичном представлении.

Текстовые файлы содержат данные в человекочитаемом виде, и их можно редактировать даже при помощи простейших встроенных в операционную систему редакторов, но платой за это является больший объем файлов, особенно при записи в них чисел. Для чтения и записи таких файлов больше подходят

функции `fprintf` и `fscanf`, подобно `printf` и `scanf` производящие форматные преобразования.

Пример последовательно выполняемых записи содержимого массива структур в двоичный файл и последующего чтения содержимого двоичного файла в другой массив структур:

```
#include <stdio.h>

int main()
{
struct my_struct {double a; double b; int c;} my_s_dst[2],
my_s_src[2]={{1.23,2.34,10},{3.45,4.56,-10}};
char file_name[]="./data/file.dat";
FILE *file;
int i;

if ((file=fopen(file_name,"wb"))==NULL)
  return 1; /* файл не открывается на запись */

if(fwrite(&my_s_src[0],
  sizeof(struct my_struct),2,file)<2)
  return 2; /* невозможно записать 2 структуры */

if ((file=freopen(file_name,"rb",file))==NULL)
  return 3; /* файл не переоткрывается на чтение */

if(fread(&my_s_dst[0],
  sizeof(struct my_struct),2,file)<2)
  return 4; /* невозможно прочитать 2 структуры */

for (i=0;i<2;i++)
  {
  printf("\n my_s_src[%d].a=%g; \
my_s_src[%d].b=%g; my_s_src[%d].c=%d",
  i,my_s_src[i].a,i,my_s_src[i].b,i,my_s_src[i].c);
  printf("\n my_s_dst[%d].a=%g; \
my_s_dst[%d].b=%g; my_s_dst[%d].c=%d",
  i,my_s_dst[i].a,i,my_s_dst[i].b,i,my_s_dst[i].c);
  }
fclose(file);

return 0;
}
```

Действие этих и других популярных функций работы с файлами описано в табл. 9.3.

Таблица 9.3. Функции работы с файлами

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
`FILE *fopen( const char *pathname, const char *type);` `stdio.h`	Открывает файл с заданными параметром `pathname` именем и путем в режиме, заданном текстовой строкой `type` (см. табл. 9.4) и создает для него поток ввода/вывода	Указатель на открытый поток при успехе, иначе `NULL`	См. `fclose`, `fcloseall`, `ferror`, `freopen`
`FILE *freopen( const char *pathname, const char *type, FILE *stream);` `stdio.h`	Закрывает файл, связанный с потоком `stream`, и переназначает поток `stream` на файл, определенный в `pathname`, в режиме `type` (см. табл. 9.4). Используется для связи потоков `stdin`, `stdout`, `stderr`, `stdaux` и `stdprn` с файлами и изменения режима работы с файлом	Указатель на поток для нового открытого файла при успехе, иначе `NULL` В любом случае первоначально открытый и связанный с потоком `stream` файл закрывается	См. `fclose`, `fcloseall`, `ferror`, `fopen`
`int fclose( FILE *stream);` `stdio.h`	Закрывает поток файла `stream`, открытый при помощи `fopen` или `freopen`. Содержимое буферов записывается в место назначения (на диск или в память в зависимости от направления ввода/вывода), а память буферов освобождается системой, кроме памяти, выделенной с помощью `setbuf` или `setvbuf`	**0**, если поток успешно закрыт, `EOF` при ошибке	См.: `fcloseall`, `fflush`, `fopen`, `freopen`

Продолжение табл. 9.3

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`int fcloseall();`
 	
`stdio.h`
	Закрывает все потоки, открытые пользователем явно. Стандартные потоки `stdin`, `stdout`, `stdprn`, `stdaux` и `stderr` обычно не закрываются	Число закрытых потоков при успехе, `EOF` при ошибке	См. `fclose`, `fflush`, `fopen`, `freopen`

`int fwrite(`
`const char *buffer,`
`size_t size,`
`size_t count,`
`FILE *stream);`
`stdio.h`
	С текущей позиции записывает `count` элементов по `size` байтов каждый из массива `buffer` в выходной поток `stream` без форматного преобразования. Указатель файла, связанный с потоком `stream`, увеличивается на число действительно записанных байтов	Число фактически записанных элементов (может быть меньше `count` в случае ошибки)	См. `fread`, `fprintf`

`int fread(`
`void *buffer,`
`size_t size,`
`size_t count,`
`FILE *stream);`
`stdio.h`
	С текущей позиции из входного потока `stream` читает `count` элементов длиной `size` каждый и помещает их в заданный массив `buffer` без форматного преобразования. Указатель файла, связанный с потоком `stream`, увеличивается на число фактически прочитанных байтов	Число фактически прочитанных элементов (может быть меньше `count` в случае ошибки или преждевременного конца файла)	См. `fwrite`, `fprintf`

Продолжение табл. 9.3

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
`int fprintf( FILE *stream, const char *format_string [, argument]n); stdio.h`	Действует аналогично `printf`, но осуществляет вывод в поток. Для данных `[, argument]n` выполняет форматные преобразования согласно строке формата `format_string` и выводит получающуюся последовательность символов в поток `stream`	Число фактически записанных символов без учета завершающего нулевого символа (может быть меньше предусмотренного в случае ошибки)	См. `fscanf`, `printf`, `sprintf`, `fwrite`
`int fscanf ( FILE *stream, const char *format_string, [, argument]n); stdio.h`	Действует аналогично `scanf`, но осуществляет ввод из потока `stream`. Выполняет форматные преобразования последовательности символов из потока `stream` согласно строке формата `format_string` и размещает получающиеся данные по указателям в списке параметров `[, argument]n`. При этом каждый указатель `argument` должен ссылаться на переменную типа, соответствующего такому же по порядку элементу `format_string`	Число фактически прочитанных и присвоенных элементов ввода (может быть меньше предусмотренного в случае ошибки или преждевременного конца файла, а также наличия прочитанных, но не присвоенных элементов). Если конец файла достигнут ранее исчерпания списка, возвращается значение `EOF`	См. `fprintf`, `scanf`, `sscanf`, `fread`
`long ftell( FILE *stream); stdio.h`	Сообщает текущую позицию в файле, связанном с потоком `stream` в виде целого неотрицательного смещения относительно начала файла	Позиция в файле при успехе, иначе **1L**. Для стандартных потоков `stdin`, `stdout`, `stderr`, `stdaux` и `stdprn` значение не определено	См. `fseek`, `rewind`

Продолжение табл. 9.3

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`int fseek(`
`FILE *stream,`
`long offset,`
`int origin);`
 	
`stdio.h`
	Перемещает указатель на позицию в файле, связанном с потоком `stream`, на новое место, которое определяется параметрами `origin` и `offset` и с которого будет осуществляться следующая операция ввода/вывода с этим потоком. Указатель может быть перемещен на произвольное место внутри файла и за конец его, но не перед началом файла. Параметр `origin` может быть равен одной из констант:
	1) `SEEK_SET` = начало файла; *2*) `SEEK_CUR` = текущая позиция в файле; *3*) `SEEK_END` = конец файла
	Значение знакового целого параметра `offset` — отступ в байтах от места, задаваемого параметром `origin`. Для текстового режима работы с файлом `'t'` значение аргумента `offset` должно быть получено с помощью функции `ftell()` или равно нулю
		0 при успешном переносе указателя, иначе ненулевое значение. Для стандартных потоков `stdin`, `stdout`, `stderr`, `stdaux` и `stdprn` значение не определено	При перемещении указателя по файлу признак наличия символа, возвращенного в поток через вызов функции `ungetc()`, обнуляется, т.е. будет забыто, что был вызов функции `ungetc()`
			
			См. `ftell`, `rewind`

Продолжение табл. 9.3

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`void rewind(`
`FILE *stream);`
 	
`stdio.h`
	Устанавливает указатель позиции в файле, связанном с потоком `stream`, на начало файла, что эквивалентно последовательному вызову функций `fseek(stream, 0L, SEEK_SET); clearer(FILE *stream);` за исключением того, что возвращаемое `fseek` значение дополнительно сообщает об успешности перемещения указателя	Нет	См. `fseek`, `ftell`

`макрос`
 	
`int feof(`
`FILE *stream);`
`stdio.h`
	Определяет, не достигнут ли конец потока `stream` (т.е. связанного с ним файла)	**0**, если файл не кончился, иначе ненулевое значение, говорящее о проблеме при последующем чтении файла	См. `clearerr`, `ferror`, `perror`

`макрос`
 	
`int ferror(`
`FILE *stream);`
 	
`stdio.h`
	Проверяет, не возникла ли ошибка при чтении/записи данных потока `stream`. Если ошибка была, для данного потока флаг ошибки остается установленным до первого из двух событий: *1*) поток закрыт, *2*) вызвана функция `rewind` или `clearerr`. До этих пор значение флага ошибки ввода/вывода может быть считано при помощи `feof`	**0**, если ошибки не было, иначе ненулевое значение	См. `clearerr`, `feof`, `perror`

Продолжение табл. 9.3

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`void perror(`
`const char *string);`
 	
используует глобальные переменные:
 	
`int errno;`
`int sys_nerr;`
`char sys_errlist[ sys_nerr];`
`stdlib.h`
	Печатает сообщение об ошибке в стандартный поток для вывода сообщений об ошибках `stderr` в последовательности: *1*) задаваемая программистом строка `string`; *2*) двоеточие; *3*) системное сообщение об ошибке, соответствующее значению глобальной переменной `errno` (описанная в стандартной библиотеке Си); *4*) символ `'\n'` перевода строки*	Нет	См. `clearerr`, `ferror`, `strerror`

`void clearer(`
`FILE *stream);`
 	
`stdio.h`
	Очищает флаг ошибки и признак конца файла для указанного потока, иначе неизменный до закрытия потока или вызова функции `rewind`	Нет	См. `feof`, `ferror`, `perror`, `rewind`

Продолжение табл. 9.3

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`void setbuf(`
`FILE *stream,`
`char *buffer);`
 	
`stdio.h`
	Задает для ранее открытого потока `stream` новый буфер в зависимости от значения аргумента `buffer`. Если оно равно `NULL`, это означает отмену буферизации. Иначе значение аргумента `buffer` определяет адрес буфера – массива символов длиной `BUFSIZ`, где `BUFSIZ` – константа, определенная в файле `stdio.h`. Новый буфер используется для буферизованного ввода/вывода указанного потока `stream` взамен выделяемого по умолчанию системного буфера**	Нет	Функция `setbuf` вызывается сразу за одной из функций: `fopen`, `freopen`, `fseek` для данного потока, иначе содержимое буфера может быть повреждено или потеряно
			
			См. `fflush`, `fopen`, `freopen`, `fseek`, `fclose`

Окончание табл. 9.3

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание

`int fflush(`
`FILE *stream);`
 	
`stdio.h`
	Для файла, открытого на запись, записывает (сбрасывает) содержимое связанного с потоком `stream` буфера в файл, связанный с этим потоком. Поток остается открытым. Для небуферизованного потока при вызове `fflush` ничего не делается. Если файл открыт на чтение, буфер очищается. Автоматически запись буфера потока происходит при его заполнении, закрытии потока или нормальном завершении программы	`EOF` при ошибке, **0** при успешном завершении, даже если поток не буферизован или открыт только для чтения	Функция `fflush` отменяет действие вызванной перед ней функции `ungetc`
			
			См. `fclose`, `setbuf`

* Системное сообщение об ошибке доступно через глобальную переменную `sys_errlist` – массив сообщений, упорядоченных по соответствующим кодам ошибок. Функция `perror` печатает сообщение, используя значение переменной `errno` как индекс в массиве `sys_errlist`. Значение глобальной переменной `sys_nerr` равно максимальному числу элементов в массиве `sys_errlist`. Для правильной работы функция `perror` вызывается сразу после библиотечной функции, которая возвратила код ошибки, иначе значение переменной `errno` может быть повреждено последующими вызовами других функций.

** Потоки `stderr` и `stdout` по умолчанию не буферизованы, но им могут быть присвоены буфера посредством `setbuf`.

9.7. Режимы открытия файлов

Допустимые и не противоречащие друг другу значения режимов открытия и последующего доступа к файлу задаются в текстовой строке (не обязательно константной) по указателю `type`. Этот параметр присутствует в функциях `fopen` и `freopen`. Если права доступа не позволяют открыть файл в требуемом режиме, возвращается ошибка.

Таблица 9.4. Режимы открытия файлов

Значение	Режим	Примечание
`"r"`	Открыть для чтения	Файл должен существовать
`"r+"`	Открыть для чтения и записи	Файл должен существовать. При переходе с чтения на запись и обратно необходимо явно перемещать указатель на нужное место в файле с помощью функций `fseek`, `rewind`, `fsetpos`
`"w"`	Открыть пустой файл для записи	Если файл на момент открытия уже существовал, все его содержимое утрачивается. При переходе с чтения на запись и обратно необходимо явно перемещать указатель на нужное место в файле с помощью функций `fseek`, `rewind`, `fsetpos`
`"w+"`	Открыть пустой файл для чтения и записи	
`"a"`	Открыть для добавления в конец файла	При открытии файла, а также с помощью функций `fseek`, `rewind` и `fsetpos` указатель на место в файле всегда будет перемещаться на конец существовавших до открытия файла данных, которые в этом режиме не могут быть затерты. Если файла не было, он создается пустым, а в некоторых реализациях выдается ошибка. При переходе с чтения на запись и обратно необходимо явно перемещать указатель на нужное место в файле с помощью функций `fseek`, `rewind` и `fsetpos`
`"a+"`	Открыть для чтения и добавления в конец файла	

Окончание табл. 9.4

Значение	Режим	Примечание
't'	Добавляется к перечисленным выше спецификаторам для открытия файла в текстовом режиме. Символ перевода на новую строку '\n' может ассоциироваться с последовательностью символов <CR><LF> (возврат каретки, переход на новую строку, что характерно для Microsoft Windows) или с одиночным символом <LF> в зависимости от традиций операционной системы*	Если в спецификаторе режима открытия файла нет ни 'b', ни 't', устанавливается задаваемый глобальной переменной _fmode режим по умолчанию
'b'	Добавляется к перечисленным выше спецификаторам для открытия файла в двоичном режиме. В файл заносится то, что было в памяти, без форматного преобразования	
* Возможна нестандартная реакция на управляющие последовательности, что характерно для предыдущих реализаций стандартной библиотеки		

9.8. Стандартные потоки

Несколько потоков доступно пользователю всегда (табл. 9.5), их не нужно открывать и закрывать. Они поддерживаются операционной системой, и с ними можно работать так, как если бы они были потоками, связанными с файлами.

Таблица 9.5. Назначение стандартных потоков

Имя потока	Назначение	Примечание
stdin	Поток ввода, по умолчанию связан с клавиатурой, из командной строки перенаправляется в файл	

Окончание табл. 9.5

Имя потока	Назначение	Примечание
stdout	Поток вывода, по умолчанию связан с окном текстового терминала/консоли. В него попадает стандартный вывод программы: имя_программы 1> имя_файла	
stderr	Поток вывода сообщений об ошибках, по умолчанию связан с окном текстового терминала/консоли: имя_программы 2> имя_файла	
stdaux	Дополнительный поток	Присутствует не во всех операционных системах
stdprn	Поток вывода на принтер	То же

9.9. Функции форматного ввода-вывода

Эти функции обеспечивают гибкий единообразный программный интерфейс преобразования информации между внутренним компьютерным представлением и человекочитаемыми форматами. Среди них выделяют (табл. 9.6) функции для работы с текстовыми строками, файлами, клавиатурой (поток stdin) и терминалом (поток stdout).

Таблица 9.6. Функции форматного ввода-вывода

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
int printf(const char *format_string [, argument]^n); stdio.h	Осуществляет вывод в стандартный системный поток stdout. Форматные преобразования для данных [, argument]^n выполняются согласно строке формата format_string, и получающаяся последовательность символов выводится в поток stdout	Число фактически выведенных символов без учета завершающего нулевого символа (может быть меньше, чем предусмотрено в случае ошибки)	См. fscanf, fprintf, sprintf, fwrite

Продолжение табл. 9.6

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
`int scanf ( const char *format_string, [, argument]n);` `stdio.h`	Осуществляет ввод из стандартного системного потока `stdin`. Выполняет форматные преобразования последовательности символов из потока `stdin` согласно строке формата `format_string` и размещает получающиеся данные по указателям в списке параметров `[, argument]n`. При этом каждый указатель `argument` должен ссылаться на переменную типа, соответствующего такому же по порядку элементу в `format_string`	Число фактически прочитанных и присвоенных элементов ввода (может быть меньше, чем предусмотрено в случае ошибки или преждевременного конца файла, а также при наличии прочитанных, но не присвоенных элементов)	См. `fprintf`, `fscanf`, `sscanf`, `fread`
`int fprintf( FILE *stream, const char *format_string [, argument]n);` `stdio.h`	Действует аналогично `printf`, но осуществляет вывод в поток. Выполняет форматные преобразования для данных `[, argument]n` согласно строке формата `format_string` и выводит получающуюся последовательность символов в поток `stream`	Число фактически записанных символов без учета завершающего нулевого символа (может быть меньше, чем предусмотрено в случае ошибки)	См. `fscanf`, `printf`, `sprintf`, `fwrite`
`int fscanf ( FILE *stream, const char *format_string, [, argument]n);`	Действует аналогично `scanf`, но осуществляет ввод из потока `stream`. Выполняет форматные преобразования последовательности символов из потока `stream` согласно строке формата `format_string` и размещает получающиеся данные по указателям в спи-	Число фактически прочитанных и присвоенных элементов ввода (может быть меньше, чем предусмотрено в случае ошибки или преждевременного конца файла, а также наличия прочитанных, но не присвоенных	См. `fprintf`, `scanf`, `sscanf`, `fread`

Окончание табл. 9.6

Функция, параметры и заголовочный файл	Назначение и передаваемые параметры	Возвращаемое значение	Примечание
stdio.h	ске параметров [, argument]n. При этом каждый указатель argument должен ссылаться на переменную типа, соответствующего такому же по порядку элементу format_string	элементов). Если конец файла достигнут ранее исчерпания списка, возвращается значение EOF	
int sprintf(char *buffer, const char *format_string [, argument]n); stdio.h	Действует аналогично printf, но осуществляет форматированный вывод в строку по указателю buffer. Выполняет форматные преобразования для данных [, argument]n согласно строке формата format_string и выводит получающуюся последовательность символов в поток stream	Число фактически записанных символов без учета завершающего нулевого символа (может быть меньше, чем предусмотрено в случае ошибки)	См. fscanf, printf, fwrite
int sscanf (char *buffer, const char *format_string, [, argument]n); stdio.h	Действует аналогично scanf, но осуществляет форматированный ввод из строки по указателю buffer. Выполняет форматные преобразования последовательности символов из строки buffer согласно строке формата format_string и размещает получающиеся данные по указателям в списке параметров [, argument...]. При этом каждый указатель argument должен ссылаться на переменную типа, соответствующего такому же по порядку элементу format_string	Число фактически прочитанных и присвоенных элементов ввода (может быть меньше, чем предусмотрено в случае ошибки или преждевременного конца файла, а также наличия прочитанных, но не присвоенных элементов)	См. fprintf, scanf, fread

9.10. Символы управления форматным вводом-выводом

Способ ввода или вывода заданной числовой величины можно задать при помощи конструкций управления форматным вводом-выводом. Спецификатор формата не содержит разделителей и пробелов между составляющими, и его самая общая запись такова:

```
%[флаги][ширина_поля_ввода_вывода][.точность]
[длина]{буква_спецификатора_формата}
```

Отдельные составляющие выполняют следующие функции:

`флаги` — управляют выравниванием символьного ввода-вывода в пределах заданного поля и заполнением пустых знако-мест,

`ширина_поля_ввода_вывода` — минимальное число символов, которое будет выведено. Если для правильного представления числового значения требуется больше символов, этот параметр игнорируется и выводится столько символов, сколько минимально требуется для правильного отображения значения. Обычно `ширина_поля_ввода_вывода` задается целым неотрицательным числом. Вместо него в `printf` и подобных функциях можно указать `'*'`. В этом случае ширина поля вывода будет доведена до `n` символов, где `n` — значение в списке параметров функции, находящееся непосредственно перед выводимым при помощи данного формата значением. Так, `printf("%*d", 6, 213)` выведет `"   213"`, где ширина поля равна 6;

`точность` — неотрицательное целое число, определяющее минимальное количество цифр (в отличие от `ширины_поля_ввода_вывода`, где имеются в виду любые символы), которое будет выведено. При необходимости вначале выводятся дополнительные нули. Если точность не указана, печатается минимальное количество цифр числа, достаточное для его правильного вывода с точностью, принятой по умолчанию:

`точность` для форматов `"%f"`, `"%F"`, `"%e"` и `"%E"` задается числом цифр, следующих за символом десятичной точки (по умолчанию обычно 6). При явном задании равной нулю `точности` символ десятичной точки не выводится;

◊ `точность` у форматов `"%g"` и `"%G"` задает, сколько значащих цифр (цифр мантиссы) будет выведено. Если значение не мо-

жет быть выражено точно заданным числом цифр, оно округляется до ближайшего подходящего;

◊ в строковом формате `"%s"` обеспечивается вывод только заданного числом `точность` количества первых символов. При нулевом значении параметра `точность` данный параметр вообще не выводится. Подобно параметру `ширина_поля_ввода_вывода`, вместо числа в `printf` и аналогичных функциях можно указать `'*'`. В этом случае полная ширина поля вывода будет ограничена `n` символами, где `n` – значение в списке параметров функции, находящееся непосредственно перед выводимым с помощью данного формата значением. Так, `printf("%.*s", 3, "околесица")` выведет `"око"` в поле шириной 3 символа.

`буква_спецификатора_формата` грубо задает, является ли вводимый или выводимый параметр целым, действительным, строкой, одиночным символом или указателем.

`длина` однозначно уточняет тип параметра, указываемый первоначально `буквой_спецификатора_формата`.

Спецификаторы формата для целочисленных типов даны в табл. 9.7.

Таблица 9.7. Спецификаторы формата для целочисленных типов

Символы управления форматным вводом-выводом	Назначение	Примечания и примеры

`'d'`	Ввод-вывод целого числа в десятичном представлении со знаком	Пример:
		`int it=-51631;`
		`printf("%d",it);`
		печатает `"-51631"`

`'i'`	Вывод целого числа в десятичном представлении со знаком, а также ввод знакового десятичного, беззнакового восьмеричного или беззнакового шестнадцатеричного числа	Синоним формата `'d'` при выводе; при вводе поддерживает дополнительно ввод чисел в восьмеричной и шестнадцатеричной записи. Восьмеричное число предполагается начинающимся с нуля, а признаком шестнадцатеричного являются `"0x"` или `"0X"` в начале. Так, 0473 будет прочитано как 315_d по формату `'i'` или как 473_d посредством формата `'d'`

Окончание табл. 9.7

Символы управления форматным вводом-выводом	Назначение	Примечания и примеры

`'o'`	Ввод-вывод целого числа в восьмеричном представлении без знака	**Пример:**
		`unsigned us=51631;`
		`printf("%o",us);`
		печатает `"144657"`

`'u'`	Ввод-вывод целого числа в десятичном представлении без знака	**Пример:**
		`unsigned us=51631;`
		`printf("%u",us);`
		печатает `"51631"`

`'x'`
или
`'X'`
	Ввод-вывод целого числа в шестнадцатеричном представлении без знака. Цифры, соответствующие 10–15, обозначаются соответственно буквами `a, b, c, d, e, f` или `A, B, C, D, E, F`. Для вывода в формате `x` число содержит `x` на нижнем регистре, а для `X` — на верхнем регистре	**Примеры:**
		`unsigned us=51631;`
		`printf("%X",us);`
		печатает `"C9AF"`,
		`printf("%x",us);`
		печатает `"c9af"`

Спецификаторы формата для типов с плавающей точкой объединены в табл. 9.8.

Таблица 9.8. Спецификаторы формата для типов с плавающей точкой

Символы управления форматным вводом-выводом	Назначение	Примечания и примеры

`'f'`
или
`'F'`
	Вывод числа `double` с плавающей точкой в нормальном представлении (с фиксированной точкой), т.е. в виде `[-]ddd.ddd`, где число цифр после десятичной точки задается указанной точностью (по умолчанию обычно 6)	Поскольку эти форматы не предусматривают записи в экспоненциальном виде, `f` и `F` различаются только в способе вывода сообщения о «неправильном значении» (англ. not a number, NaN) или бесконечности (англ. infinity). В таком случае для формата `'f'` будет выведено `nan`, `inf` или `infinity`, для формата `'F'` соответственно `NAN`, `INF` или `INFINITY`

Окончание табл. 9.8

Символы управления форматным вводом-выводом	Назначение	Примечания и примеры

		Примеры:
		`printf("%F",2.5e6);` печатает `"2500000.000000";`
		`printf("%F",2.5e600);` вероятно (зависит от системы) выведет `"INF"`

`'e'`
или
`'E'`
	Вывод числа `double` с плавающей точкой в экспоненциальной форме записи, т.е. в формате
	`[-]d.ddd {e или E}`
	`[+ или -]dd[d]`,
	где `d` – десятичное число, а число цифр после десятичной точки задается указанной точностью (по умолчанию обычно 6). Экспонента состоит не менее чем из двух цифр. Для формата `'e'` число содержит символ экспоненты `'e'` на нижнем регистре, а для `'E'` – на верхнем регистре: `E`
		Примеры:
		`double dk=-93.45e5;`
		`printf("%e",dk);`
		печатает `"-9.345000e+6"`, а `printf("%E",dk);`
		печатает `"-9.345000E+6"`

`'g'`
или
`'G'`
	Ввод-вывод числа с плавающей точкой с автоматическим выбором между нормальным (с фиксированной точкой) и экспоненциальным представлением чисел
	Если экспонента < –4 или больше либо равна точности, используется формат `'e'` или `'E'`, иначе `'f'` или `'F'`.
	Для формата `'g'` число содержит символ экспоненты `'e'` на нижнем регистре, а для `'G'` – на верхнем регистре: `'E'`
		Конечные нули из дробной части результата удаляются, а символ десятичной точки появляется, только если справа от него есть цифра
		Примеры:
		`double d1=-93.45e3, d2=-93.45e5;`
		`printf("%G",d1);`
		печатает `"-93450"`,
		`printf("%G",d2);`
		печатает `"-9.345E+06"`

Дополнительные спецификаторы формата и управляющие символы объединены в табл. 9.9.

Таблица 9.9. Дополнительные спецификаторы формата

Символы управления форматным вводом-выводом	Назначение	Примечания и примеры

`'c'`	Формат ввода-вывода одиночного символа	Пример:
		`char str[]="строчка";`
		`printf("%c",str[5]);`
		печатает `'к'`

`'s'`	Формат ввода-вывода строки текста, завершенной символом `'\0'` (последний символ не выводится)	Пример:
		`char chr[]="строчка";`
		`printf("%s",chr);`
		печатает `"строчка"`

`'p'`	Формат вывода значения указателя типа `void*`	Пример:
		`double a[1000];`
		`printf("%p",(void*)a);`

`'n'`	Формат, помещающий число уже напечатанных символов по указателю типа `int*`, находящемуся на соответствующем месте в списке аргументов; никакого вывода по этому формату не осуществляется. Этот формат дает возможность программисту выяснить истинный объем форматного вывода, во многих ситуациях определяемый автоматически	Пример:
		`int c;`
		`printf ("%d %n\n", 12345, &c);`
		присваивает `c=5`, так как выведено 5 символов

`"%%"`	Употребление в строке формата двойного знака процента выводит одиночный символ процента на экран, для этого не требуется аргумента в списке параметров	Пример: `printf ("%f %%\n", 100.2);` печатает `"100.2%"`

Возможные флаги формата, каждый из которых применим обычно к нескольким форматам, сведены в табл. 9.10.

Таблица 9.10. Флаги формата

Флаг	Назначение	Примечания и примеры
`'-'`	Выравнивание выводимого числа осуществляется по левой границе поля вывода (по умолчанию по правой границе)	
`'+'`	В сочетании со знаковыми форматами `"%d"`, `"%i"`, `"%e"`, `"%E"`, `"%f"`, `"%F"`, `"%G"` и `"%g"` заставляет всегда печатать знак числа даже для положительных значений, в противном случае печатается только знак «минус»	
`' '` (пробел)	Для знаковых форматов `"%d"`, `"%i"`, `"%e"`, `"%E"`, `"%f"`, `"%F"`, `"%G"` и `"%g"` ставит один пробел перед числом при отсутствии знака (т.е. для положительных значений) для выравнивания длины вывода положительных и отрицательных чисел	Поддерживается не всеми реализациями языка Си
`'#'`	Для форматов с плавающей точкой `"%f"`, `"%F"`, `"%e"`, `"%E"`, `"%g"` и `"%G"` точка выводится всегда. Для форматов `"%g"` и `"%G"` незначащие нули после точки не удаляются. Для восьмеричного и шестнадцатеричных форматов `"%o"`, `"%x"` и `"%X"` префиксы `"0"`, `"0x"` и `"0X"` соответственно предпосылаются всякому ненулевому числу (иначе эти префиксы не выводятся, что может повлечь неверное чтение некоторых значений)	Пример: `printf( "%#g", 100);` печатает `"100.000"`
`'0'`	Предшествующие числу пробелы (если они полагаются) в поле вывода заменяются тем же числом незначащих нулей, устанавливаемых после знака, если он есть. Данный флаг игнорируется при наличии флага `'-'` или при явном указании точности	Пример: `printf( "%05d", 14);` печатает `"00014"`
Число	Добавляет пробелы слева до требуемого размера поля. Если стоит после `'0'`, отсутствующий знак у положительных чисел заменяется пробелом	

Возможное содержимое поля `длина` приведено в табл. 9.11.

Таблица 9.11. Возможное содержимое поля длина

Модификатор длина	Назначение	Примечания и примеры
"hh" или "HH"	Для целочисленных форматов "%u", "%d" и "%i" значение размера unsigned или int получено преобразованием соответственно из unsigned char или char	Любой аргумент типа char, unsigned char, short и unsigned short по умолчанию приводится к типу int (для форматов "%i" и "%d") или unsigned (для форматов "%o", "%u", "%x", и "%X") Пример: unsigned char q='J'; printf("%hhu",q);
'h' или 'H'	Для целочисленных форматов "%u", "%d" и "%i" значение размера unsigned или int получено преобразованием соответственно из unsigned short или short	

'l'	Для целочисленных форматов "%u", "%d" и "%i" указывает, что аргумент имеет тип соответственно unsigned long или long	Пример:
		unsigned long y=12L;
		printf("%lu",y);

"ll"	Для целочисленных форматов "%u", "%d" и "%i" указывает, что аргумент имеет тип unsigned long long или long long	Пример: unsigned long long z=10000; printf("%llu",z);

'L'	Для форматов с плавающей точкой: "%f", "%F", "%e", "%E", "%g" и "%G" указывает, что аргумент имеет тип long double	По умолчанию любой аргумент типа float автоматически преобразуется в double
		Пример:
		long double dd=-1.2e4;
		printf("%Lg",dd);

'z'	Для целочисленных форматов сообщает о том, что данный аргумент имеет тип системного беззнакового целого size_t	size_t — беззнаковый целочисленный тип, гарантирующий корректное хранение выраженного в байтах размера любого объекта, например массива или структуры

Окончание табл. 9.11

Модификатор длина	Назначение	Примечания и примеры
'j'	Для целочисленных форматов сообщает о том, что данный аргумент имеет тип intmax_t	intmax_t — самый длинный знаковый целочисленный тип в данной реализации языка Си
't'	Для целочисленных форматов сообщает о том, что данный аргумент имеет тип ptrdiff_t	ptrdiff_t — знаковый целочисленный тип, гарантирующий корректное представление разности двух любых указателей в пределах одного массива или структуры

Примеры:

1. Строка формата

```
"|%6d|%-6d|%+7d|%+-7d|%6d|%06d|%6.0d|%6.2d|%d|\n"
```

выводит каждое из чисел 0, 1, –2, 100000 целого типа следующим образом:

```
|     0|0     |     +0|+0     |     0|000000|      |    00|0|
|     1|1     |     +1|+1     |     1|000001|     1|    01|1|
|    -2|-2    |     -2|-2     |    -2|-00002|    -2|   -02|-2|
|100000|100000|+100000|+100000|100000|100000|100000|100000|100000|
```

2. `printf("|%6u|%6o|%6x|%8#X|%10.8x|%*X|\n",x,x,x,x,x,10,x);`

В списке параметров форматной строки перед последним аргументом стоит число 10, задающее ширину поля для последнего формата. Она выводит беззнаковые числа 0, 1, 100000, последовательно загружаемые в переменную x следующим образом:

```
|     0|     0|     0|     0X0|  00000000|         0|
|     1|     1|     1|     0X1|  00000001|         1|
|100000|303240| 186a0| 0X186A0|  000186a0|     186A0|
```

3. Строка формата `"|%12.4f|%12.4e|%12.4G|\n"`
выводит каждое из чисел 0, 1, –2, 300, 4000, 50000, 12345, 123456 типа double следующим образом:

```
|      0.0000|  0.0000e+00|           0|
|      1.0000|  1.0000e+00|           1|
|     -2.0000| -2.0000e+00|          -2|
|    300.0000|  3.0000e+02|         300|
|   4000.0000|  4.0000e+03|        4000|
|  50000.0000|  5.0000e+04|       5E+04|
|  12345.0000|  1.2345e+04|   1.234E+04|
| 123456.0000|  1.2346e+05|   1.235E+05|
```

Для этих значений аналогичный вывод получился бы с форматной строкой `"|%12.4Lf|%12.4Le|%12.4LG|\n"` для типа `long double`.

Пример: в результате выполнения приведенного ниже фрагмента кода

```
struct {double a[100]; int b; } stru;
size_t st;
intmax_t imt=99;
ptrdiff_t pdt;
st=sizeof(stru);
pdt=((void*)&(stru.b))-((void*)&(stru.a[10]));
printf("\nst=%zu; imt=%jd; pdt=%tx",st,imt,pdt);
```

может быть выведен следующий текст:

```
st=808; imt=99; pdt=2d0
```

(конкретный результат в отношении `st` и `pdt` зависит от компилятора и системы вследствие различий в разрядности и выравнивании).

9.11. Базовые математические функции

В современном Си имеются три способа разной точности для представления действительных чисел, это отражается введением трех эквивалентных по смыслу функций. Тип возвращаемого значения, за исключением некоторых функций преобразования к целому, совпадает с типом принимаемого значения. Наиболее употребительные математические функции стандартной библиотеки даны в табл. 9.12. Ими далеко не исчерпывается мощь математической части стандартной библиотеки Си, делающей доступными почти все возможности блоков вычислений с плавающей точкой новейших процессоров. Для более полного знакомства рекомендуется обратиться к соответствующему заголовочному файлу `"math.h"` системы программирования – в нем приведены не только прототипы функций, но и полезные для программной реализации математических алгоритмов константы.

Таблица 9.12. Базовые математические функции

№ п/п	Функция, параметры и заголовочный файл	Назначение, передаваемые параметры и возвращаемое значение

1	`float fabsf(float x);`
	`double fabs (double x);`
	`long double fabsl(long double x);`
	`math.h`
	`int abs(int x);`
	`long labs(long x);`
	`long long llabs(long long x);`
	`stdlib.h`
		Вычисляет абсолютное значение (модуль) `x`: $\|x\|$

2	`float logf(float x);`
	`double log (double x);`
	`long double logl(long double x);`
	`float log2f(float x);`
	`double log2 (double x);`
	`long double log2l(long double x);`
	`float log10f(float x);`
	`double log10 (double x);`
	`long double log10l(long double x);`
	`math.h`
		Вычисляет натуральный логарифм числа `x`: $\ln(x)$
		Вычисляет двоичный логарифм числа `x`: $\log_2(x)$
		Вычисляет десятичный логарифм числа `x`: $\log_{10}(x)$

3	`float expf(float x);`
	`double exp (double x);`
	`long double expl(long double x);`
	`float exp2f(float x);`
	`double exp2 (double x);`
	`long double exp2l(long double x);`
	`math.h`
		Вычисляет экспоненту числа `x` с натуральным показателем `e`: e^x
		Вычисляет экспоненту числа `x` с показателем `2`: 2^x

4	`float sqrtf(float x);`
	`double sqrt (double x);`
	`long double sqrtl(long double x);`
		Вычисляет квадратный корень числа `x`: $\sqrt{x}$
		Очевидно, `x` должно быть неотрицательным. Если это условие нарушено, возвращается NaN (англ. not a number)

Продолжение табл. 9.12

№ п/п	Функция, параметры и заголовочный файл	Назначение, передаваемые параметры и возвращаемое значение

	`float cbrtf(float x);`
	`double cbrt (double x);`
	`long double cbrtl(long double x);`
	`math.h`
		Вычисляет кубический корень числа `x`: $\sqrt[3]{x}$

5	`float powf(float base, float exp);`
	`double pow (double base, double exp);`
	`long double powl(long double base, long double exp);`
	`math.h`
		Возводит число `base` в степень `exp`: $base^{exp}$.
		Если `base<0`, `exp` должна быть целой. Если `base==0`, `exp` должна быть неотрицательной. Если одно из этих условий нарушено, возвращается NaN

6	`float sinf(float x);`
	`double sin (double x);`
	`long double sinl(long double x);`
		Вычисляет синус `x`: `sin(x)`

	`float asinf(float x);`
	`double asin (double x);`
	`long double asinl(long double x);`
	`math.h`
		Вычисляет арксинус `x`: `arcsin(x)`. Аргумент `x` должен по модулю не превосходить 1, иначе возвращается NaN

7	`float cosf(float x);`
	`double cos (double x);`
	`long double cosl(long double x);`
		Вычисляет косинус `x`: `cos(x)`

	`float acosf(float x);`
	`double acos (double x);`
	`long double acosl(long double x);`
	`math.h`
		Вычисляет арккосинус `x`: `arccos(x)`. Аргумент `x` должен по модулю не превосходить 1, иначе возвращается NaN

8	`float tanf(float x);`
	`double tan (double x);`
	`long double tanl(long double x);`
		Вычисляет тангенс `x`: `tg(x)`

	`float atanf(float x);`
	`double atan (double x);`
	`long double atanl(long double x);`
		Вычисляет арктангенс `x`: `arctg(x)`. Возвращаемое значение находится в диапазоне $[-\pi/2, \pi/2]$

Продолжение табл. 9.12

№ п/п	Функция, параметры и заголовочный файл	Назначение, передаваемые параметры и возвращаемое значение

	`float atan2f(float y, float x);`
	`double atan2 (double y, double x);`
	`long double atan2l(long double y, long double x);`
	`math.h`
		Вычисляет арктангенс `x` с выяснением квадранта: `arctg(x)`. Возвращаемое значение находится в диапазоне $[-\pi,\pi]$

9	`float ceilf(float x);`
	`double ceil (double x);`
	`long double ceill(long double x);`
		Находит ближайшее целое, не меньшее `x`: $\lceil x \rceil$

	`float floorf(float x);`
	`double floor (double x);`
	`long double floorl(long double x);`
		Находит ближайшее целое, не большее `x`: $\lfloor x \rfloor$

	`float truncf(float x);`
	`double trunc (double x);`
	`long double truncl(long double x);`
		Находит ближайшее целое, не превосходящее по модулю `x`

	`float roundf(float x);`
	`double round (double x);`
	`long double roundl(long double x);`
	`long lroundf(float x);`
	`long lround (double x);`
	`long lroundl(long double x);`
	`long long llroundf(float x);`
	`long long llround (double x);`
	`long long llroundl(long double x);`
	`math.h`
		Находит ближайшее целое. При этом дробная часть $\pm 0{,}5$ округляется в большую по модулю сторону (т.е. дальше от нуля)

10	`float fmodf(float x, float y);`
	`double fmod (double x, double y );`
	`long double fmodl(long double x, long double y);`
		Возвращает остаток от деления `x/y`. Знак остатка тот же, что у делимого `x`

Продолжение табл. 9.12

№ п/п	Функция, параметры и заголовочный файл	Назначение, передаваемые параметры и возвращаемое значение

	`float modff(float x, float* iptr);`
	`double modf (double x, double* iptr);`
	`long double modfl(long double x, long double* iptr);`
	`math.h`
		Расщепляет число на целую и дробную части того же типа. Обе части имеют знак исходного числа. Функция возвращает дробную часть, а целая часть записывается по указателю `iptr`

	`div_t div(int x, int y);`
	`ldiv_t ldiv(long x, long y);`
	`lldiv_t lldiv(long long x, long long y);`
	Здесь:
	`struct div_t {int quot; /*частное*/ int rem; /* остаток */};`
	`struct ldiv_t {long quot;/*частное*/ long rem; /* остаток */};`
	`struct lldiv_t {`
	`long long quot; /* частное */`
	`long long rem; /* остаток */};`
	`stdlib.h`
		Одновременно возвращает целое частное и остаток от деления двух целых. Результат представлен в виде структуры специального вида

11	`float copysignf(float x, float y);`
	`double copysign (double x, double y);`
	`long double copysignl(long double x, long double y);`
	`макрос signbit(arg)`
	`math.h`
		К модулю числа `x` приписывается знак числа `y`: `|x|·sign(y)`
		Возвращает ненулевое значение, если число с плавающей точкой отрицательно; иначе 0

Продолжение табл. 9.12

№ п/п	Функция, параметры и заголовочный файл	Назначение, передаваемые параметры и возвращаемое значение

12	`float fmaxf(float x, float y);`
	`double fmax (double x, double y);`
	`long double fmaxl(long double x, long double y);`
	`float fminf(float x, float y);`
	`double fmin (double x, double y);`
	`long double fminl(long double x, long double y);`
	`math.h`
		Возвращает наибольшее из двух чисел: `max(x,y)`
		Возвращает наименьшее из двух чисел: `min(x,y)`

13	макрос
	`isfinite(arg)`
	макрос
	`isinf(arg)`
	макрос
	`isnan(arg)`
	макрос
	`isnormal(arg)`
	макрос
	`fpclassify(x)`
		Возвращает 0, если передано бесконечное значение любого знака: ±∞; иначе возвращает ненулевое значение
		Возвращает ненулевое значение, если передано бесконечное значение любого знака: ±∞; иначе 0
		Возвращает ненулевое значение, если передано не число (недостоверное значение, не представимое с плавающей точкой); иначе 0
		Возвращает ненулевое значение, если число нормализовано (относительная погрешность представления находится в норме); иначе 0
		Определяет характер числа `x`. Возвращает одно из следующих значений: *1*) `FP_INFINITE` — бесконечное значение любого знака: ±∞; *2*) `FP_NAN` — не число (значение совершенно недостоверно); *3*) `FP_NORMAL` — нормализованное число (отно-

Окончание табл. 9.12

№ п/п	Функция, параметры и заголовочный файл	Назначение, передаваемые параметры и возвращаемое значение
	`math.h`	сительная погрешность представления находится в норме); *4*) `FP_SUBNORMAL` – ненормализованное число (относительная погрешность представления повышена); *5*) `FP_ZERO` – машинный нуль

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Исследуйте на примерах различные функции округления до целого из стандартной библиотеки.
2. Напишите собственную реализацию функции `strcpy` стандартной библиотеки.
3. Средствами любых функций стандартной библиотеки реализуйте аналог функции `printf`, в котором роль определяющего формат символа `'%'` играл бы символ `'@'`.
4. Все ли верно в данном фрагменте программы:
```
float *pa, *pb, pc[N];
pb=(float*)malloc(sizeof(float*)*N);
for (i=0; i<=N; i++) pb[i]=*(pc+i)=0.01*i;
pa=pc; pc=pb; pb=pc; free(pa);
```
Если да, аргументируйте это; если нет, исправьте.
5. Структура вида
```
struct stst {unsigned L; double *a; };
```
реализует вектор длиной `L`, состоящий из действительных чисел. Имеется вектор произвольной длины `struct stst c[N]`, каждый элемент которого – такая структура. Реализуйте пару функций:

а) создания точной независимой копии подобных `c` векторов произвольной длины `N` с использованием динамического выделения памяти. Вложенные векторы типа `struct stst` также имеют произвольные неодинаковые длины `L`. Под независимостью понимается следующее: все, что относится к оригиналу, и то, что относится к копии, занимают непересекающиеся области памяти;

б) освобождения памяти динамически выделенных копий, полученных в пункте *а* задачи.

6. Можно ли выяснить, адресует ли данный указатель объект на стеке или в куче? Если нет – почему? Если да – как?
7. Реализуйте два варианта записи и чтения вектора произвольной длины `N` чисел `double` в (из) файл(а) – двоичный и текстовый. Какие вы-

воды можно сделать относительно компактности и читаемости файла человеком?

8. Запрограммируйте функцию записи в файл вектора длиной `N` одинаковых структур типа: `typedef struct {unsigned NameLength; char Name[128]; } my_st_type;` а также эффективную функцию чтения одной `n`-й по счету структуры из файла, `0≤n<N`, с применением позиционирования в файле средствами стандартной библиотеки.

9. Создайте свой менеджер памяти (аналоги функций `malloc`, `free`) на базе функций стандартной библиотеки. Идея его работы такова: сначала выделяется большой непрерывный блок памяти заданного размера. В дальнейшем из него по требованию внешней программы ваш менеджер каждый раз выдает первый попавшийся блок подходящего размера. Также по требованию внешней программы области памяти возвращаются вашему менеджеру (помечаются в менеджере как свободные). При возможности две смежные области свободной памяти и более объединяются. Напишите внешний программный модуль, интенсивно запрашивающий и освобождающий в случайном порядке блоки памяти случайного допустимого для вашего менеджера размера. Поэкспериментируйте. Какие свойства простого менеджера памяти вы заметили?

10. Усовершенствуйте менеджер памяти предыдущей задачи так, чтобы по запросу выдавались фрагменты памяти наименьшего подходящего размера. Поэкспериментируйте.

ГЛАВА 10 ПРЕПРОЦЕССОР

10.1. Общие сведения о директивах препроцессора

Как следует из названия (англ. preprocessor, предварительный обработчик), препроцессор подвергает обработке исходные тексты программы до того, как они поступят на вход компилятора. При помощи препроцессора можно:

- ◊ убирать бесполезные с точки зрения компилятора комментарии;
- ◊ вводить альтернативным модификатору `const` способом именованные (имеющие собственное уникальное имя) константы;
- ◊ создавать именованные макроопределения (аналогичные термины: макросы, макро, макроподстановки, макровызовы, макрозамещения, англ. macro), смысл которых состоит в замене препроцессором имени такого макроопределения (всюду в исходном тексте файла, где оно определено) на его содержимое – некоторый исходный текст на Си. Самая мощная возможность – введение макроопределений с параметрами, действующих подобно функциям;
- ◊ управлять компиляцией в том смысле, что можно автоматически включать, исключать или менять фрагменты кода программы непосредственно перед компиляцией в зависимости от факта определения именованных констант и макроопределений и значений именованных констант;
- ◊ до некоторой степени управлять диагностическими сообщениями компилятора и осуществлять привязку этих сообщений к заданным фрагментам программы.

Ограниченное константное выражение (ОКВ) – это допустимое в Си выражение целого типа, которое может содержать:

- целые константы, преобразуемые к типу `long` или `unsigned long` в зависимости от их значения;
- символьные константы, интерпретируемые компилятором в соответствии с набором символов по умолчанию, зависящему от операционной системы, на которой установлены средства программирования, ее настроек, а также самого компилятора. В частности, они могут быть типа `signed char` или `unsigned char` и длиной 1 или 2 байта. Привязка программистом символьных констант к конкретным числовым значениям является плохим стилем программирования и может вести к некорректной работе программ в системах с другими настройками;
- арифметические, логические, побитовые операции, операции сдвига и сравнения;
- идентификаторы, допускаемые в некоторых реализациях, не являющиеся макросами и рассматриваемые как имеющие нулевое значение;
- логическая операция препроцессора `defined`;
- макроподстановки, вызываемые перед вычислением ОКВ.

В ОКВ не допускается использовать что-либо, учитывающее информацию об особенностях системы программирования и архитектуры компьютера, т.е. то, что может «знать» только компилятор. К недопустимым элементам ОКВ в зависимости от реализации могут относиться:

- операции явного приведения типа;
- выражение `sizeof`;
- вещественные константы;
- значения констант перечислимого типа `enum`.

Все значения типа `enum` так же, как и все идентификаторы, не являющиеся макроподстановками, обычно рассматриваются как нулевые константы. Можно сказать, что возможности записи ОКВ существенно ограничены. Кодировки значений «истина» и «ложь» для ОКВ те же, что и в обычных выражениях.

Все директивы препроцессора, кроме выражения `defined`, предваряются символом `'#'`, который располагается всегда только в самой левой позиции строки. Между `'#'` и последующей директивой препроцессора может находиться произвольное число пробелов или табуляций. Таким об-

разом, начинающиеся с '#' строки (и только они) обрабатываются исключительно препроцессором и в своем оригинальном виде отсутствуют в исходных текстах, попадающих на вход компилятора. Директивой препроцессора считается все последующее содержимое строки вплоть до символа окончания строки. Следующая строка, если в ее начальной позиции отсутствует символ '#', распознается как обычная строка программы.

На конце всех директив препроцессора отсутствует точка с запятой. На это имеются следующие основания. Во-первых, директива препроцессора не является оператором, а только операторы могут завершаться символом ';'. Во-вторых, даже если препроцессор что-то вставляет в исходный текст программы, хороший стиль – оформление этого фрагмента без точки с запятой на конце, которую при необходимости ставят в основном исходном тексте. Только в таком случае гарантируется корректность кода, а программа имеет регулярный вид.

10.2. Включение файла в файл директивой #include

Большие проекты чрезвычайно неудобно размещать в одном файле: код становится необозримым, а коллективная работа программистов над программой – крайне затруднительной. В ходе эволюции языков программирования возникло понимание, что в одном файле оптимально хранить одну среднюю или большую функцию (100 операторов и более) или несколько мелких, тесно связанных своим назначением. Для того чтобы в других файлах проекта использовать функции, переменные и константы из данного файла, в этих файлах должны быть их декларации. Но при этом одинаковые декларации дублируются во множестве файлов, что неудобно, особенно когда необходимо одинаково поправить декларацию в десятках файлов. На помощь приходит одна из возможностей препроцессора – включать полностью текст одного файла в

нужное место другого файла посредством всего одной директивы:

```
#include "путь_и_имя_файла_для_вставки_в_данный_файл"
```

В отличие от операторов в конце директив препроцессора точка с запятой не нужна. Ровно в место, где встретилась такая директива, препроцессор «мысленно» вставит полностью текст из нужного файла.

Например:

```
#include "../inc/my_definitions.h"
```

Путь к файлу записывается так, как принято в данной операционной системе. Если указывать так называемый абсолютный путь, начиная с корня диска, текст программы будет плохо переносимым на новое место: придется править множество файлов. Поэтому путь к нужному *включаемому заголовочному* файлу рекомендуется приводить относительно данного файла. Из-за перемещения всего дерева папок с проектом на новое место проект не перестанет быть работоспособным. Все современные средства разработки программного обеспечения могут быть настроены так, что программист им единожды указывает список папок, в которых находятся все включаемые файлы проекта. Поэтому поиск их месторасположения сужается до такого списка папок, а также текущей папки. О расположении файлов, дающих описание ресурсов стандартных библиотек, за редчайшим исключением, система программирования «знает» сама. Программисту остается лишь включить необходимые файлы при помощи директивы, отличающейся от рассмотренной выше видом кавычек:

```
#include <имя_файла_для_вставки_в_данный_файл>
```

Пара знаков сравнения < > вокруг имени включаемого файла сообщает препроцессору, что используются готовые библиотечные заголовочные файлы, и дает подсказку, где их искать.

Хотя включаемые файлы в принципе могут содержать все допускаемое языком Си, они выполняют особую миссию в программе и опытные программисты обычно не сомневаются, что именно следует вынести во включаемые заголовочные

файлы, а что оставить в основном файле. Хороший стиль программирования предусматривает разделение содержимого, представленное в табл. 10.1.

Таблица 10.1. Разделение содержимого файлов проекта

Включаемые файлы	Основной код программы
Декларации глобальных переменных и констант, прототипы функций, обычно используемые более чем в одном файле программы	*Определения* переменных, констант, функций вне зависимости от того, используются они локально в данном файле или к ним обращаются из других мест программы
Встраиваемые (inline) функции, используемые более чем в одном файле программы	Локально действующие встраиваемые (inline) и статические (static) функции
Определения разделяемых многими модулями констант и выражений, созданных при помощи #define	Все декларации, прототипы, константы и макроопределения, востребованные только в одном файле
Введение макросов	Практически весь исполнимый код программы
Введение пользовательских типов данных, в частности с привлечением typedef, struct, enum	–

Как видно из табл. 10.1, включаемые файлы используются в основном для деклараций и макроопределений, что выступает одной из основных причин осуществлять их включение в начало файлов. Именно поэтому включаемые файлы часто называют *заголовочными* (англ. header) и дают им расширение .h, чтобы подчеркнуть назначение. В начале файлов может быть много директив #include, поэтому следует тщательно продумать последовательность включения, поскольку включаемые файлы могут зависеть друг от друга и сами включать друг друга. Хороший стиль написания заголовочных файлов предполагает проверку того, не включен ли этот файл, и автоматическое включение всякого заголовочного файла в файл программы строго единожды. Для этого применяются директивы препроцессора #define, #ifdef, #ifndef и #endif, а также defined.

10.3. Простые макроопределения. Директивы #define и #undef

Общая форма записи простого макроопределения без параметров такова:

```
#define ИМЯ_МАКРООПРЕДЕЛЕНИЯ\
[значение_макроопределения]
```

Косая черта в конце первой строки указывает на то, что следующая строка служит продолжением макроопределения (см. далее). Если все макроопределение уместилось в одной строке, необходимости в косой черте нет. Точка с запятой здесь отсутствует, если только она не входит в состав значения_макроопределения. В действительности причин ставить ее на конце обычно не возникает. Как и прочие имена Си, ИМЯ_МАКРООПРЕДЕЛЕНИЯ должно удовлетворять общим правилам. Для того чтобы отличать имена макроопределений от прочих имен, во многих программистских сообществах действует соглашение использовать в них только прописные буквы, знак подчеркивания и арабские цифры. С момента введения макроопределения во всяком месте, где встречается ИМЯ_МАКРООПРЕДЕЛЕНИЯ, оно до компиляции дословно заменяется на значение_макроопределения.

Например:

```
#define A_PLUS_B ((a)+(b))
…
double a=2, b=3, c=4;
c+=A_PLUS_B;
```

В переменную c попадет значение 9, поскольку записанное в последней строке эквивалентно c+=((a)+(b)). С момента определения A_PLUS_B служит буквальным синонимом ((a)+(b)).

Если необязательное значение_макроопределения отсутствует, тогда ИМЯ_МАКРООПРЕДЕЛЕНИЯ всюду заменяется на «пустое место», т.е. его «начинка» игнорируется. Тем не менее само макроопределение, называемое ИМЯ_МАКРООПРЕДЕЛЕНИЯ, существует, но является пустым.

Одно из основных назначений #define — введение именованных констант и ОКВ.

Например:

```
#define E_BY_2 (2.718281828*2.0)
```

Далее по тексту числовое значение будет всюду подставляться вместо имени константы.

Например:

```
d*= E_BY_2;
```

умножит переменную `d` на `5.436563656`. Поскольку `2.718281828*2.0` является ОКВ, все, необходимое для его вычисления, доступно еще до компиляции. Препроцессор вычисляет значение ОКВ единожды, далее подставляя его всюду, где встречается `ИМЯ_МАКРООПРЕДЕЛЕНИЯ`.

При выборе способа введения константы следует сравнить возможности ее задания с помощью препроцессора или посредством модификатора `const`. Особенности каждого способа даны в табл. 10.2.

Таблица 10.2. Сравнение способов введения константы

Средства препроцессора	Модификатор `const`
Действует с момента определения до конца файла или до соответствующей директивы `#undef`, невзирая на границы программных блоков `{}`	Области существования и видимости совпадают с таковыми у переменных
Создается препроцессором до компиляции	Создается компилятором
Может быть отменена директивой `#undef` в произвольном месте исходного текста	Не может быть искусственно отменена в произвольном месте исходного текста
Может быть переопределена (изменено значение)	Не может быть изменена
Видна везде от определения до отмены определения или до конца файла	Может быть замаскирована (*перекрыта*) одноименной переменной или константой во внутреннем блоке

Однажды введенное макроопределение действует до конца файла, в котором оно введено. Однако имеется возможность сузить сферу его действия до меньшего фрагмента программы, используя директиву отмены макроопределения:

```
#undef ИМЯ_МАКРООПРЕДЕЛЕНИЯ
```

Эта директива осуществляет прекращение действия предшествующей директивы:

```
#define ИМЯ_МАКРООПРЕДЕЛЕНИЯ\
[значение_макроопределения]
```

С момента появления `#undef` ни препроцессор, ни компилятор «не знают» указанного макроопределения. Важная особенность директивы отмены макроопределения состоит в том, что, если `ИМЯ_МАКРООПРЕДЕЛЕНИЯ` предварительно не объявлено, препроцессор сообщит об ошибке и аварийно завершит работу. Ошибка возникает также при попытке повторно объявить макроопределение с тем же именем. При этом неважно, совпадает или нет второе определение с первым. Предварительное использование `#undef` устраняет предупреждение или сообщение об ошибке при необходимости переопределения существующего макроопределения.

Например:

```
#define MY_PI 3.14
a=MY_PI;
...
#undef MY_PI
...
#define MY_PI 3.1415926535
b=MY_PI;
```

В результате переменная `a` получит значение числа с точностью 3 десятичные цифры, а переменная `b` содержит 11 значащих десятичных цифр. Несмотря на такую возможность, переопределения макроопределений следует использовать как можно реже во избежание путаницы в программе.

10.4. Защита макроопределений

Корректное написание простых макроопределений требует заключения в круглые скобки каждой упомянутой в нем именованной «псевдопеременной» или «псевдоконстанты» и всего макроопределения. Это называется *защитой макроопределения* (англ. macro protection), а получившееся мак-

роопределение — безопасным (англ. safe macro, secure macro). О необходимости защищать макроопределения свидетельствует следующий пример. Введем незащищенное макроопределение с операцией невысокого приоритета внутри:

```
#define ABC a+b+c
```

В данном фрагменте кода:

```
long a=10, b=100, c=1000, d;
d=ABC;
```

незащищенное макроопределение работает правильно, и в переменную `d` попадает верное значение 1110. В этом фрагменте программы результат оказывается неожиданным:

```
long a=10, b=100, c=1000, d;
d=ABC*10;
```

Переменная `d` содержит не 11100, как хотелось бы, а 10110, поскольку после препроцессора на входе компилятора окажется:

```
long a=10, b=100, c=1000, d;
d=a+b+c*10;
```

Чтобы избежать подобной свойственной начинающим программистам оплошности, надо поместить тело макроопределения, возвращающего числовое значение, в круглые скобки, которые заставят сначала вычислить выражение, вводимое макроопределением, а затем использовать его во внешней программе:

```
#define ABC (a+b+c)
```

Но этого недостаточно. Так как макроопределения можно использовать внутри других макроопределений, при написании каждого из них нельзя быть уверенным в природе «псевдопеременных» или «псевдоконстант» `a`, `b` и `c`: они могут оказаться обычными переменными и константами, как в приведенном примере, а могут быть выражениями, определенными в других макроопределениях. Тогда снова можно неправильно задать приоритеты при буквальной подстановке макроопределения в макроопределение, если где-то будут забыты скобки.

Например:

```
#define X 15+x
#define Y (y-10)
```

```
#define XYC (X*Y*c)
double x=10, y=100, c=1000, d;
d=XYC;
```

Два первых макроопределения подставятся в третье, и результат снова окажется неверным:

```
d=(15+x*(y-10)*c);
```

Возможна ситуация, когда первое макроопределение находится в другом файле и программист не обратил внимания на отсутствие в нем скобок. Такую проблему выявить обычно непросто. Поэтому лучше перестраховаться с установкой скобок:

```
#define XYC ((X)*(Y)*(c))
```

Теперь на выходе препроцессора все в порядке независимо от того, как были написаны два первых макроопределения:

```
d=((15+x)*((y-10))*(c));
```

Изложенное не относится к макроопределениям, включающим в себя операторы. Они не застрахованы от так называемого эффекта «рассыпания».

Например:

```
#define SWAP_A_B tmp=a; a=b; b=tmp;
unsigned char a='A', b='Я', tmp;
SWAP_A_B /* сработало правильно */
if (a>b) SWAP_A_B /* макрос "рассыпался" */
```

Действительно, второй вызов `SWAP_A_B` на входе компилятора будет выглядеть так:

```
if (a>b) tmp=a; a=b; b=tmp;
```

В сферу действия оператора `if` попало только первое из трех присваиваний: `tmp=a`, а остальные два выполнятся всегда вне зависимости от результата сравнения.

Как метод консолидации тела макроопределения, содержащего операторы, используется его заключение в фигурные скобки:

```
#define SWAP_A_B { tmp=a; a=b; b=tmp; }
```

Недостатком этого решения является то, что при наличии непарных фигурных скобок внутри такого макроопределения компилятор неправильно назовет причину непорядка, указав

не на макроопределение, а на совершенно другое место в тексте программы, где оно используется. Эту неприятность призвано устранить более изощренное выделение тела макроопределения:

```
#define ИМЯ_МАКРООПРЕДЕЛЕНИЯ do\
{ тело_макроопределения } while (0)
```

Потенциально возможная проблема со скобками заведомо останется внутри цикла, и компилятор укажет именно на цикл `do...while`[1], подсказывая программисту истинную причину ошибки. Отсутствие на конце оператора `do...while` точки с запятой позволяет использовать макрос как оператор, сохранив единообразное написание:

```
if (a>b) SWAP_A_B;
```

10.5. Условная компиляция

Простая условная компиляция: `#if`, `#ifdef`, `#ifndef`, `#endif`. *Директива* `#if` по принципу действия напоминает оператор `if`:

```
#if ОКВ
код, который будет откомпилирован, если ОКВ истинно
#endif
```

Директива `#endif` служит для указания конца условно компилируемого блока, играя в препроцессоре роль, аналогичную роли `}` в компиляторе. Отличия от условного оператора заключаются в следующем:

◊ вне зависимости от значения проверяемого выражения в операторе `if` код, который должен исполниться в случае истинности этого значения, всегда будет откомпилирован и присутствует в программе. Напротив, если ОКВ на момент компиляции

[1] Применение для этих же целей операторов `while` или `for` также возможно, но требует больше усилий для обеспечения однократного выполнения тела цикла вследствие проверки вначале. Использование вырожденного `if(1){...}` приносит меньший эффект, так как в программе обычно число и вложенность условных операторов значительно превышают число и вложенность циклов `do...while`, поэтому труднее найти место ошибки, опираясь на диагностические сообщения компилятора.

дает значение «ложь», последующий фрагмент исходного кода будет выброшен из программы еще до компиляции;

◊ вследствие сказанного выше, `#if` в отличие от `if` управляет не ходом программы, а ее компиляцией.

Директива `#ifdef` включает последующий фрагмент исходного кода, если `ИМЯ_МАКРООПРЕДЕЛЕНИЯ` на данный момент определено.

```
#ifdef ИМЯ_МАКРООПРЕДЕЛЕНИЯ
код, который будет откомпилирован,
если макроопределение определено
#endif
```

Противоположное действие осуществляется директивой `#ifndef`.

```
#ifndef ИМЯ_МАКРООПРЕДЕЛЕНИЯ
код, который будет откомпилирован,
если макроопределение НЕ определено
#endif
```

Таким образом, макропроцессор дает возможность вводить новые имена констант и выражений, а также управлять компиляцией.

Например:

```
#ifndef MY_E
#define MY_E 2.71828182845904523536028747135 27
#include "my_math.h"
#endif
```

Альтернативные ветви: `#else, #elif`. Чтобы эффективнее управлять компиляцией, в Си предусмотрена директива `#else` с тем же смыслом, что и конструкция `else` в операторе `if`.

```
#if ОКВ
код, который будет откомпилирован, если ОКВ истинно
#else
код, который будет откомпилирован, если ОКВ ложно
#endif
```

Еще более сложные конструкции можно строить с применением директивы #elif, являющейся аналогом комбинации #else и #if.

Например:

```
#if ОКВ_1
код, который будет откомпилирован, если ОКВ_1 истинно
#elif ОКВ_2
код, который будет откомпилирован, если ОКВ_1 ложно и
ОКВ_2 истинно
#elif ОКВ_3
код, который будет откомпилирован, если ОКВ_1 и ОКВ_2
ложны, а ОКВ_3 истинно
#endif
```

10.6. Проверка множественных условий и defined

В директиве #if можно использовать произвольные комбинированные ОКВ. Если бы была возможность включать в них проверку самого факта существования тех или иных макроопределений, сложные составные условия стали более компактными без большой вложенности #ifdef. С этой целью в ОКВ (только внутри директив #if и #elif) можно применять следующее логическое ОКВ:

```
defined ИМЯ_МАКРООПРЕДЕЛЕНИЯ
```

Это выражение возвращает значение «истина», если ИМЯ_МАКРООПРЕДЕЛЕНИЯ было введено предшествующей директивой #define, иначе возвращает «ложь». С помощью defined удобно конструировать сложные составные ОКВ. Очевидно, следующие две конструкции эквивалентны:

```
#if defined ИМЯ_МАКРООПРЕДЕЛЕНИЯ
#ifdef ИМЯ_МАКРООПРЕДЕЛЕНИЯ
```

и следующая пара производит одинаковый эффект:

```
#if !defined ИМЯ_МАКРООПРЕДЕЛЕНИЯ
#ifndef ИМЯ_МАКРООПРЕДЕЛЕНИЯ
```

Часто ИМЯ_МАКРООПРЕДЕЛЕНИЯ в директиве defined заключают в скобки во избежание путаницы с приоритетами и для придания выражению функционального вида.

10.7. Управление диагностикой программы

Свои сообщения об ошибках (англ. error) и предупреждениях (англ. warning) препроцессор и компилятор обычно дают с указанием номеров строк, где проблема обнаружена. Иногда при работе с включением текста одного файла внутрь другого бывает полезно «подсунуть» компилятору иную нумерацию строк и даже иное имя файла:

```
#line КОНСТАНТА ["НОВОЕ_ИМЯ_ФАЙЛА"]
```

где КОНСТАНТА — целое положительное число — навязываемый для вывода диагностики препроцессором и компилятором номер строки, следующей за данной.

> Например, компилятор укажет на ошибку в строке 25 при обработке следующего фрагмента исходного текста программы вне зависимости от того, где этот фрагмент располагается на самом деле:
>
> ```
> #line 25
> while (i==k} a[i]=b[k];
> /* намеренно допущенная синтаксическая ошибка */
> ```

Директива

```
#line 12 "new_source_file_name.c"
b[0]=1.0;
```

заставляет препроцессор и компилятор полагать, что начиная со следующей строки они обрабатывают файл, имеющий имя

```
new_source_file_name.c,
```

а строка с присваиванием `b[0]` единицы имеет номер 12. Это отражается в их диагностических сообщениях.

Иногда бывает нужно, например при автоматическом обнаружении неправильного сочетания макроопределений или констант, еще на этапе компиляции программы вызывать специфическую ошибку с выводом диагностического сообщения и остановкой дальнейшей обработки исходного текста программы. Это осуществимо при помощи директивы `#error`, синтаксис которой очень прост:

```
#error текст_сообщения_об_ошибке
```

Здесь `текст_сообщения_об_ошибке` — текстовая константа без двойных кавычек. Предупреждающее сообщение выводится директивой

```
#warning текст_предупреждающего_сообщения
```

без аварийного прерывания препроцессирования и компиляции.

В инструментарий расширенной диагностики входят встроенные (англ. built-in), заранее определенные (англ. predefined, predetermined) макроимена (называемые в литературе также «макропеременные», «переменные препроцессора», «псевдопеременные»), доступные препроцессору во время обработки исходного текста программы. Везде, где препроцессор встретит эти макроимена, он заменяет их на конкретные значения. Всякое имя предопределенной макропеременной начинается и завершается двумя символами подчеркивания. Макроимена позволяют получить и использовать для любых целей в программе следующую информацию, справедливую на момент препроцессирования данного файла:

1) `__FILE__` – завершенная нулем строка символов, содержащая имя компилируемого файла. Имя изменяется всякий раз, когда препроцессор встречает директиву `#include` с указанием имени другого файла. Когда включение файла по команде `#include` завершается, восстанавливается предыдущее значение макроимени `__FILE__`. Текущее значение макроимени можно изменить принудительно директивой `#line`. Оно действует до следующего переопределения или конца файла;

2) `__LINE__` – десятичная положительная константа – номер текущей обрабатываемой строки файла с программой на Си. Принято, что номер первой строки исходного файла равен 1. Возможен такой вывод диагностики:

```
printf("\nline %d in file %s passed",
__LINE__, __FILE__);
```

3) `__DATE__` – завершенная нулем строка символов в формате: «месяц число год», определяющая дату начала обработки исходного файла. Например, следующий оператор выведет дату препроцессирования данного файла:

```
printf("\nDate of preprocessing: %s", __DATE__);
```

4) `__TIME__` – завершенная нулем строка символов вида «часы:минуты:секунды», соответствующая времени начала обработки препроцессором исходного файла. Пример использования:

```
char buffer[1000];
strcpy(&buffer[0]," preprocessed at ");
strcat(buffer, __TIME__);
```

5) __STDC__ – константа, равная 1, если компилятор работает в соответствии с ISO/IEC-стандартом. В противном случае значение макроимени __STDC__ либо не определено, либо имеет иное значение. Стандарт языка Си предполагает, что наличие __STDC__ определяется реализацией, так как макроимя __STDC__ относится к нововведениям стандарта. Его возможное применение:

```
#if __STDC__!=1
#   warning This is not a standard compiler
#endif
```

В конкретных реализациях набор предопределенных имен нередко дополняется нестандартными именами. Более полные сведения о предопределенных препроцессорных именах можно найти в документации по данному компилятору.

10.8. Расширенное использование макроопределений

Препроцессор предоставляет значительно более мощные возможности, чем инструмент введения констант и средства условной компиляции. С помощью директивы #define можно создавать завершенные функциональные конструкции.

Например:

```
#define NEW_LINE printf("\n")
```

Всякий раз, когда в исходном тексте программы встречается лексема NEW_LINE, она заменяется на printf("\n"). Знатоки языка Pascal имеют техническую возможность ввести в Си следующие макроопределения:

```
#define begin {
#define end }
```

После этого исходный текст приобретает нестандартный вид.

Например:

```
if (a>b) begin c=2*d; d++; end
```

Подобные определения не относятся к хорошему стилю программирования, поскольку не улучшают полезные свойства или читаемость программы.

Как указывалось в § 10.1, конструкции препроцессора занимают одну строку. При введении большого макроопределения его пришлось бы записывать неудобочитаемым образом, если бы не было символа продолжения макроса на следующую строку '\' (склеивания строк). Это в действительности тот же символ, который рассматривался в § 3.4 при переносе длинных текстовых констант на новую строку. Этот символ обрабатывается препроцессором, вызывая непосредственную пристыковку к концу предшествующей ему строки начала последующей. Символ завершения строки удаляется, и на вход компилятора поступает сплошная объединенная строка. Такой трюк можно осуществлять последовательно несколько раз.

Например:

```
#define REVERSE_ARRAY_A do\
{unsigned long i; for (i=0;i<I/2;++i)\
{ tmp=a[i]; a[i]=a[I-1-i]; a[I-1-i]=tmp; }} while(0)
```

Отсутствие закрывающей косой черты '\' у последней строки макроса сообщает о его завершении, другими словами, о том, что следующая строка к нему не относится. В теле макроопределения можно пользоваться внешними по отношению к нему переменными и константами (`tmp` в приведенном примере). При этом требуется осторожность, чтобы их полезное значение вне макроса не повредить внутри него, а также соблюсти соответствие типов. Если тело макроса заключено в фигурные скобки, т.е. образует блок, в нем можно создавать свои внутренние переменные и константы (`unsigned long i` в приведенном примере). Это значительно безопаснее, но исполнимый код получается немного менее эффективным вследствие необходимости выделения и освобождения занимаемого ими места при всяком обращении к макросу.

10.9. Макроопределения с параметрами

Параметрические (или параметризованные, англ. parametrized) макроопределения являются одной из наиболее сложных и гибких возможностей препроцессора. По способу записи и производимым действиям они похожи на функции и в некоторых случаях могут заменять вызовы настоящих функций или служить короткой заменой часто повторяющихся логически аналогичных фрагментов текста. При вызове извне макроопределению передаются параметры, которыми оно может оперировать как функция. Список параметров приводится сразу следом за именем макроопределения в круглых скобках без пробелов[1].

Общая форма:

```
#define ИМЯ_МАКРООПРЕДЕЛЕНИЯ\
([имя_параметра,]^n имя_параметра) [тело_макроопределения]
```

При этом никогда не указывается тип передаваемых параметров макроопределения, поскольку это невозможно и не нужно.

Во-первых, типизация переменных — это особенность языка Си, а не препроцессора, который просто осуществляет манипуляции с фрагментами текста программы, «не понимая», что этот текст означает.

Во-вторых, отсутствие типизации в макроопределениях позволяет при некоторой изобретательности и аккуратности создавать макросы, реализующие алгоритмы или их фрагменты в «чистом виде», без привязки к конкретным типам данных. В этом смысле они предоставляют программисту инструмент, отдаленно напоминающий шаблоны в C++.

В-третьих, в макроопределениях указание типов таким же образом, как в функциях, просто невозможно, так как придется записывать весь список параметров без единого пробела. Это связано с тем, что препроцессор отделяет имя макроса от его содержимого по первому пробельному символу (пробел

[1] Здесь в отличие от ряда руководств считается недопустимой установка пробелов после имени макроопределения перед открывающейся круглой скобкой с параметрами макро, а также где-либо в списке параметров макроопределения потому, что в ряде препроцессоров заголовок макроопределения отделяется от его тела по месту расположения первого пробела.

или табуляция) после имени макроопределения. Список параметров относится к имени макроса, поэтому записывается слитно. Другие способы отделения имени от тела макроопределения не являются универсальными и гибкими, так как макроопределение может быть произвольным и содержать практически любые разрешенные в Си лексемы. В отличие от заголовка в теле макроопределения использование пробелов неограниченно.

В качестве простейших примеров макроопределений приведем нахождение модуля и максимума пары чисел, а также произведения трех чисел[1]:

```
#define abs(x) (((x) > 0)?(x) : -(x))
#define min(x,y) ((x)<(y)?(x):(y))
#define mul3(a,b,c) ((a)*(b)*(c))
```

У первого макроса один параметр, у второго – два, а у третьего – три. Тела макроопределений заключены в круглые скобки, чтобы избежать эффекта «рассыпания». Все они являются макросами-выражениями и возвращают числовое значение.

Крайне важно, что параметры макроопределений внутри тела также окружены круглыми скобками; такой прием называется *изоляцией параметров макроопределения* (англ. macro parameter insulation). Дело в том, что в качестве параметров на вход могут быть поданы не только имена переменных и констант, но и произвольные выражения. Так как подстановка параметров в макроопределение осуществляется препроцессором, он всюду внутри макроса «чисто механически» заменяет этими выражениями имя данного параметра. При отсутствии скобок следующий вызов третьего макроопределения привел бы к неверной работе:

```
#define mul3(a,b,c) (a*b*c)
int x=-2, y=3, z=4;
x=mul3(x+y,y+z,z+x);
/* компилятор получит на входе: x=x+y*y+z*z+x */
```

Как уже отмечалось, в виде макросов можно оформлять не только выражения, но и последовательности операторов. Такие макроопределения называются *операторными*. Если они

[1] Обратите внимание, что эти макросы годятся для параметров любого числового типа.

образуют блок, допускается даже заводить собственные внутренние переменные.

Например:

```
#define copy_vector(source,destination,first,last) do\
{int k; for(k=(first);k<=(last);k++)\
(destination)[k]=(source)[k]; } while(0)
```

Внутренние переменные не нужно заключать в скобки, так как их природа понятна и не может привести к неприятным сюрпризам. Мощность препроцессора открывает возможность реализации целых алгоритмов как макроопределений.

Например:

```
#define sort_vector(vector,length) do\
{ int j, unsorted; double tmp;\
  do\
    {\
    for (j=1, unsorted=0; j<length; j++)\
      if (vector[j-1]>vector[j])\
        { tmp=vector[j-1];\
        vector[j-1]=vector[j];\
        vector[j]=tmp; unsorted=1; }\
    }\
  while (unsorted); }\
while (0)
```

Приведенное выше макроопределение осуществляет сортировку элементов вектора `vector` произвольного типа длиной `length`. Поскольку запись алгоритма содержит собственные фигурные скобки, желательно изолировать тело макроопределения от «внешней среды» не просто фигурными скобками, а при помощи вырожденного оператора `do...while`, что упрощает поиск ошибок. Благодаря наличию в каждой строке, кроме последней, символа принудительного закрытия строки с продолжением на следующей строке весь макрос размещен в одну строку на входе компилятора, что является обязательным требованием. Соответственно на эту строку будут ссылаться при возникновении проблемы и сообщения компилятора. Очевидно, это уже будет не номер строки, где определен макрос, а номер одной из тех строк, где он применяется.

Логичен вопрос: когда предпочтительно применение макроопределений, когда встраиваемых (`inline`) функций, а когда обычных функций? Выбор до некоторой степени определяется вкусом программиста, однако есть объективные предпосылки, сведенные в табл. 10.3.

Таблица 10.3. Сравнение функций и макроопределений

Традиционная функция	Встроенная функция	Макроопределение
Требует накладных расходов по передаче параметров при вызове	Не требует передачи параметров, так как не происходит вызова функции в традиционном понимании: текст встроенной функции или макроопределения подставляется во всякое место вызова	
Имеется полная проверка типов данных при передаче		Нет проверки типов данных до компиляции, да и та слабее, чем у функций
Исполнимый код наиболее компактен для больших функций	Итоговый код получается бóльшим, поскольку код встроенной функции или порождаемый макроопределением вставляется столько раз, сколько есть вызовов	
Возможна рекурсия	Рекурсия невозможна	
Возможно переменное число входных параметров	Переменное число входных параметров не поддерживается	
Сложность алгоритма произвольная	Сложность алгоритма невысока, во многих реализациях не разрешается использование условного оператора и циклов	Сложность алгоритма невысока, ограничениями являются размер исполнимого кода и сложность отладки макроопределения
Скорость выполнения кода невысока из-за передачи параметров	Скорость выполнения кода несколько выше	
Невозможно создание шаблонов – реализаций алгоритмов, не зависящих от типов данных		Принципиально возможно создание шаблонов с очень ограниченными по сравнению с C++ возможностями
Риск повредить внешние по отношению к функции данные отсутствует при грамотном программировании		Повышен риск испортить внешние данные, так как внутреннее пространство имен не полностью изолировано от внешнего

Все три инструмента помогают реализовать принципы структурного и функционального программирования, выделяя многократно используемые или функционально обособленные части программы.

10.10. Вложенные макроопределения

Действующие макроопределения могут быть использованы для введения новых.

Например:

```
#define MY_PI 3.14
#define MY_PI4 ((MY_PI)/4.0)
#define MY_SIN_PI4 (sin(MY_PI4))
…
#undef MY_PI
#define MY_PI 3.1415926535
…
printf("\n sin(MY_PI4)=%f",MY_SIN_PI4);
```

Возникает вопрос: в какой момент времени макроопределение *будет раскрыто*, т.е. будет вычислено `MY_SIN_PI4`, — в момент определения или в момент использования в функции `printf`? В зависимости от ответа будут получены разные результаты. Стандарт предусматривает раскрытие макроопределения всегда в момент его использования. При этом происходит рекурсивный вызов препроцессора, вложенность которого в современных средствах неограниченна. В функции `printf` надо раскрыть макроопределение `MY_SIN_PI4`. Для этого препроцессор обратился к его определению, и потребовалось знание `MY_PI4`. Чтобы выяснить его значение, препроцессор снова вызывает себя и обнаруживает необходимость предварительно знать значение `MY_PI`. Он еще раз вызывает себя, находит последнее актуальное на данный момент определение `MY_PI` и применяет его, что дает возможность благополучно завершить рекурсию. Таким образом, в вычислении `MY_SIN_PI4` в функции `printf` примет участие более позднее и точное значение `MY_PI`.

10.11. Скрытые проблемы в макроопределениях

Множество потенциально возможных ошибок в макроопределениях неисчерпаемо, но целесообразно остановиться на характерных побочных эффектах, связанных с непреднамеренным изменением внешних по отношению к макроопределению переменных. Им подвержены макроопределения с параметрами (см. § 10.9), в которые могут быть переданы выражения, модифицирующие значения переменных. В качестве примера рассмотрим введенные в § 10.9 макроопределения:

```
#define min(x,y) ((x)<(y)?(x):(y))
#define mul3(a,b,c) ((a)*(b)*(c))
```

Ничто не запрещает их применить так:

```
float a=-14, b=-9, c=2, d, e, f, g, h;

d=min(a+=3,b);
e=mul3(++c,++c,10); c=2;
f=mul3(++c,c++,10); c=2;
g=mul3(c++,++c,10); c=2;
h=mul3(c++,c++,10);
```

В результате переменная `a` будет увеличена на 3 дважды: первый раз в проверяемом логическом выражении трехместной операции, а второй раз – в выбранном возвращаемом выражении и `d` получит значение –8, о котором можно сказать, что оно неожиданно и больше, чем `b=-9`.

Второе макроопределение тоже преподносит сюрприз. При вычислении `e` произойдет перемножение: 4*4*10=160, а при вычислении `f` и `g` результат равен 3*3*10=90, а `h`=2*2*10=40 вопреки ожиданиям.

Избежать таких странностей можно двумя способами:

- ◊ не употреблять в качестве параметров макроопределений выражения, изменяющие переменные, хотя это ограничение снижает выразительность программирования;
- ◊ оформлять макроопределения как блоки, создавая внутри них временные автоматические переменные, в которые при вызове макроопределения попадают значения параметров, но это снижает эффективность кода и привязывает макроопределения к конкретным типам данных, что не всегда приемлемо.

10.12. Манипуляции лексемами и макроопределения с параметрами

Директива `#define` позволяет «склеивать» лексемы как строки. Для этого достаточно разделить их знаками `##`. Препроцессор объединит такие лексемы в одну.

Например, есть определение

```
#define full_name(prefix,suffix) prefix##suffix
```

При вызове `full_name(Double,Array)` оно автоматически в программе образует идентификатор `DoubleArray`.

Например:

```
full_name(Double,Array)[i]=100;
```

эквивалентно

```
DoubleArray[i]=100;
```

Одиночный символ `'#'`, размещаемый в `#define` перед аргументом макроопределения, показывает, что следующий за ним аргумент должен быть преобразован в символьную строку (грубо говоря, препроцессор возьмет его имя), а не будет взят по значению, как обычно. Другими словами, конструкцию вида

```
#формальный_параметр
```

препроцессор заменит на конструкцию

```
"фактический_параметр".
```

Например, при отладке программы может пригодиться макроопределение:

```
#define out_int(var) printf (#var "=%d\n",var)
```

Фрагмент исходного текста:

```
int i=0; k=10;
…
out_int(i);
…
out_int(k);
```

препроцессор превратит в следующий:

```
int i=0; k=10;
```

```
…
printf ("i=%d\n",i);
…
printf ("k=%d\n",k);
```

Таким образом, можно без лишних усилий выводить на экран значения переменных в формате имя=значение. Очевидно, операции препроцессора # и ## могут встречаться только в теле макроопределения, вводимого #define. Извне их действие доступно лишь путем вызова этого макроопределения.

10.13. Введение альтернативных имен: typedef против #define

В целом использование typedef более эффективно, чем директивы #define, для создания лаконичного альтернативного имени типа, например структуры. В случае препроцессора у компилятора меньше возможностей для оптимизации: ему требуется «догадаться», что подставляемые препроцессором в разных местах программы объемные определения эквивалентны. Конструкция typedef оказывается изящнее и при введении на основе некоторого типа производных типов – указателя, массива, массива указателей и т.д. Поэтому везде, где возможно, следует отдавать предпочтение typedef.

Однако преимущество typedef не абсолютно. Следующая завершенная программа иллюстрирует гибкость препроцессора при конструировании нового типа по шаблону:

```
#include <stdio.h>

/* макрос с параметром позволяет гибко конструировать ти-
пы структур */
#define my_s(type_a) struct {type_a a; unsigned b;\
 double *c; }

int main()
{
double z=5.91e-12;
```

```
/* производим структуры первого вида: первое поле типа
int */
my_s(int) xi={-999,0xFF00FF00,&z},
  yi[2]={{777,0X12345678,&z},{-555,0xABCD,0}};
/* производим структуры второго вида: первое поле типа
long double */
my_s(long double) xd={1.234e5,0x00FF00FF,&z},yd[2]=
  {{-0.3456789,0x999A999A,&z},{123.456,0x1A2B3C4D,0}};
unsigned i;

printf("\n xi={%d, 0x%x, %p}",xi.a,xi.b,xi.c);
printf("\n xd={%Lg, 0x%x, %p}",xd.a,xd.b,xd.c);

for (i=0;i<sizeof(yi)/sizeof(my_s(int));i++)
  {
  printf("\n yi[%u]={%d, 0x%x, %p}",
    i,yi[i].a,yi[i].b,yi[i].c);
  printf("\n yd[%u]={%Lg, 0x%x, %p}",
    i,(yd+i)->a,(yd+i)->b,(yd+i)->c);
  }
return 0;
}
```

Даже из этого простого примера видны идеи гибкого порождения новых типов силами препроцессора. Привлечение манипуляции лексемами одновременно увеличивает масштаб возможностей и вероятность запутаться. На этом пути требуются повышенные внимание и осторожность, поскольку у препроцессора значительно меньше инструментов контроля правильности конструкций и побочных последствий, чем у компилятора.

10.14. Директивы #pragma

Многие средства разработки предлагают различные нестандартные расширения языка Си, зависящие от операционной системы и аппаратных средств компьютера. Для использования этих дополнительных возможностей в стандарте предусмотрена «лазейка»: в исходном тексте программы размещается директива вида:

```
#pragma текст_расширительной_директивы
```

Если препроцессор поддерживает такую `расширительную_директиву`, он об этом сообщает компилятору. Иначе, согласно стандарту, `#pragma` должна игнорироваться, но это происходит, к сожалению, не всегда и иногда приводит к необычным сообщениям препроцессора или компилятора.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Небольшую процедуру с тремя входными параметрами и одной внутренней переменной (все типа `long`) реализовали в двух вариантах: *а*) при помощи встроенной `inline` функции и *б*) безопасного макроса. Что можно сказать о быстродействии обоих вариантов? Проведите эксперимент.
2. Выясните, как в вашем компиляторе можно получить текстовый вывод результатов обработки текста программы препроцессором. Поэкспериментируйте с результатами препроцессинга двух-трех небольших программ, использующих директивы `#define` и `#undef`.
3. Как создать макроопределения, гарантированно лишенные побочных действий?
4. Всякому используемому в вашей программе типу данных (не менее трех) при помощи препроцессора сопоставьте очень короткое альтернативное название, например:
```
#define sc signed char
#define ull unsigned long long
```
Можно ли реализовать макроопределение с параметрами
`make_var(короткое_имя_типа,имя_переменной)`,
получающее на вход короткое название типа и имя переменной и создающее объединенное имя переменной, начинающееся с `короткого_имени_типа`, затем следует символ подчеркивания, а в завершение — `имя_переменной`. Обоснуйте ваш ответ, если он отрицательный, иначе реализуйте это макроопределение.
5. Реализуйте макроопределение, автоматически выводящее в месте применения:
а) имя файла исходного текста программы;
б) номер текущей строки в файле с исходным текстом программы;
в) переданное в виде параметра макроопределения имя переменной;
г) значение этой переменной в данном месте программы.
6. Можно ли средствами препроцессора ввести зависящий от трех параметров тип двухмерного массива:
`тип_массива [размерность_1][размерность_2]`
Если можно, запрограммируйте, если нет, аргументируйте почему.
7. При помощи препроцессора реализуйте два варианта одной функции: с двумя принимаемыми параметрами и с тремя принимаемыми пара-

метрами (первые два совпадают с двухпараметрическим вариантом). Запрограммируйте определение функции, ее прототип и вызов из `main`. Обязательно ли писать тело функции дважды? Можно ли оба варианта использовать в программе одновременно?

8. Некоторому целочисленному типу при помощи `typedef` или препроцессора дано новое имя (старое неизвестно). Имеется переменная этого типа. Как выяснить, это знаковый тип или нет?

9. Реализуйте пару безопасных реентерантных макросов, первый из которых будет запоминать номер строки в том месте исходного текста, где он применен, а второй – распечатывать разность строк между номером строки, где он вызван, и номером строки вызова первого макроса. Обязательно ли у каждого из этих макросов должны быть параметры?

10. Предположим, что архитектура вашего компьютера поддерживает одно- двух- и четырехбайтные беззнаковые целочисленные типы. Как на базе встроенных типов ввести пользовательские типы гарантированной разрядности `uint8`, `uint16` и `uint32`? Сделайте так, чтобы ваша реализация вела себя правильно в том смысле, что при наличии хотя бы одного подходящего встроенного типа для каждого из вновь вводимых типов он брался, а при отсутствии подходящего типа возникала бы ошибка компиляции. Продемонстрируйте правильность своего подхода на примере короткой программы.

Рабочая тетрадь для контроля знаний

К материалу каждой главы пособия ниже даны тесты для проверки качества усвоения материала. В отличие от контрольных вопросов и заданий в конце глав приведенные ниже тесты предусматривают быстрые короткие ответы. Предлагаются тесты трех типов:
а) впишите недостающие формулировки или термины в соответствующие поля;
б) зачеркните неправильные варианты ответа среди нескольких предложенных;
в) ответьте на вопросы.

Тесты к введению

1. Последовательное устройство работы с данными характеризуется _____
_____
_____
_____

2. Утверждение: «Драйверы являются разновидностью прикладного программного обеспечения» [ИСТИННО] [ЛОЖНО]

3. Высказывание: «Язык Си пригоден для создания любых типов программного обеспечения» [ИСТИННО] [ЛОЖНО]

4. Оптимизация программы – это _____
_____
_____
_____

5. Программный интерфейс приложения – это _____
_____
_____
_____

6. Утверждение: «Зарезервированные слова языка программирования являются частным случаем его атомов» [ИСТИННО] [ЛОЖНО]

7. Лексема – это _____
_____
_____
_____

8. Формулировка: «Интерпретация отличается от компиляции пошаговым выполнением конструкций текста программы без перевода всей программы в исполнимый код процессора» [ИСТИННА] [ЛОЖНА]

9. Разделителями в языке Си являются следующие символы: _____

10. Утверждение: «Обработка текста программы осуществляется в такой последовательности: компиляция → препроцессирование → сборка модулей» [ИСТИННО] [ЛОЖНО]

11. Верно ли, что выражение состоит из операций над операндами? [ДА] [НЕТ]

12. Оператор отличается от операции тем, что _____

К главе 1

1. Знаковый бит в числе может находиться:

а) В разных местах, в зависимости от того, число представлено в целом формате или с плавающей точкой [ДА] [НЕТ]

б) Всегда в самом последнем (правом) разряде [ДА] [НЕТ]

в) Всегда в первом (левом) разряде [ДА] [НЕТ]

г) В зависимости от типа процессора компьютера [ДА] [НЕТ]

2. Справедливо ли утверждение «Нуль в знаковом разряде числа всегда является признаком отрицательности числа»? [ДА] [НЕТ]

3. Перечислите все беззнаковые типы данных языка Си: _____

4. Высказывание: «Всякое рациональное число в поддерживаемом компьютером диапазоне значений всегда может быть точно представлено одним числом с плавающей точкой» [ИСТИННО] [ЛОЖНО]

5. Утверждение: «Узкоспециализированные целочисленные типы данных полезны в некоторых программах, но без них всегда можно обойтись» [ИСТИННО] [ЛОЖНО]

6. Перечислите спецификаторы хранения, исключающие хранение переменной на стеке: _____

7. Назовите спецификаторы хранения, превращающие определение переменной или константы в ее декларацию: _____

8. Имеет ли смысл сочетание `volatile const`? [ДА] [НЕТ]

9. Спецификатор хранения `static` нужен для _____

10. Если не указать ни одного спецификатора хранения, переменная будет размещена _____

11. Может ли ошибочное включение спецификатора хранения `volatile` привести к неправильной работе программы? [ДА] [НЕТ] [ИНОГДА]

12. Допустимо ли сочетание спецификаторов хранения `auto` и `static`? [ДА] [НЕТ] [ИНОГДА]

13. Выпишите все пары взаимоисключающих спецификаторов хранения: _____

14. Выпишите все встроенные типы данных, у которых отсутствие модификаторов `signed` и `unsigned` приводит к неопределенности в отношении возможности хранения отрицательного значения: _____

К главе 2

1. Может ли имя константы состоять из подчеркивания, за которым следует одна цифра? [ДА] [НЕТ]

2. Тезис: «Имя переменной может включать в себя зарезервированное слово как составную часть, но не должно с ним совпадать»
[ИСТИНЕН] [ЛОЖЕН]

3. Допустимо ли в имени переменной использовать цифры? [ДА] [НЕТ]

4. Будут ли различаться переменные, если имя одной из них получено из имени другой заменой строчной буквы на заглавную? [ДА] [НЕТ]

5. Блок программы – это _____

6. Сделать временно невидимой переменную или константу можно при помощи _____

7. Утверждение: «Определение переменной или константы всегда приводит к выделению для нее памяти» [ИСТИННО] [ЛОЖНО]

8. Если константа введена при помощи модификатора `const`, можно ли впоследствии в этом же блоке программы переопределить ее значение? [ДА] [НЕТ]

9. Строковая константа отличается от символьной наличием _____

_____

_____

_____

10. Верно ли, что символьная константа является просто формой записи числовой константы и ее числовое значение ни от чего не зависит? [ДА] [НЕТ]

11. Высказывание: «Область определения переменной или константы всегда шире ее области видимости» [ИСТИННО] [ЛОЖНО]

12. Утверждение: «Символьная константа является частным случаем целой константы» [ИСТИННО] [ЛОЖНО]

13. Тезис: «Положительное число некоторого целого знакового типа всегда будет автоматически корректно преобразовано в такой же беззнаковый тип» [ИСТИНЕН] [ЛОЖЕН]

14. В разных блоках одной и той же программы определены одноименные переменные. Всегда ли при этом верно утверждение: «Это абсолютно разные переменные и в общем случае их можно сделать разного типа»? [ДА] [НЕТ]

15. Верно ли высказывание: «Инициализация автоматической переменной в месте ее определения в блоке будет происходить при всяком попадании в этот блок»? [ДА] [НЕТ]

К главе 3

1. В неинициализированной области массива находится _____

_____

_____

_____,

что является по своей сути _____

_____

_____

2. Адреса соседних элементов вектора-массива отличаются на величину ____

_____

_____

_____

3. Чтобы массив проинициализировать не полностью, необходимо _____

_____

_____

_____

4. Утверждение: «Трехмерный массив в Си – это вектор векторов, каждый из которых является вектором базовых элементов массива» [ИСТИННО] [ЛОЖНО]

5. Указатели можно использовать в сдедующих целях:
а) Ускорение работы программы [ДА] [НЕТ]
б) Повышение безопасности и надежности программы [ДА] [НЕТ]
в) Увеличение гибкости организации данных [ДА] [НЕТ],\
г) Уменьшение объема памяти, занимаемого данными [ДА] [НЕТ].

6. Над указателями можно осуществлять следующие операции: _____

_____

_____

_____

7. Утверждение: «Возможности скобочной индексации элемента массива тождественны возможностям доступа к ним по указателю» [ИСТИННО] [ЛОЖНО]

8. Спецификатор `restrict` нужен для _____

_____

_____

_____

9. Модификатор `const` для указателя:
а) Позволяет экономить память [ДА] [НЕТ]
б) Снижает возможность допустить незамеченную ошибку [ДА] [НЕТ]
в) Повышает быстродействие программы [ДА] [НЕТ]

10. Вычитание двух указателей недопустимо, если они _____

_____

_____

_____

11. При выделении места для строковой константы необходимо иметь в виду особенность ее хранения _____

_____

_____

_____

К главе 4

1. Использование избыточных пар круглых скобок при задании приоритета операций в выражении вызывает:
а) Заметное повышение времени компиляции программы [ДА] [НЕТ]
б) Увеличение времени работы программы [ДА] [НЕТ]
в) Потенциальные проблемы при автоматической оптимизации программы компилятором [ДА] [НЕТ].

2. Результатом выполнения любой операции является _____

_____

_____

_____

3. Результат выполнения логической операции всегда имеет тип _____

_____

_____

_____

4. Точка следования – это _____

_____

_____

_____

5. Операция `sizeof` дает размер объекта, выраженный в _____

_____

_____

_____

6. Верно ли высказывание «Операция явного преобразования типа может исказить преобразуемое значение» [ДА] [НЕТ]

7. Тип сложного (составного) выражения определяется _____

_____

_____

_____

8. Может ли возникнуть ситуация, когда все выражение справа от операции присваивания необходимо заключить в круглые скобки? [ДА] [НЕТ]

9. Верно ли утверждение: «Значения всех выражений, стоящих слева от самого правого знака `=` при цепочечном присваивании, будут утрачены»? [ДА] [НЕТ]

10. Всегда ли можно обойтись без круглых скобок при вложенной трехместной операции? [ДА] [НЕТ]

11. Верно ли высказывание: «Результатом выполнения логической операции могут оказаться только числа 0 и 1»? [ДА] [НЕТ]

12. Соответствует ли действительности утверждение: «Результатом выполнения побитовой операции может оказаться любое целое число»? [ДА] [НЕТ]

13. Операция сдвига вправо для знаковых и беззнаковых целых отличается тем, что _____

_____

_____

_____

14. Верно ли, что при сдвиге влево положительного знакового числа оно может стать отрицательным? [ДА] [НЕТ]

К главе 5

1. В отличие от всех остальных типов данных структуры позволяют хранить

_____

_____

_____

2. Можно ли точно выяснить размер поля структуры путем простого вычитания указателей на поля? [ДА] [НЕТ]

3. Эквивалентны ли выражения: `g->h` и `(*g).h`? [ДА] [НЕТ]

4. Структура `s` включена в состав структуры `t` как ее поле. Ко всем ли полям структуры `t` можно обратиться, зная указатель на `s`? [ДА] [НЕТ]

5. Применение `typedef`:
а) Просто дает типу новое имя [ДА] [НЕТ]
б) Способно существенно улучшить читаемость программы [ДА] [НЕТ]
в) Повышает скорость работы программы [ДА] [НЕТ]
г) Осуществляет диагностику ошибок при введении нового типа данных [ДА] [НЕТ]
6. Верно ли утверждение: «Перечислимый тип не является необходимым в языке, но полезен для улучшения качества программ»? [ДА] [НЕТ]
7. Можно ли при помощи `typedef` ввести такой тип данных, который иначе создать нельзя? [ДА] [НЕТ]
8. Битовые поля и структуры:
а) Похожи только по форме записи, иных общих свойств у них нет [ДА] [НЕТ]
б) Требуют различных способов программирования обращения к полям [ДА] [НЕТ]
в) Подвержены однотипному выравниванию по границе слова [ДА] [НЕТ]
г) Могут быть взаимно вложены друг в друга (т.е. как битовое поле в структуру, так и структура в битовое поле) [ДА] [НЕТ]
9. Объединения и структуры:
а) Похожи только по форме записи, иных общих свойств у них нет [ДА] [НЕТ]
б) Имеют различные способы программирования обращения к полям [ДА] [НЕТ]
в) Подвержены однотипному выравниванию по границе слова [ДА] [НЕТ]
г) Могут быть взаимно вложены друг в друга (т.е. как объединение в структуру, так и структура в объединение) [ДА] [НЕТ]
10. Структуры:
а) В принципе могут быть вложенными [ДА] [НЕТ]
б) Могут содержать вложенную структуру того же типа внутри себя [ДА] [НЕТ]
в) Могут содержать указатель на тот же структурный тип [ДА] [НЕТ]
г) Содержат поля, всегда подвергающиеся одному и тому же выравниванию для одного типа структуры и одного вида компьютера [ДА] [НЕТ]
д) Содержат поля, выравнивание которых зависит только от типа структуры и настроек компилятора [ДА] [НЕТ]

К главе 6

1. Составной оператор – это _____
_____
_____
_____
2. Справедливо ли утверждение: «Без применения пустого оператора всегда можно обойтись»? [ДА] [НЕТ]
3. Верно ли утверждение: «Для записи произвольного алгоритма достаточно операторов `if` и `goto`, операторы цикла удобны, но не обязательны»? [ДА] [НЕТ]

4. Для реализации дерева решений путем сравнений величины с эталоном при `n` возможностях достаточно _____ операторов `if`.

5. Следует избегать применения оператора `goto`, за исключением ситуаций, когда _____

_____

_____

_____

6. Верно ли, что при большом числе `case` и существенно меньшем числе `break` эффективность оператора `switch` выше, чем аналогичной конструкции из операторов `if`? [ДА] [НЕТ]

7. Управление безопасно передавать внутрь программного блока при помощи оператора безусловного перехода только тогда, когда _____

_____

_____

_____

8. Верно ли утверждение: «При помощи исключительно управляющих операторов `switch`, `break`, `if` можно реализовать цикл»? [ДА] [НЕТ]

9. Оператор `continue` нужен для того, чтобы _____

_____

_____

_____

10. Верно ли высказывание: «Неправильное использование операторов `break` и `continue` потенциально может привести к тому, что переменная окажется не проинициализированной»? [ДА] [НЕТ]

11. Справедливо ли утверждение: «Оператор цикла `do...while` более опасен, чем другие операторы цикла, из-за непредсказуемости последствий первой итерации»? [ДА] [НЕТ]

12. В операторе `for`:

а) Все три секции в заголовке могут быть пустыми, и подобный цикл не просто синтаксически допустим, но является осмысленным при использовании `break` [ДА] [НЕТ]

б) Может быть несколько параллельно изменяющихся и одновременно проверяемых переменных цикла [ДА] [НЕТ]

в) Приращение переменной цикла может быть неравномерным (т.е. переменная цикла от итерации к итерации изменяется на разную величину) [ДА] [НЕТ]

г) Могут быть предусмотрены дополнительные проверки условия продолжения в середине и даже в конце тела цикла [ДА] [НЕТ]

К главе 7

1. Может ли существовать работоспособная полная Си-программа, не использующая ни одной функции? [ДА] [НЕТ]

2. Декларация функции нужна для того, чтобы _____

_____

Декларацию функции также называют _____

3. Справедливо ли высказывание: «В декларации функции допустимо опустить имена параметров, оставив лишь перечисление их типов через запятую»? [ДА] [НЕТ]

4. Большой массив внутрь функции можно передать следующими способами: _____

5. Верно ли утверждение: «Не существует средств по переданному в функцию указателю на некоторый элемент массива автоматически определить размерность и размеры этого массива»? [ДА] [НЕТ]

6. Выделенный в памяти внутри функции массив может быть передан вовне функции посредством _____

7. Выделенный в памяти внутри функции массив может быть безопасно использован вне функции только в случае, если _____

8. Обмен данными с функцией через стек с применением указателей:

а) Позволяет передать как угодно много однородных элементов данных из функции при малом объеме стека [ДА] [НЕТ]

б) Позволяет вернуть как угодно много однородных элементов данных из функции при малом объеме стека [ДА] [НЕТ]

в) Вынуждает программиста явно выделить соответствующее место на стеке, а затем освободить его [ДА] [НЕТ]

г) Вынуждает программиста передавать размеры массивов явно [ДА] [НЕТ]

9. Явное преобразование типа фактического параметра при передаче в функцию требуется, если _____

10. Неявное преобразование типа фактического параметра при передаче в функцию происходит, если _____

11. Три точки после запятой в конце списка формальных параметров функции используются для того, чтобы _____

_____

_____

_____

12. Получение значений необязательных параметров функции с переменным числом параметров становится возможным при помощи макроопределений _____

_____

_____, находящихся в заголовочном файле

_____

_____

_____

13. Объявление передаваемого в функцию параметра как константного:

а) Помогает программисту правильно понимать назначение параметров при вызове функции [ДА] [НЕТ]

б) Позволяет обнаруживать некоторые программистские ошибки еще на этапе компиляции [ДА] [НЕТ]

в) Влияет на надежность и скорость выполнения генерируемого компилятором исполнимого кода, вводя в него автоматические проверки [ДА] [НЕТ]

г) Иногда затрудняет написание тела функции, так как соответствующие параметры нельзя использовать внутри функции как вспомогательные переменные для получения результата [ДА] [НЕТ]

д) При возможности это сделать выступает признаком хорошего стиля при программировании [ДА] [НЕТ]

К главе 8

1. Программа может содержать в себе от _____ до _____ функций `main`.

2. Рекурсивной называется функция, которая _____

_____

_____

_____

3. Верно ли высказывание: «Функция `main` может быть рекурсивной»? [ДА] [НЕТ]

4. Необязательный параметр `argc` функции `main` указывает, _____

_____

_____

_____

5. Тип необязательных параметров `argv` и `argp` функции `main` таков: _____

_____

_____

_____

6. Может ли в нескольких различных файлах одной программы содержаться полностью идентичная декларация функции с применением спецификатора `extern`? [ДА] [НЕТ]
7. Могут ли в нескольких различных файлах одной программы содержаться различающиеся, но одноименные определения функции с применением спецификатора `static`? [ДА] [НЕТ]
8. Спецификатор `inline`:
а) Применим к произвольной функции [ДА] [НЕТ]
б) Ускоряет работу всякой функции путем отказа от обмена данными через стек [ДА] [НЕТ]
в) Всегда может быть проигнорирован компилятором [ДА] [НЕТ]
9. Указатель на функцию:
а) Указывает на место в памяти, где размещается начало исполнимого кода функции [ДА] [НЕТ]
б) Жестко привязан к прототипу функции [ДА] [НЕТ]
в) В предыдущих версиях языка эквивалентен имени функции [ДА] [НЕТ]
г) С точки зрения хранения в памяти подобен всякому иному указателю [ДА] [НЕТ]
10. Среди прочих действий `typedef`:
а) Дает новому типу имя [ДА] [НЕТ]
б) Может одному и тому же типу дать два различных имени и более [ДА] [НЕТ]
в) Позволяет лаконично описывать сложные виды данных [ДА] [НЕТ]
г) Осуществляет контроль типов, на основе которых вводится новый тип [ДА] [НЕТ]
11. Справедливо ли утверждение: «Использование указателя на функцию в массиве или структуре приводит к автоматическому размещению тела функции в массиве или структуре кода»? [ДА] [НЕТ]
12. Верно ли высказывание: «Указатель на функцию дает возможность произвольно изменять типы всех ее параметров путем явного преобразования типа указателя перед использованием»? [ДА] [НЕТ]

К главе 9

1. Утверждение: «Всякая программа бессмысленна, если не использует средства ввода и вывода стандартной библиотеки» [ИСТИННО] [ЛОЖНО]
2. Поток ввода-вывода – это _____
_____
_____
_____
3. Буферизация потока ввода-вывода заключается в _____
_____
_____
_____
и служит для _____
_____
_____
_____

4. Справедливо ли высказывание: «Функции работы с динамической памятью применимы также к областям автоматической памяти, но для работы с ней есть более удобные средства»? [ДА] [НЕТ]
5. С точки зрения стандартной библиотеки файл – это _____
6. Верно ли, что с одним и тем же файлом можно последовательно работать в текстовом и в двоичном представлении? [ДА] [НЕТ]
7. Справедливо ли утверждение: «Средства форматного ввода-вывода не могут работать с файлами, открытыми в двоичном режиме, а только в текстовом»? [ДА] [НЕТ]
8. В Си имеется поддержка следующих стандартных потоков ввода-вывода: _____
9. Соответствует ли действительности высказывание «Функции работы с текстовыми строками инвариантны к способу выделения под них места в памяти»? [ДА] [НЕТ]
10. Различаются ли с точки зрения стандартной библиотеки средства работы с файлами на жестком диске или в компьютерной сети? [ДА] [НЕТ]
11. Перечислите известные вам функции с плавающей точкой из стандартной библиотеки Си: _____

К главе 10

1. Все действия препроцессора сводятся к манипуляциям с _____
2. Можно ли при помощи директивы `#include` в одном и том же программном проекте одновременно включить файл `a.c` в файл `b.c` и файл `b.c` в `a.c`? [ДА] [НЕТ]
3. Повторное определение (переопределение) макро посредством директивы `#define` возможно при условии, что _____
4. Уязвимые макроопределения – это _____

5. Справедливо ли утверждение: «Всегда в любом месте программы можно осуществить замену: `#if` → `if (...)`, `#else` → `else`, соответствующим образом с помощью фигурных скобок превратив области действия директив в программные блоки»? [ДА] [НЕТ]

6. Выяснить факт введения именованного макроса до данного места программы можно при помощи _____

7. Осуществлять дополнительную диагностику программы можно при помощи следующих средств препроцессора: _____

8. Для заведомо корректной работы параметры макроопределений требуется _____

9. Две лексемы можно объединить в одну при помощи директивы препроцессора _____

10. Отметьте, какие высказывания верны:

а) `typedef` дает больше возможностей для введения новых типов в программу, чем `#define` [ДА] [НЕТ]

б) `#define` дает больше возможностей для введения новых типов в программу по сравнению с `typedef` [ДА] [НЕТ]

в) Возможности `typedef` и `#define` эквивалентны, и выбор среди них – исключительно вопрос вкуса программиста [ДА] [НЕТ]

г) `typedef` и `#define` предоставляют различающиеся возможности введения новых типов в программу, и выбор средства зависит от конкретной ситуации [ДА] [НЕТ]

Библиографический список

1. *Бочков С.О., Субботин Д.М.* Язык программирования Си для персонального компьютера. М. : Радио и связь, 1990. 384 с.
2. *Керниган Б.У., Ритчи Д.М.* Язык программирования С ; пер. с англ. 2-е изд. М. : Вильямс, 2009. 304 с.
3. *Котлинская Г.П., Галиновский О.И.* Программирование на языке Си. Минск : Вышэйшая школа, 1991. 156 с.
4. *Кочан С.* Программирование на языке Си ; пер. с англ. 3-е изд. М. : Вильямс, 2006. 496 с.
5. *Подбельский В.В., Фомин С.С.* Программирование на языке Си : учеб. пособие. 2-е доп. изд. М. : Финансы и статистика, 2004. 600 с.
6. *Прата С.* Язык программирования С : лекции и упражнения ; пер. с англ. М. : Вильямс, 2006. 960 с.
7. *Уинер Р.* Язык Турбо Си ; пер. с англ. М. : Мир, 1991. 304 с.
8. *Шилдт Г.С.* Полное руководство. Классическое издание ; пер. с англ. М. : Вильямс, 2011. 704 с.
9. Cygwin project. – Official Website: http://cygwin.com/
10. GCC. The GNU Compiler Collection. – Official Website: http://gcc.gnu.org/
11. ISO/IEC JTC1/SC22/WG14, The international standardization working group for the programming language C. – Official Website: http://www.open-std.org/jtc1/sc22/wg14/
12. *King K.N.* C Programming: A Modern Approach. 2nd ed. W.W. Norton & Company, 2008. 830 p.

Оглавление