1 Алгоритмы и средства их записи.

Одним из фундаментальных понятий математики и программирования как одной из ее ветвей является понятие алгоритма. Грубо говоря, алгоритм — это полностью определенная инструкция для решения какой-либо задачи. Один из самых ярких примеров достаточно простых алгоритмов — это алгоритм, по записи элементарной функции как выражения выдающий запись ее производной.

Алгоритмы могут быть представлены на материальных носителях по-разному. В сущности, любой язык программирования и существует как раз для записи алгоритмов. Однако, текст на некотором языке программирования — не единственный способ представления алгоритмов.

В данном разделе речь пойдет о записи алгоритмов в виде определенных рисунков. Эти рисунки называются блок-схемами алгоритмов. Они состоят из определенных узлов, описывающих элементарные действия или акты выбора, и соединительных линий между ними, указывающих на последовательность прохождения узлов, иногда в зависимости от выбора, сделанного в соответствующем узле.

Соединительные линии на одном из концов имеют стрелку, показывающую их направление. Поэтому для узла на одном из концов линии она будет исходящей, на другом — входящей.

Узлы, описывающие действия, как правило, выглядят, как прямоугольники, в которых написан текст, указывающий на действие, которому соответствует этот узел. Он может иметь несколько входных линий, но только одну выходную, показывающую, какой узел должен быть рассмотрен следующим.

Узлы, в которых делается выбор, имеют обычно вид ромбов, в которых стоит условие. Входных дуг здесь тоже может быть несколько, но выходных будет уже две — одна из них выбирается, если условие истинно, и другая — если оно ложно.

Кроме того, обычно имеется еще несколько узлов в виде прямоугольников, у которых к боковым сторонам присоединены полукруги. Один из этих узлов надписан словом «начало», все остальные — «конец». Начальный узел не имеет входящих дуг, и только одну исходящую, показывающую, в каком узле начинается выполнение алгоритма, представленного данной диаграммой. Конечные узлы имеют сколько угодно входящих дуг, но ни одной исходящей. Если рассмотрение узлов дошло до такого узла, на этом выполнение алгоритма прекращается.

Наконец, для описания алгоритмов, в том числе и в виде блок-схем, используются так называемые переменные. Переменная представляет собой слово (в обобщенном смысле, поскольку обычно в этом слове кроме букв разрешается иметь и цифры, и даже знак подчеркивания), обозначающее нечто, называющееся значением этой переменной, в качестве чего может выступать почти все, что угодно (числа, целые или вещественные, строки, имена других переменных, тогда переменная называется указателем, другие объекты, участвующие в работе алгоритма, для тех, кто знает — массивы, структуры и т. д.). Переменная может участвовать в действиях, имеющихся в составе алгоритма, двояко — либо нас интересует ее значение, участвующее в выражениях, либо мы хотим это значение изменить. Значение переменной может меняться в процессе работы алгоритма, и действие, меняющее значение переменной, называется присваиванием этой переменной нового значения. Значением переменной в определенный момент времени считается последнее присвоенное ей до этого момента значение. Извлечение значения переменной, которой еще ничего не было присвоено, считается ошибкой и запрещено, т. е. алгоритм, в котором такое возможно, считается неправильным.

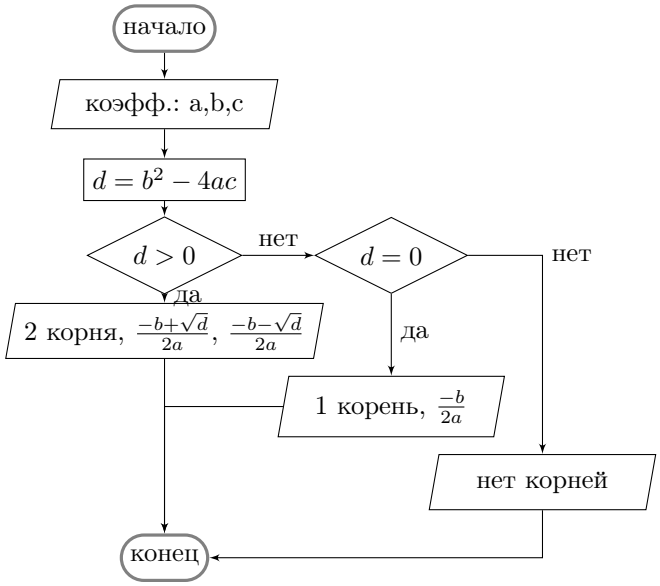
Чтобы алгоритм, записанный в виде блок-схемы, мог получать и выдавать информацию, в нем обычно имеется еще несколько узлов в виде параллелограммов, в которых написаны тексты, которые должны быть напечатаны при рассмотрении таких узлов, а также набор выражений — тех, значения которых должны быть напечатаны (как еще говорят, выведены), и тех переменных, значения которых должны быть считаны (как еще говорят, введены). В остальном узлы-параллелограммы ведут себя так же, как и прямоугольники.

Процесс выполнения так описанного алгоритма состоит в последовательном рассмотрении узлов, начиная с того, на котором написано «начало». Далее рассмотрение переходит на узел, куда ведет исходящая из него линия. Если нужно рассмотреть прямоугольный узел, выполняются указанные в нем действия, и рассмотрение перемещается на тот узел, куда ведет исходящая из только что рассмотренного узла линия. Если нужно рассмотреть ромбовидный узел, выясняется истинность его условия, и в зависимости от этого выбирается соответствующая исходящая линия, и рассмотрение переходит на тот узел, на котором она кончается. При рассмотрении параллелограмма печатается текст, указанный в этом узле, а также значения выводимых выражений; если в этом узле есть вводимые переменные, выполнение алгоритма приостанавливается, и система ждет, пока не будут введены значения считываемых переменных. После этого введенные значения присваиваются соответствующим переменным, и выполнение алгоритма продолжается. Наконец, если рассмотрение дошло до узла с надписью «конец», то процесс прекращается.

Иногда алгоритм представляется не одной такой диаграммой, а несколькими. Например, в главной диаграмме встречаются узлы с действиями, описанными отдельными диаграммами.

Чтобы рисовать подобные блок-схемы, можно воспользоваться программой MS Visio из состава MS Office. Достаточно простое руководство для начинающих доступно здесь. Также можно посмотреть сюда.

В качестве простого, но не совсем тривиального, примера рассмотрим решение квадратного уравнения. Мы будем обозначать его коэффициенты буквами a, b и c, причем предполагается, что $a \neq 0$. В конце мы должны получить число решений (нет, 1 или 2), и сами решения (если они есть).



Этот пример показывает, как можно записывать алгоритмы с простым ветвлением.

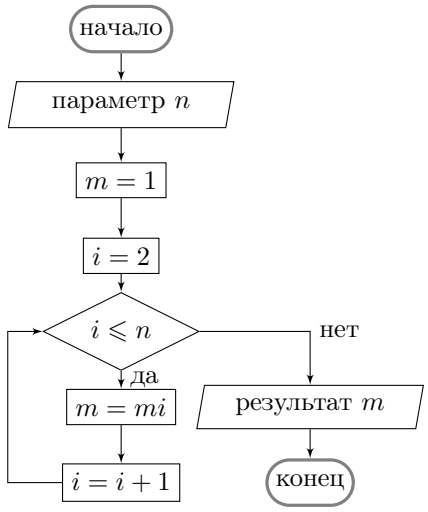
Задачи.

1. Усовершенствовать вышеприведенный алгоритм так, чтобы он мог выдавать правильные результаты для любого уравнения степени не выше двух, т. е. и в тех случаях, когда старшие коэффициенты равны нулю (в этом случае уравнение будет линейным или даже вовсе не зависящим от неизвестной — добавляются два подслучая бесконечного или пустого множества решений).

2*. Аналогичным образом решить систему двух линейных уравнений с двумя неизвестными, аккуратно рассмотрев все возможные случаи (нет решений; есть единственное решение (оно должно быть вычислено); есть бесконечно много решений (одна переменная может принимать любое значение, а другая через нее выражается — нужно узнать, как именно); обе переменные могут принимать любые значения).

Блок-схемы позволяют записывать и более сложные алгоритмы, в частности, алгоритмы с повторением некоторых действий, причем число повторений в этом алгоритме заранее неизвестно. В качестве примера приведем вычисление факториала целого числа.

Изначально вводится целое число, обозначенное буквой n. Наша цель состоит в том, чтобы получить и затем вывести его факториал, который мы будем обозначать буквой m. Кроме того, в процессе вычислений нам понадобится счетчик, который будет обозначаться буквой i. На момент окончания вычислений мы хотим, чтобы m обозначало $n!$. При этом пользоваться можно только арифметическими операциями и операциями сравнения. Тогда блок-схема алгоритма будет выглядеть примерно так:



Здесь запись «$m = mi$» означает следующее: обозначить буквой m новое значение, полученное как результат умножения старого значения на значение, обозначенное буквой i.

Задачи.

1. Придумать алгоритм, который для введенного вещественного значения получает число, сколько раз к нему надо применить синус, чтобы результат попал в интервал от $-\frac{1}{4}$ до $\frac{1}{4}$, и нарисовать соответствующую блок-схему (если значение x сразу попадает в указанный интервал, результат будет 0).

2. Придумать алгоритм, который проверяет, является ли входная последовательность символов записью целого числа, и если является, то в конце работы алгоритма нужно вывести значение введенного числа; если же нет, то нужно вывести текст «не число». Для решения этой задачи можно пользоваться только вводом отдельных символов, т. е. их кодов (каждый символ, который можно ввести с клавиатуры или отобразить на экране компьютера, имеет свой код, т. е. номер, как картинка, в таблице, которая называется кодировкой), в целые переменные; можно считать, что коды цифр от «0» до «9» равны, соответственно, целым числам от 48 до 57. Заканчивается ввод символом пробела (код 32), частью проверяемого набора символов он, естественно, не считается.

Записью целого числа является любая строка, содержащая (необязательный) знак − (код 45), и за ним непустую последовательность цифр. Если эта последовательность состоит из более, чем одной цифры, то первая из них не может быть нулем. Нарисовать блок-схему этого алгоритма.

3*. Придумать алгоритм, который проверяет, является ли входная последовательность символов записью вещественного числа с плавающей запятой, и нарисовать соответствующую блок-схему (коды символов «.», «+», «e» и «E» см. в интернете по запросу «таблица ASCII»).

2 Введение в C

Язык C появился в начале 70-х годов прошлого века (см. статью «Си (язык программирования)» в Википедии). Он поддерживает процедурный стиль программирования.

Основная цель языка C состоит в том, чтобы программы, написанные на нем, работали как можно быстрее, и за вполне предсказуемое время. Эти цели накладывают серьезные ограничения на использование в программах определенных понятий. Например, по этой причине в C не может быть автоматической сборки мусора, и, как следствие, нет поддержки такого понятия, как замыкание.

Любая программа на C состоит из нескольких текстовых файлов, содержащих ее запись. Под текстовыми файлами здесь понимаются те, которые можно открыть и отредактировать в блокноте (notepad). Файлы более сложной структуры, содержащие дополнительную информацию о форматировании, например, файлы редактора Word, или файлы web-страниц (обычно с расширением .html или .htm) для записи программы на C не подходят.

Файлы, содержащие запись программы, по смыслу делятся на две категории: так называемые заголовочные файлы и файлы с исходными текстами. Первые обычно имеют расширение «.h». Файлы с исходными текстами обычно имеют расширение «.c». Такая организация программы обслуживает идею, называющуюся раздельной компиляцией. Она будет обсуждаться в этом тексте чуть дальше.

Язык C чувствителен к регистру букв, т. е. различает большие и маленькие буквы.

В Internet имеется достаточно много руководств и справочников по C (например, `https://docs.microsoft.com/en-us/cpp/c-language/c-language-reference?view=vs-2017`, `https://docs.microsoft.com/en-us/cpp/preprocessor/c-cpp-preprocessor-reference?view=vs-2017`, `https://en.wikibooks.org/wiki/C_Programming/Standard_library_reference`, ...)

В качестве соглашения примем, что слова, написанные в примерах по-английски, обозначают ключевые слова и должны быть в программах написаны в точности так, как в примере; напротив, слова, написанные по-русски, обозначают соответствующие понятия, и в программах должны заменяться на конкретные тексты, описывающие данные понятия.

3 Раздельная компиляция

Как уже говорилось ранее, программа на C состоит из нескольких текстовых файлов двух разновидностей: заголовочных и исходных файлов. Теперь пришло время обсудить это подробнее.

Вообще говоря, программа на C — это набор подпрограмм (забегая несколько вперед, подпрограмма — это отдельный поименованный фрагмент всей программы, который может быть использован в любом месте программы при помощи указания его имени и некоторой дополнительной информации, которая будет обсуждаться позже). Практически любой процедурный или объектно-ориентированный язык имеет понятие подпрограммы, только называться это понятие в разных языках может по-разному. Например, в C и C++ подпрограммы называются функциями, в паскале — функциями или процедурами, в зависимости от того, вырабатывает ли эта подпрограмма значение или нет, в Java — методами, и т. д.

В языке C реализация разных подпрограмм может размещаться в разных исходных файлах (но каждая конкретная подпрограмма должна быть целиком реализована в одном из этих файлов).

Чтобы программу, написанную на языке C, запустить, ее нужно преобразовать в эквивалентную, т. е. делающую то же самое, программу на машинном языке. Этот процесс обычно называется сборкой. Программа на машинном

языке — это просто набор чисел, который можно записать в оперативную память компьютера и который затем можно выполнить как программу при помощи процессора этого компьютера. Дело в том, что процессор обычно понимает и может выполнить только программы, представляющие собой набор определенных чисел, в котором некоторые числа являются номерами команд процессора, а некоторые указывают на то, какую информацию процессор должен использовать, чтобы выполнить ту или иную команду.

Сборка программы на C состоит из нескольких этапов. Сначала каждый исходный файл обрабатывается препроцессором. Препроцессор — это небольшая программа, читающая файл с исходным текстом, и выдающая результирующий файл, также текстовый, как правило, гораздо длиннее исходного. Процесс преобразования исходного файла препроцессором управляется при помощи так называемых директив препроцессора, включаемых непосредственно в исходный файл программы. Препроцессор позволяет включать в программу другие файлы, определять и использовать макроопределения (некоторые правила, в соответствии с которыми одни фрагменты программы нужно заменять на другие), включать в программу или пропускать определенные ее фрагменты (так называемая условная компиляция). Кроме того, имеется специальная конструкция, выглядящая как директива препроцессора, но на самом деле управляющая не препроцессором, а компилятором.

После обработки каждого исходного файла программы препроцессором, полученный файл должен быть скомпилирован, т. е. содержащиеся в нем фрагменты программы на C преобразуются в эквивалентные, но записанные на языке ассемблера. Язык ассемблера — это язык, на котором записаны текстовые представления команд процессора. Этот язык, вообще говоря, зависит от компьютера, на котором мы собираемся запускать нашу программу, и от операционной системы.

После компиляции файл с текстом программы на языке ассемблера подается на вход ассемблеру, который превращает его в файл, содержащий представления команд процессора в виде чисел, т. е. в тот вид, который может быть выполнен непосредственно процессором. Результат работы ассемблера называется объектным модулем. В операционной системе Windows объектные модули имеют расширения .obj .

Несмотря на то, что в объектном модуле команды процессора записаны в том виде, в котором они могут быть выполнены процессором, запустить объектный модуль на выполнение нельзя. Дело в том, что программа может состоять из нескольких исходных файлов, из которых получается несколько объектных модулей, каждый из которых содержит определенную часть нашей программы.

Для запуска программы нужно еще собрать вместе все объектные модули, из которых она состоит. Программа, которая это делает, называется компоновщиком. Результатом работы компоновщика является готовый к запуску файл программы (в Windows такие файлы имеют расширение .exe).

При написании программы на C можно также использовать средства стандартной библиотеки C, а также средства из других библиотек. Библиотека — это архив из некоторых объектных модулей, и если в программе используются средства из какой-то библиотеки, ее объектные модули также должны участвовать в процессе компоновки.

Такая технология программирования (использование отдельно препроцессора, компилятора, ассемблера и компоновщика) называется раздельной компиляцией. Раздельная компиляция широко применяется в программировании. Практически для упрощения процесса получения работающей программы из исходных текстов в большинстве систем программирования используются дополнительные инструменты, как применяемые из командной строки, так и с графическим интерфейсом.

У раздельной компиляции есть ряд преимуществ по сравнению с теми способами написания программ, в которых весь текст программы помещается в один огромный файл:

1) Такая организация программы поддерживает концепцию модульности, когда каждый из исходных файлов отвечает за один небольшой, законченный по смыслу фрагмент программы, что сильно упрощает и ускоряет разработку программ, а также резко упрощает командную разработку, когда над одной программой трудится (большой) коллектив разработчиков.

2) Если изменяется один исходный файл, перекомпилировать нужно только его. Правда, перекомпоновывать нужно все, но компоновка обычно выполняется быстрее компиляции.

3) Продавцы библиотек могут распространять свои библиотеки в скомпилированном виде, что затрудняет извлечение из них применяемых там алгоритмов и структур данных.

4) Наконец, иногда такой подход позволяет писать разные фрагменты программы на разных языках программирования. Здесь, правда, нужно соблюдать определенную осторожность. В качестве примера назовем два возможных препятствия: а) разные способы вызова функций, например, соглашения о том, в каком порядке параметры функции хранятся в памяти, и б) так называемое декорирование имен, т. е. преобразование имен функций на стадии компиляции, так что компоновщик видит не те имена функций, которые указаны в тексте программы, а результаты их преобразования компилятором.

Для языка C существует много разных библиотек, в том числе и с открытым исходным кодом.

Возвращаясь к идее о том, что программа на C может состоять из нескольких исходных файлов, скажем кратко о причинах появления такой возможности.

Дело в том, что в C, как и во многих других императивных или объектно-ориентированных языках, описание подпрограммы состоит из двух частей: заголовка и тела. Заголовок описывает тип возвращаемого подпрограммой

значения, если такое есть, имя подпрограммы и ее параметры (дополнительная информация, необходимая подпрограмме для работы). Что же касается тела, оно представляет собой описание собственно того фрагмента программы, которому назначается имя, указанное в заголовке. Для использования такого фрагмента программы в другом ее месте используется конструкция, называемая вызовом — там указывается имя подпрограммы и значения ее параметров.

Так вот, для правильной компиляции вызова подпрограммы ее тело, вообще говоря, не нужно, а нужен только заголовок (разумеется, все используемые в программе функции должны быть где-то реализованы, но эта реализация важна на этапе компоновки, а не компиляции, так что можно определять функцию в одном файле, а использовать в других, лишь бы там был заголовок такой функции). Поэтому в конкретном исходном файле вместо полных описаний используемых там подпрограмм достаточно указать их заголовки. Поэтому описания подпрограмм бывают двух видов: объявления (указан только заголовок; такие описания называются также прототипами подпрограмм), и определения (указано полное описание подпрограммы, т. е. и заголовок, и тело). Для того, чтобы правильно скомпилировать некоторый исходный файл, в нем должны быть указаны объявления всех используемых в нем подпрограмм. Если их много, то гораздо проще собрать их все по смыслу в несколько разных файлов и затем включить эти файлы в текст соответствующего исходного файла при помощи директивы препроцессора #include. Такие файлы (содержащие наборы заголовков подпрограмм) и называются заголовочными. Файлы же, в которых присутствуют определения некоторых подпрограмм, называются исходными или файлами реализации.

Помещать определения подпрограмм, кроме редких исключений, в заголовочные файлы не нужно, потому что в этом случае содержащиеся в них определения функций попадут, скорее всего, в несколько исходных файлов, и компоновщик не будет в состоянии решить, какие из этих определений использовать для работы программы.

В обычных ситуациях, компилировать нужно только исходные файлы, получая из них объектные модули и компонуя их затем с нужными библиотеками для получения готовой программы. Однако, встречаются расширения языка, в которых заголовочные файлы также должны обрабатываться определенным образом, в результате чего могут появиться дополнительные исходные файлы, которые тоже должны быть скомпилированы и их объектные модули прикомпонованы к окончательной программе.

4 Программы для разработки на C

Как очевидно можно сделать вывод из предыдущих разделов, для программирования на C требуется ряд инструментов. Их можно разделить на обязательные и дополнительные.

Обязательными инструментами являются текстовый редактор, препроцессор, компилятор, ассемблер и компоновщик, равно как и стандартная библиотека (набор заголовочных файлов и соответствующих ему объектных модулей; как правило, объектные модули стандартной библиотеки объединяются в один или несколько файлов, которые называются библиотеками). Важным для первых трех из этого набора инструментов является вопрос о том, какую версию стандарта языка C они поддерживают. Для оставшихся важным является вопрос о том, какие они поддерживают форматы объектных модулей, исполняемых файлов и библиотек.

В мире на данный момент существует достаточно много разных компиляторов языка C. Самым известным из них является, наверно, GCC (Gnu Compiler Collection) благодаря тому, что он бесплатен и открыт, а также является основой очень многих, если не всех, дистрибутивов Linux.

Также широко известным является компилятор, входящий в состав Visual Studio. Вообще говоря, он платный, но, как и у других, у Visual Studio есть версия Community Edition, распространяемая бесплатно. Правда, этот компилятор имеется только под Windows.

Следующим, наверно, следует упомянуть clang. Он также бесплатен и открыт, как GCC, и обычно используется в маках, но может быть установлен и в Linux.

Наконец, упомянем еще два компилятора. Первый из них — это продукт корпорации Intel. Он платный, но для некоммерческой разработки доступен бесплатно; правда, под некоммерческой разработкой понимается деятельность, за которую ее участники не получают ни гроша ни в каком виде. В частности, ни преподавание, ни академические научные разработки под эту статью не подпадают.

Последний компилятор, который будет здесь упомянут — компилятор фирмы Oracle, поставляемый вместе со средой разработки Oracle Developer Studio. Он бесплатный, но не открытый.

Кроме этих пяти, имеются и многие другие.

Очень часто производители компиляторов предлагают компиляторы C и C++ в комплекте, а также в составе среды разработки, хотя они и запускаются разными командами операционной системы.

К дополнительным средствам, в частности, можно отнести:

- Обобщенные команды. Такая команда принимает имена исходных файлов, а также библиотек, и по очереди сама вызывает препроцессор, компилятор, ассемблер и компоновщик; в конце своей работы она формирует готовую программу для запуска.
- Системы для автоматической сборки больших проектов. К этому классу средств однозначно относится программа make и ее надстройки типа cmake или autoconf.

- Отладчики, т. е. программы, позволяющие исполнять другую, отлаживаемую, программу пошагово, отслеживая по ходу дела значения переменных.

- Профилировщики, т. е. программы, позволяющие автоматически исследовать работу другой программы, собирая при этом некоторые сведения, полезные для определения ее корректности и эффективности. Например, профилировщик может определить, какую долю времени программа тратит на выполнение каждой из своих частей, что полезно для выявления узких мест программы, нуждающихся в оптимизации в первую очередь. Некоторые профилировщики также могут обнаруживать неправильные обращения к памяти и ряд проблем параллельного программирования.

- Среды разработки, т. е. комплексы из нескольких разных средств, перечисленных выше. Как правло, в их состав входят текстовый редактор с возможностью обнаруживать некоторые ошибки по мере ввода программы, средства обслуживания проектов для оптимизации процесса сборки, далее — интерфейс к компилятору, отладчику, профилировщику.

Из бесплатных сред разработки нужно, наверно, в первую очередь упомянуть eclipse. Благодаря модульной архитектуре и баснословному числу всяких плагинов она может служить средой разработки на практически любом современном языке, не только программирования, но и разметки (в качестве примеров последнего назовем TEXlipse — надстройка для написания статей и книг в системе TEX).

Далее упомянем NetBeans и созданную на ее основе OracleDeveloperStudio — компилятор из ее состава уже упоминался выше.

Упомянем, конечно, и Visual Studio. Из ее недр вышел редактор Visual Studio Code, бесплатно доступный на Windows, Linux и MacOS.

Последняя среда, которую мы здесь упомянем — CLion от JetBrains. Она коммерческая, но для студентов и преподавателей доступна бесплатно.

Кроме перечисленных выше сред разработки, существует множество других, правда, они меньше перечисленных по объему и, соответственно, по возможностям.

Мы будем изучать программирование на языке C в среде Visual Studio.

Для начала запустим Visual Studio. Обычно при первом запуске она спрашивает, на каком языке мы будем программировать. Нужно выбрать Visual C++ (для программирования на C, а не на C++, просто расширение исходного файла должно быть .c, а не .cpp). Затем Visual Studio будет производить первоначальную настройку; здесь придется подождать (как правило, несколько минут — не удивляйтесь).

Далее, для написания программы на C нужно создать проект. Для этого достаточно щелкнуть мышью на соответствующей ссылке на стартовой странице («Создать проект...»). В появившемся окне нужно выбрать тип проекта («Консольное приложение Win32»), а также задать его имя и расположение (в какой хотите папке, но на Вашем сетевом диске W:), после чего нажать «OK».

Наконец, нужно определить свойства проекта. Для этого в появившемся окне нужно нажать «Далее» и в следующем окне сбросить флаги «Предварительно скомпилированный заголовок» и «Проверки жизненного цикла разработки безопасного ПО (SDL)», установить флаг «Пустой проект» и нажать «Готово».

Если все было сделано правильно, создастся пустой (не содержащий файлов) проект и откроется в обозревателе решений («Обозреватель решений», при обычных настройках — правая верхняя часть окна). Для добавления в проект файлов, содержащих текст программы, нужно щелкнуть правой кнопкой мыши по заголовку проекта (имя проекта жирным шрифтом) и в появившемся контекстном меню выбрать пункт «Добавить», затем «Создать элемент... (Ctrl+Shift+A)» и в появившемся окне задать тип добавляемого файла («Файл C++ (.cpp)», файл с программой на C++) и его имя, с явно указанным расширением .c (если в указанном имени явно присутствует символ точка, то автоматически расширение не добавляется). После этого нужно щелкнуть на кнопке «Добавить», и создается указанный файл, добавляется в проект и открывается в редакторе. Теперь можно набирать программу.

Для запуска программы нужно воспользоваться меню «ОТЛАДКА», пункт «Запуск без отладки (Ctrl+F5)» и в появившемся окне нажать «Да». Программа компилируется, компонуется, и, если не было ошибок, запускается, для чего открывается новое текстовое окно терминала (как для ввода команд в командной строке). После завершения работы программы выдается сообщение «Press any key to continue . . . », и после нажатия любой клавиши окно терминала закрывается.

Важное замечание состоит в том, что если программа запущена, то перекомпилировать ее невозможно — нужно сначала дождаться завершения работы программы или принудительно закрыть ее, и только потом компилировать новую версию.

Также, для каждой программы нужно создавать свой проект. Для этого после завершения работы над программой нужно закрыть проект, выбрав пункт «Закрыть решение» меню «ФАЙЛ», и затем создать для следующей программы новый проект, щелкнув на кнопке из панели инструментов с подсказкой «Создать проект (Ctrl+Shift+N)».

5 Препроцессор

1. Общие сведения.

Препроцессор, как мы уже знаем, обрабатывает исходный файл программы на C до того, как этот файл поступит на вход компилятора.

Препроцессор позволяет решать следующие задачи: включать в текст программы содержимое других файлов, определять и использовать макроопределения, включать или исключать определенные фрагменты программы из файла в зависимости от истинности некоторых условий (так называемая условная компиляция) и некоторые другие.

Для управления работой препроцессора используются так называемые директивы препроцессора. Каждая директива препроцессора обязательно должна занимать одну отдельную строку и начинаться с символа #, за которым идет имя директивы. Кроме имени, директива может содержать и другую информацию, которая отделяется от имени пробелами. Если текст директивы с параметрами не помещается на одной строке, можно воспользоваться механизмом продолжения строки, поставив в самом конце каждой строки (непосредственно перед символами, разделяющими строки) символ \ .

2. Включение файлов.

Включение других файлов в программу в основном используется для нужд раздельной компиляции. Для этого используется директива #include. Она имеет один параметр — имя включаемого в программу файла, возможно, с указанием его пути. Этот параметр должен быть заключен либо в угловые скобки, и тогда препроцессор будет искать соответствующий файл в специальных системных папках, либо в двойные кавычки. Такой вариант означает поиск файла в папках, отведенных для пользовательских заголовочных файлов (обычно входящих в проект).

3. Макроопределения.

Следующая группа директив препроцессора касается макроопределений. Для создания макроопределнний используется директива #define, для уничтожения — #undef. Макроопределения бывают с параметрами или без них.

Макроопределение без параметров — это просто идентификатор, каждое вхождение которого в текст программы (не как часть другого иденификатора или строковой константы) заменяется на определенную строку. В языке C достаточно долго это был единственный способ описывать константы. Например, следующая строка определяет константу pi:

```
#define pi 3.1415926
```

В этом случае в выражении pi+1 слово pi будет заменено на 3.1415926. А вот в выражениях pie-3 или printf("pi=") такая замена уже не будет выполнена.

Макроопределения бывают также с параметрами. При определении их после имени макропределения в скобках стоят через запятую имена параметров, которые также могут встречаться и в той строке, на которую макроопределнние надо заменять (в дальнейшем мы будем называть эту строку текстом замены). При использовании макроопределения с параметрами после его имени в скобках через запятую указываются значения параметров. Вся эта конструкция заменяется на текст замены, в котором имена параметров макроопределения заменены на их значения. Например, если определить

```
#define sqr(X) X*X
```

то в выражении sqr(a)+1 первое слагаемое будет заменено на a*a. Однако, в выражении sqr(x+1)+3 первое слагаемое будет заменено на x+1*x+1, что, скорее всего, неправильно, но будет скомпилировано без ошибок и даже предупреждений, приводя к тому, что программа молча выдаст неверный ответ. Так что лучше писать такие макроопределения в следующем виде:

```
#define sqr(X) ((X)*(X))
```

Параметры макроопределения могут быть не только идентификаторами или, в большей общности, выражениями, но и фрагментами выражений, например, так:

```
#define dbl(X,O) ((X)O(X))
```

что может быть использовано не только как dbl(x,*) (заменяется на ((x)*(x))), но и как dbl(x,+) (заменяется на ((x)+(x))). С одной стороны, это достаточно удобно, так как позволяет иногда сильно сократить текст программы, объединяя в вызовы одного макроопределения (почти) совпадающие фрагменты программы. Однако, такой подход может быть опасным, поскольку при использовании макроопределений практически отсутствуют какие-либо проверки на правильность параметров макроопределений. Например, мы можем использовать наше макроопределение

и так: dbl(x,1), что будет заменено на ((x)1(x)), и бессмысленность результата такой замены будет обнаружена не препроцессором, а компилятором, уже после завершения работы препроцессора. Такие ошибки трудно обнаруживать программисту, поскольку он в своем файле видит только вызов (использование макроопределения, в нашем случае dbl(x,1)), тогда как компилятор видит результат замены этого вызова, т. е. ((x)1(x)). Проблема в том, что препроцессору все равно, что на что заменять — он не вникает в смысл тех строк, которые подставляет вместо макроопределений.

Задача. Написать макроопределение, вычисляющее функцию $f(x) = x^3 - x^2 + 5x - 3$.

Имеются также две удобные операции препроцессора, обычно используемые для параметров макроопределений. Это операция «закавычивания», представленная значком # и операция «сцепления лексем» ##. Первая используется тогда, когда нужно превратить значение параметра макроопределения в строковую константу, т. е. представление текста в программе на C++.

Например, мы хотим написать макроопределение OUT(X), заменяющееся на printf("содержимоеX"). Вариант

```
#define OUT(X) printf("X")
```

не годится, поскольку внутри кавычек никакие замены не производятся, в частности, и X не будет заменяться на значение параметра макроопределения, т. е., например, текст OUT(+++) будет заменяться на printf("X"). Вариант

```
#define OUT(X) printf(X)
```

тоже не годится, поскольку в этом случае содержимое X будет поставлено без кавычек, т. е., например, текст OUT(+++) будет заменяться на printf(+++). Эта проблема решается при помощи операции «закавычивания» # так:

```
#define OUT(X) printf(#X)
```

и тогда текст OUT(+++) будет заменяться на printf("+++").

Задача. Написать макроопределение W с двумя параметрами X и Y, заменяющееся на

```
содержимоеX = "содержимоеY";
```

Операция сцепления лексем используется тогда, когда мы хотим приписать к значению параметра макроопределения вплотную (без пробелов) число, слово или значение другого параметра. Например, мы хотим написать макроопределение conc, принимающее в качестве параметров два слова и заменяющееся на результат их сцепления. Например, conc(one,two) должно заменяться на onetwo. Вариант

```
#define conc(X,Y) XY
```

не годится, поскольку здесь и X, и Y в тексте замены являются частями большего слова и потому не заменяются на значения параметров макроопределения, т. е., например, текст conc(one,two) будет заменяться на XY. Вариант

```
#define conc(X,Y) X Y
```

тоже не годится, поскольку в этом случае между значениями параметров будет стоять пробел, т. е., например, текст conc(one,two) будет заменяться на one two. Эта задача решается при помощи операции «сцепления лексем» ## так:

```
#define conc(X,Y) X##Y
```

и тогда текст conc(one,two) будет заменяться на onetwo.

Задача. Написать макроопределение W с двумя параметрами X и Y, заменяющееся на

```
содержимоеXсодержимоеYсодержимоеX
```

Для удаления макроопределения используется директива #undef, единственным параметром которой является имя удаляемого макроопределения. После этой директивы до следующего определения того же макроопределения (если оно вообще последует) это имя перестает распознаваться как имя макроопределения, и следовательно, заменяться на что бы то ни было, а попадает на выход препроцессора как есть.

4. Условная компиляция.

Следующая группа директив обслуживает так называемую условную компиляцию. Директивы этой группы позволяют проверять истинность некоторых условий и в зависимости от этого включать в программу или исключать из нее те или иные фрагменты. Такая возможность используется для автоматической настройки программы на компиляцию на разных платформах (операционная система+компилятор+процессор) или для получения из одного исходного файла нескольких вариантов программы, например, отладочной версии, проверяющей много дополнительных условий и осуществляющей отладочный вывод, и рабочей версии, в которой ничего этого нет.

Директивы условной компиляции в самом общем виде используются следующим образом:

```
#if условие1
текст1
#elif условие2
текст2
...
#elif условиеN
текстN
#else
текстN+1
#endif
```

Смысл этой конструкции состоит в следующем. Если условие1 истинно, то вместо всей этой конструкции в выход препроцессора попадает только текст1; если условие1 ложно, но условие2 истинно, то текст2, и т. д. Наконец, если все условия ложны, в выход попадает только текстN+1. В итоге мы получаем возможность выбирать тот вариант, который больше подходит для конкретной ситуации.

Теперь об условиях. В самом общем случае условие может содержать следующее: целые константы (явно выписанные целые числа), арифметические, битовые и логические операции, операции сравнения, макроопределения (которые заменяются на свои значения), специальные конструкции, из которых упомянем только конструкцию defined(имя), имеющую значение 1, если имя является именем некоторого макроопределения, и 0 иначе (при этом совершенно не важно значение этого макроопределения, т. е. на какую строку его надо заменять). Трактовка значений условий следующая: 0 — ложь, все остальное — истина.

Для сокращения текстов программ имеются две специальные разновидности директивы #if: #ifdef имя и #ifndef имя. Первая проверяет, что имя является именем некоторого макроопределения (синоним #if defined(имя)), а вторая — наоборот (синоним #if ! defined(имя)).

Вместе с директивами условной компиляции часто используется директива #error текст. Она выдает текст в качестве сообщения об ошибке и завершает компиляцию. Например, если у нас в программе определяются две константы M и N, определенные при помощи #define, и среди всех возможных пар их значений по логике программы осмысленны только такие, где M<N, то это условие может быть проверено на этапе компиляции при помощи следующего фрагмента:

```
#if M>=N
#error M must be < N
#endif
```

Задачи. 1. Написать фрагмент, проверяющий на этапе обработки программы препроцессором следующее условие: из отрезков длины P, Q, R можно составить треугольник, и если это не так, выводящий сообщение об ошибке и прекращающий обработку. Подсказка: операция логического «и» выглядит на C как «&&».

2*. Написать фрагмент, проверяющий следующее условие: значение N содержит ровно 20 единичных битов (N — константа, определенная при помощи #define, влезающая в 32-битовое целое число).

В качестве еще одного примера рассмотрим так называемый сторож (конструкция, которая приводит к тому, что защищенный с ее помощью файл может быть включен в любой другой файл не более одного раза).

```
#ifndef имя_макро
#define имя_макро

содержимое_файла

#endif
```

Здесь вместо имя_макро должно стоять уникальное имя, обычно связанное с именем того файла, который защищается сторожем. Если мы включаем такой файл в какой-то другой первый раз, такого макроопределения там еще нет и содержимое нашего файла действительно попадает в этот файл, но вместе с тем там появляется и это макроопределение, так что при повторных попытках включить защищаемый файл в тот же файл, директива условной компиляции не допустит этого, поскольку теперь уже упоминающееся в ней имя будет именем макроопределения и условие будет ложным.

Эта конструкция встречается настолько часто, что специально для нее было придумано сокращение, а именно в начале файла достаточно просто написать #pragma once

Задача. Написать конструкцию подобного рода (назовем ее суперсторож), которая препятствует включению защищаемого ею файла в другой файл более двух раз (один и два раза можно, а больше не включается). Увы, конструкции #pragma twice в C нет.

5. Определение положения в файле.

Препроцессор поддерживает обычно несколько заранее определенных макроопределений, которые описывают его состояние в данный момент. Например, имеются две так называемые псевдопеременные __FILE__ (содержит имя текущего файла) и __LINE__ (содержит номер текущей строки). Их содержимое используется для указания положения ошибки в случае ее обнаружения в программе; однако, их можно использовать и в других случаях. Самый очевидный пример — традиционное макроопределение assert, принимающее в качестве параметра условие и встраивающее в программу код, проверяющий истинность этого условия. Если условие истинно, программа продолжает выполнение, а если нет — выдается сообщение об ошибке, включающее имя исходного файла и номер строки в нем, на которой было вызвано это макроопределение, и программа завершается.

Присваивать этим переменным новые значения напрямую нельзя, но есть специальная директива препроцессора, которая позволяет задавать эти значения. Выглядит ее использование следующим образом:

```
#line номер "имя_файла"
```

Это бывает нужно нечасто и, в основном, тогда, когда имеется средство автоматического порождения программы из фрагментов, предоставляемых пользователем. В этом случае в автоматически порожденной программе используется такая директива, чтобы ошибки, обнаруженные в тех фрагментах порожденной программы, которые были предоставлены пользователем, выдавались с указанием на их места не в порожденной программе, а в файлах пользователя, из которых были взяты соответствующие фрагменты.

6. Управление процессом компиляции.

Теперь рассмотрим конструкцию, выглядящую как директива препроцессора, но на самом деле не имеющую к препроцессору почти никакого отношения. Эта конструкция имеет следующий синтаксис:

```
#pragma параметры
```

Она используется для указания опций компилятору, т. е. для управления его работой. Как правило, параметры начинаются с трехбуквенного кода, который говорит о том, какому компилятору адресованы указания или о чем они, и если тот компилятор, который компилирует эту программу, не понимает этого кода, то указания просто игнорируются. Хороший стиль написания программ состоит в том, чтобы программа работала и выдавала правильный результат независимо от того, понимает конкретный компилятор содержащиеся в ней указания или нет (т. е. указания должны влиять только на оптимизацию, но не на работоспособность программы). Чаще всего такого рода указания касаются распараллеливания программы, и если компилятор их не понимает, программа просто получается последовательной (т. е. без распараллеливания).

7. Препроцессор как отдельная команда.

В некоторых системах программирования на C препроцессор доступен как отдельная команда. Например, в операционной системе Linux эта команда называется cpp. Это позволяет применять его не только к программам на C, но и к другим текстовым файлам.

Что касается Windows, на Visual Studio препроцессор вмонтирован в компилятор, но есть опции, позволяющие запустить только препроцессор (для VS 2017 это опции /E и /EP — интересующихся подробностями отсылаем к соответствующей документации `https://docs.microsoft.com/en-us/cpp/preprocessor/preprocessor`).

6 Общая структура программы

В качестве примера возьмем простейшую программу на C, состоящую из всего одного текстового файла:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hello, world!\n");
    return EXIT_SUCCESS;
}
```

Разберем эту программу более подробно. Первые две строки — это, как мы уже знаем, директивы препроцессора. Данные директивы требуют включить в программу текст из файлов stdio.h и stdlib.h. Это означает, что перед компиляцией из нашего текстового файла будет построен новый файл, в котором сначала будет представлено содержимое файла stdio.h (вместо первой директивы #include), затем — файла stdlib.h (вместо второй директивы), и,

наконец, оставшиеся пять строк нашей программы. Именно этот новый файл и будет скомпилирован для получения объектного модуля, а затем и окончательной программы на машинном языке, которую можно запустить. Включаемые файлы содержат информацию, необходимую для правильной компиляции нашей программы. Какую именно — узнаем несколько позже. Далее, очевидно, что мы этих файлов не писали — они предоставляются нам разработчиками системы программирования на языке C — в нашем случае Visual Studio.

Наконец, оставшаяся часть программы — это определение функции main. В каждой программе должна быть ровно одна функция main, поскольку запуск программы как раз и состоит в выполнении функции main.

Ключевое слово int, стоящее перед именем функции, означает тип целых чисел (аналог integer в Pascal). Любая программа работает с информацией разного рода — числами, строками, более сложными объектами. Тип объекта показывает, каким образом информация, относящаяся к объекту, хранится в памяти компьютера, и какие операции над этими объектами можно выполнять. Например, целое число 1 и вещественное число 1.0 математически представляют собой одно и то же, но в памяти компьютера хранятся совершенно по-разному, потому что первое считается относящимся к целому типу (int), а второе — к вещественному (double). Они различаются компилятором при помощи следующего нехитрого правила: если в записи числа нет ни символа «.», ни E или e (указание порядка), число считается целым, а если есть (даже при том, что дробная часть может быть равна нулю) — вещественным.

Тип, стоящий перед именем функции, указывает тип результата данной функции. Некоторые функции могут вырабатывать в процессе своей деятельности определенный результат. Например, функция sin, вычисляющая синус, вырабатывает вещественное число, так что тип ее результата будет double. Функция main всегда должна вырабатывать целое число; в некоторых учебниках до сих пор иногда встречается конструкция void main, что указывает на отсутствие у функции main результата, но это противоречит стандартам как C, так и C++. По этим стандартам, поведение такой функции считается неопределенным, т. е. она вполне имеет право делать все, что угодно, в том числе отформатировать Вам жесткий диск.

Пустые скобки после имени функции указывают на отсутствие у данной функции параметров. Параметры — это некоторая дополнительная информация, необходимая для работы функции. Например, у все той же функции sin должен быть один вещественный параметр, содержащий значение того угла, синус которого она будет вычислять. В отличие от Pascal, даже если у функции нет параметров, эти скобки (пустые!) нельзя выкинуть. Правила о постановке символа «;» после заголовка функции в C сложнее, чем в Pascal. Этот символ ставится только в том случае, если мы опускаем тело функции (а такое в C вполне возможно; заголовок функции без ее тела называется прототипом функции или ее объявлением, в отличие от определения, когда тело функции указывается); если же тело функции присутствует (как в нашей программе), то точку с запятой после заголовка ставить нельзя.

Фигурные скобки в C++ ограничивают тело функции, т. е. собственно тот фрагмент программы, которому назначается имя при определении функции (тело функции, если оно присутствует, должно располагаться сразу за ее заголовком, и может отделяться от него разве лишь несколькими пробельными символами). Фигурные скобки в данном случае нужны даже тогда, когда тело функции состоит из всего одного оператора (оператор — единица действия в императивных языках программирования, т. е. таких, как Fortran, Pascal, C, C++, Ada, Java и т. п.).

В нашем случае тело функции main состоит из двух операторов. Первый из них — оператор, сделанный из выражения, смысл его состоит в вычислении этого выражения, а синтаксис — выражение и затем точка с запятой. В отличие от Pascal, точка с запятой не разделяет операторы, а завершает оператор, сделанный из выражения, и ряд других разновидностей операторов. В частности, в C допустим пустой оператор, состоящий только из точки с запятой (оператор, сделанный из пустого выражения, который ничего не делает; зачем это бывает нужно — обсудим чуть позже).

Данное выражение состоит из вызова функции вывода (printf), примененной к строковой константе "Hello, world!\n". В отличие от Pascal, в C различаются строковые и символьные (тип которых — один символ, char, как в Pascal) константы.

\n — так называемая escape-последовательность, означающая символ перевода строки, чтобы следующий за ее выводом текст начинался в следующей строке.

Вообще говоря, функция вывода printf возвращает целое число — код ошибки, показывающий, успешно она выполнилась или нет. Здесь результат этой функции просто проигнорирован, потому что в C можно игнорировать результат выражения.

Наконец, оператор return возвращает результат функции main. В отличие от Pascal, основная функция программы должна возвращать целое число в качестве результата. Это число используется в операционной системе для указания того, успешно ли завершилась программа или в процессе работы возникли ошибки. Такие указания важны в том случае, если Ваша программа вызывается автоматически из другой программы, которая должна знать, был ли этот запуск успешным.

Для этого в файле stdlib определены две константы — EXIT_SUCCESS и EXIT_FAILURE. Смысл этих констант ясен из их названия. Их использование переносимо, т. е. должно работать везде, где система программирования на C соответствует стандарту. Вообще говоря, программа может возвращать и другие значения, по которым можно судить, какие именно ошибки имели место, но это, как правило, непереносимо, т. е. привязывает программу к определенной операционной системе.

Здесь можно сделать еще одно замечание. В отличие от Pascal, языки C и C++ чувствительны к регистру букв, т. е. различают большие и маленькие буквы. Так что если написать return exit_success; — такое работать не будет.

Задачи.

0. Запустите Visual Studio, создайте проект, введите программу из начала этого файла и запустите ее.

1. Напишите и запустите программу, выводящую на экран строку «Here was I».

2. Напишите и запустите программу, выводящую на экран две строки «Line one» и «Line two» на разных строках, используя только один вызов printf.

Завершим этот раздел, сказав пару слов о таком явлении в языках программирования, как комментарии. Это некоторая дополнительная информация, предназначенная для людей, читающих программу. Компилятор, преобразующий программу в машинно-исполняемый вид, обычно их просто игнорирует. В C бывают комментарии двух видов: комментарии, начинающиеся с текста //, заканчиваются концом текущей строки. Если же комментарии начинаются с текста /*, то они заканчиваются текстом */. При этом комментарии одного и того же типа не могут быть вложенными один в другой (разных типов — могут).

Иногда в языках программирования встречаются и такие комментарии, которые влияют на компиляцию: например, в языке Turbo Pascal при помощи такого рода комментариев можно устанавливать различные режимы компиляции и другие опции компилятора. Например, при помощи определенного вида комментариев можно управлять тем, проверяется ли переполнение чисел при арифметических вычислениях. В стандартном C такого рода комментарии не приняты, однако различные расширения языка вполне могут их использовать.

Наконец, в последнее время широкое распространение получили так называемые документирующие комментарии. Их смысл в том, что существуют специальные программы, извлекающие такие комментарии из текстов программ и порождающие по ним документацию к программам автоматически. В качестве примера для C можно назвать программу doxygen.

7 Понятие типа в C

Всякая программа на C так или иначе манипулирует различными значениями. Как и для предметов в реальной жизни, среди значений встречается много различных, но похожих между собой. Про такие значения в программировании говорят, что они принадлежат одному типу.

Всякое значение, которым манипулирует программа, так или иначе хранится в памяти компьютера. Поскольку память компьютера — это просто набор ячеек, каждая из которых хранит небольшое целое число, то для манипуляции записанным в памяти значением нужно знать, каким способом это значение записано, т. е. представлено в виде набора небольших целых чисел. Именно такая информация о значении и называется его типом. Узнать, сколько ячеек занимает значение того или иного типа, можно при помощи конструкции sizeof(тип). Например, чтобы узнать, сколько ячеек занимают значения типа int, можно воспользоваться такой программой:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
using namespace std;
int main()
{
    setlocale(LC_ALL, "Russian");
    printf("Размер значения типа int равен %d байтов.\n", sizeof(int));
    return EXIT_SUCCESS;
}
```

В этой программе включается заголовочный файл locale.h, а также вызывается функция setlocale для того, чтобы в окне командной сроки правильно выводились русские буквы. Также, конструкция %d в строке у printf означает, что в этом месте нужно напечатать целое число, переданное как дополнительный параметр.

Задача. Узнать, сколько ячеек занимают значения типов _Bool, char, short, int, long int, long long int.

Например, пусть у нас имеется тип беззнаковых (неотрицательных) целых чисел, называющийся unsigned int. Значения этого типа на тех компьютерах, на которых мы работаем, занимают 4 ячейки, т. е. байта. При этом содержимое этих ячеек представляет собой цифры в записи целого числа в 256-ричной системе счисления. Отсюда следует, что в этом типе можно представить 2^{32} различных значений, расположенных в диапазоне от 0 до $2^{32}-1$ включительно.

Иногда требуется также представлять целые числа, которые могут быть отрицательными. Для этого существуют знаковые типы, например int. Значения этого типа на тех компьютерах, на которых мы работаем, занимают те же 4 ячейки, но способ трактовки содержащихся в них чисел несколько отличается от того, который был рассмотрен ранее. Числа в диапазоне от 0 до $2^{31}-1$ представляются ровно также, а вот остальные значения используются для представления отрицательных чисел $-x$ значением $2^{32}-x$ для x от 1 до 2^{31}. Такой способ представления отрицательных

чисел называется «дополнением до двух». Он удобен тем, что способ выполнения операций сложения и вычитания одинаков для знаковых и беззнаковых чисел.

Различные цели использования тех или иных значений могут определять необходимые для их представления диапазоны, и, следовательно, объем памяти, который они должны занимать. В C есть разные типы целых чисел, отличающиеся один от другого количеством занимаемых ячеек, объединенные в две системы:

1) Старая система состоит из типов char, short int, int, long int, long long int. О количестве ячеек, знимаемых значениями этих типов, можно утверждать только, что оно не убывает слева направо в этом списке. В частности, на тех компьютерах, на которых мы работаем, тип int ничем не отличается от long int.

2) Новая система позволяет явно указать число битов (двоичных цифр), занимаемых значениями указанного так типа: int8_t, int16_t, int32_t, int64_t. Чтобы использовать типы из новой системы, необходимо в начале программы включить файл stdint.h.

Кроме размера, можно указать, нужно ли представлять отрицательные числа. Если нет, то перед именем типа можно поставить ключевое слово unsigned, превращая соответствующий тип в беззнаковый. Как правило, значения беззнакового типа занимают столько же ячеек, сколько и значения соответствующего знакового типа, но диапазон допустимых значений сдвигается так, чтобы начинаться с 0. Например, для типа char диапазон будет от -128 до 127, тогда как для типа unsigned char диапазон будет от 0 до 255.

Также в C имеются типы и для вещественных значений. В зависимости от занимаемой памяти различаются три типа: float, double, long double. Как и в случае с целыми числами первой системы, утверждать о них можно только то, что объем памяти для хранения их значений не убывает слева направо в этом списке. Например, в Visual Studio long double ничем не отличается от double.

Теоретически, для хранения вещественных чисел имеются две возможности: формат с фиксированной запятой и с плавающей. В первом случае обычно число представляется в виде дроби, знаменатель которой фиксирован раз и навсегда, а хранится в памяти только числитель, как обычное целое число.

Во втором случае число представлено, как произведение конечной двоичной дроби в диапазоне от половины до единицы, которая называется мантиссой, и степени двойки, показатель которой называется порядком. При этом для запоминания числа в памяти хранятся мантисса и порядок. Такой способ хранения вещественных чисел называется экспоненциальным или научным. Он принят, потому что позволяет представлять числа из гораздо более широкого диапазона, чем его альтернатива (с фиксированной запятой). Поэтому вещественные числа представлены в памяти компьютера совсем не так, как целые.

Кроме вышеперечисленных, в языке C есть много и других типов, и даже возможность определять свои собственные типы. Таких возможностей две: можно создать синоним уже существующего типа, а можно создать свой собственный новый тип, либо указав явно список его значений (так называемый перечислимый тип), либо указав структуру его значений, используя при этом уже имеющиеся типы как строительный материал (так называемый структурный тип).

Из всех этих возможностей упомянем пока только создание своего собственного типа, заданного набором своих значений. Это очень удобно в тех случаях, когда нужно запомнить выбор одного из нескольких возможных вариантов. Можно, конечно, явно закодировать эти варианты целыми числами, но этот способ требует всегда помнить способ кодирования и провоцирует много ошибок.

Чтобы определить перечислимый тип, в программе нужно написать

```
enum имя { список_значений_через_запятую };
```

Например, для определения типа, выбирающего один из дней недели, нужно написать

```
enum Day_of_week
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};
```

По традиции, имена возможных значений перечислимого типа пишутся прописными буквами. На самом деле, эти значения рассматриваются компилятором как константы целого типа. Если их значения не указаны явно, как в приведенном примере, они получают последовательные значения, начиная с 0.

Задача.

Написать определение перечислимого типа, значения которого представляют собой последовательные месяцы.

Однако, можно и явно указать численные эквиваленты значений перечислимого типа. Это нужно, например, для того, чтобы иметь возможность комбинировать эти значения, например, при помощи битовой операции «или» (подробнее это будет объяснено далее). В качестве примера приведем тип, значения которого представляют собой разрешения на определенные действия с файлом в операционных системах семейства UNIX (Linux, BSD, и т. д.).

```
enum File_permissions
{
    NONE,
    OTHER_EXEC,
    OTHER_WRITE,
    OTHER_READ = 4,
    GROUP_EXEC = 8,
    GROUP_WRITE = 16,
    GROUP_READ = 32,
    USER_EXEC = 64,
    USER_WRITE = 128,
    USER_READ = 256
};
```

После этого можно указать, например, что файл можно читать хозяину файла (пользователю, который его создал) и группе, а писать только хозяину, написав USER_READ | GROUP_READ | USER_WRITE. Здесь в качестве значений используются степени двойки, чтобы за каждое разрешение на определенное действие отвечал отдельный бит итогового значения, и не было путаницы, когда две разные комбинации разрешений представляются одним и тем же числом. Если бы в качестве значения OTHER_READ вместо 4 было бы указано 3, то тогда было бы непонятно, что означает число 3 (OTHER_READ или OTHER_EXEC | OTHER_WRITE?).

Задача.

Написать определение перечислимого типа, значения которого представляют собой отдельные ингредиенты пиццы: морковь, помидоры, кетчуп, ветчина, сыр, перец — чтобы их можно было использовать в любых сочетаниях независимо один от другого.

Вообще, назначение каждому значению и каждому месту для его хранения определенного типа и последующая проверка соответствия этих типов — вещь очень полезная для обнаружения ошибок еще на этапе компиляции программы, и она играет в программировании примерно ту же роль, какую играет в физике проверка формулы по единицам измерения входящих в формулу величин.

8 Явно выписанные константы.

В выражениях языка C можно использовать явно указанные целые числа (как в математике), а также вещественные числа в обычном виде, как десятичные дроби, только вместо десятичной запятой используется точка. Например, запись 2.731 означает «2 целых 731 тысячная». Можно также использовать вещественные числа в так называемом экспоненциальном или научном формате, когда к числу сзади дописывается (английская) буква «e» и за ней целое число (вся эта запись не должна содержать пробелы), которое означает, что исходное число надо умножить на 10 в указанной степени. Например, число 1.23e2 означает просто 123, а 1e-6 означает одну миллионную. Поскольку, как мы уже знаем, в памяти компьютера числа, целые или вещественные, записаны в двоичной системе, компилятору, встречающему десятичную запись чисел, приходится автоматически преобразовывать эти числа из десятичной записи в двоичную; для вещественных чисел это обстоятельство означает, увы, что большинство вещественных констант представляются в памяти компьютера лишь приближенно. Например, число 0.1 конечной двоичной дробью не является, и потому оно хранится в компьютере с небольшой, но ненулевой погрешностью.

В большинстве языков программирования, и C не является исключением, нельзя использовать слишком большие целые или вещественные числа. Более того, диапазон используемых чисел зависит от компьютера, операционной системы, компилятора и того, как эти числа записаны в программе. Как мы уже знаем, объем памяти, отводимый под то или иное число, определяется его типом.

Тип целого числа, явно указанного в программе, может определяться двумя способами. Первый состоит в том, чтобы компилятор определял его сам — если число содержит только цифры, компилятор определяет тип числа как минимальный, начиная с int, достаточный для его хранения. Однако, тип может быть указан и явно при помощи так называемых суффиксов — одной или нескольких букв после числа без пробелов. Суффикс u означает unsigned, l означает long, и ll — long long. Эти суффиксы можно комбинировать — ul означает unsigned long.

Также, целые числа можно записывать не только в десятичной системе, но и в некоторых других. Если запись числа начинается с ненулевой цифры, то оно рассматривается как записанное в десятичной системе, если с нуля, за которым следует цифра — в восьмеричной, с 0x и 0b начинаются шестнадцатеричная и двоичная запись соответственно. При этом в шестнадцатеричной записи начальные буквы английского алфавита служат цифрами от 10 (A) до 15 (F).

Задача. Напишите программу, выводящую на экран десятичную запись чисел 17253_8, $1AC9F_{16}$, 11010010011_2, где нижний индекс означает основание системы счисления.

Суффиксы позволяют явно назначать константам тип и в случае вещественных чисел. Так, суффикс f означает число типа float (по умолчанию, явно выписанное вещественное число относится к типу double).

Кроме чисел, есть еще две разновидности явно выписанных констант. Это символьные и строковые константы.

Символьные константы — это удобный способ получения в программе кодов отдельных символов. Например, запись 'A' представляет собой значение типа char, равное коду символа A. На тех компьютерах, на которых мы работаем (и на любых других, использующих кодировки ASCII или UTF-8), это будет 65.

Задача. Напишите программу, выводящую на экран коды символов B, C, D, a, b, {.

Кроме обычных, печатных символов, таким образом можно получать и другие, управляющие, символы. Для этого существуют так называемые escape-последовательности. Они начинаются с символа \и затем одного или нескольких обычных символов. Например, конструкция '\n' означает код символа перевода строки. Если вывести такой символ на экран, курсор переместится в начало следующей строки. Также можно указать явно код нужного нам символа, в восьмеричной (3 цифры) или шестнадцатеричной (x и затем две шестнадцатеричные цифры) системе счисления. Например, все тот же символ перевода строки (код 10 в десятичной системе счисления) может быть записан как '\012' или '\x0A'.

Символьные константы могут содержать только один символ. Некоторые компиляторы позволяют иметь несколько символов в одной символьной константе, но лучше так не делать.

Строковые константы представляют собой текст в двойных кавычках. В строковых константах допустимы любые символы, кроме символа с кодом 0 (не путать с символом '0', код которого — 48). Для представления в строковой константе управляющих символов годятся те же escape-последовательности, что и в символьных константах; в частности, для записи символа двойной кавычки перед ним нужно поставить \. Соответственно, чтобы представить сам символ обратной косой черты, нужно написать его дважды.

Если какая-либо константа используется в программе достаточно часто, ей можно назначить имя при помощи следующей конструкции:

```
const тип имя = значение;
```

В этой записи мы пользуемся тем соглашением из самого первого раздела данного курса, что слова по-английски означают ключевые слова, а по-русски — понятия. Применительно к данной записи это означает, что в программе, если мы хотим дать целочисленной (тип — int) константе 123456 имя nc, мы должны в программе написать

```
const int nc = 123456;
```

Бывает и так, что мы хотим запустить нашу программу несколько раз с разными значениями некоторой константы. В этом случае нам тоже может помочь назначение ей имени и в дальнейшем использование этого имени везде, где по смыслу требуется значение этой константы; иначе для изменения программы придется просмотреть программу полностью и исправить все те места, где эта константа используется, что может быть непросто. Например, если наша программа использует константу со значением, скажем, 20, то для каждого вхождения этого числа в программу нужно будет решить, это значение нашей константы или что-то другое (случайное совпадение чисел?). Также, вполне возможно, что число 19 в нашей программе — это на самом деле наша константа−1, вычисленное заранее для оптимизации.

9 Переменные

Сколько-нибудь сложное манипулирование со значениями требует запоминать эти значения и затем ими пользоваться в выражениях. Для этого и было изобретено понятие переменной. В C++ переменная — это поименованная область памяти для хранения в ней того или иного значения.

Каждая переменная, кроме хранящегося в ней значения, имеет имя и тип. Тип переменной говорит о том, как надо трактовать ту информацию, которая хранится в переменной. Имя переменной используется для указания на нее в выражениях.

Имя переменной должно быть, как говорят, идентификатором, т. е. словом из букв, цифр и знака подчеркивания, не начинающимся с цифры. Также имя переменной не должно совпадать ни с одним ключевым словом, например, int или else.

При использовании переменной в выражении важно, чтобы ее имя не являлось частью другого идентификатора или строковой константы(как и имена макроопредлений и их параметров).

Чтобы можно было использовать переменную, она должна быть объявлена или определена. Определение переменной означает указание компилятору зарезервировать память для хранения ее значения и адрес этой области памяти связать с имеющимся в определении переменной именем. Объявление переменной означает утверждение о том, что

переменная с такими именем и типом определена где-то еще (в другом файле или даже в том же самом файле, но позже). Определение или объявление переменной называется декларацией.

Простейшее определение переменной выглядит следующим образом:

```
тип имя;
```

Например, если мне нужна переменная для запоминания целого числа, и я решил назвать ее x, в программе я должен написать

```
int x;
```

Такая надпись означает, что компилятор выделил под эту переменную область памяти, достаточную для хранения целого числа типа int, и связал ее адрес с именем x, но значение этой переменной еще не определилось. Несколько усложнив эту запись, можно сразу же определить и значение переменной:

```
int x = 7;
```

Если мне нужны несколько переменных одного и того же типа, можно обойтись одним упоминанием этого типа:

```
int x = 7, y = 2, z = 5;
```

После определения переменной я могу записать в нее какое-либо значение, написав

```
x = 10;
```

Эта конструкция называется присваиванием. Если у переменной уже было какое-то значение, оно пропадает, и переменная получает новое значение. В отличие от Pascal и многих других языков программирования, в C присваивание — не оператор, а операция, возвращающая то значение, которое было присвоено. Это обстоятельство означает, в частности, следующее:

1) Если мы собираемся присвоить одно и то же значение 10 двум переменным x и y одновременно, можно сократить эту запись так:

```
x = y = 10;
```

2) Чтобы выполнить присваивание в программе, из этого выражения нужно сделать оператор, добавив в конце символ «;».

Для того, чтобы вспомнить значение, записанное в переменную, и воспользоваться им, достаточно просто указать ее имя:

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

Такая конструкция выведет значение переменной на экран; при этом значение переменной не исчезнет, оно останется записанным в ней до следующего присваивания. Наконец, можно использовать в присваиваниях выражения, в том числе содержащие имя этой самой переменной:

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

Такая конструкция увеличит значение переменной x на 1. В C имеется целый набор так называемых совмещенных операций присваивания. В данном случае текст этого присваивания можно сократить до

```
x += 1;
```

Отличие такого выражения присваивания от предыдущего только в том, что адрес переменной x будет вычисляться один раз, а не два. Если речь идет о простой переменной, это не имеет значения. Но если в качестве переменной используется достаточно сложное выражение, да еще с побочными эффектами, это становится очень важным.

В данном конкретном случае, это выражение можно еще сократить до

```
x++;
```

Наконец, скажем здесь о том, как можно определить синоним некоторого уже существующего типа (об этой возможности упоминалось в разделе про типы, но там не было сказано, как конкретно это делается). Для этого нужно написать декларацию переменной, которая имеет имя, как у нужного нам типа, а тип — тот, синоним которого мы хотим определить (без присвоения начального значения этой переменной, даже если тип — константный!). Далее, чтобы превратить эту декларацию в определение нужного нам типа, перед ней нужно поставить ключевое слово typedef.

Например, мы хотим для краткости создать новый тип uli, синоним типа unsigned long int. Сначала определим переменную uli с типом unsigned long int:

```
unsigned long int uli;
```

Наконец, перед этой декларацией поставим ключевое слово typedef:

```
typedef unsigned long int uli;
```

В итоге мы получаем декларацию, создающую синоним uli для типа unsigned long int.

В дальнейшем мы изучим много других разновидностей типов, но это правило построения синонимов для типов останется в силе.

Определенные таким образом синонимы типов действительно являются взаимозаменяемыми с теми типами, которых они синонимы, т. е. если у нас есть две разных переменных, типы которых — синонимы, то компилятор считает, что эти переменные имеют один и тот же тип, и мы запросто можем присвоить значение одной из этих переменных другой (конечно, если это не массив, — для тех, кто уже знает, что такое массив!).

10 Ввод значений

Некоторые программы для своей работы требуют дополнительных данных. Такие данные обычно вводятся с клавиатуры или читаются из файлов.

Для ввода данных с клавиатуры обычно используется следующий синтаксис:

```
scanf("%спецификатор", &имя_переменной);
```

В этом случае значение указанной переменной считывается из стандартного потока ввода (как правило, соответствует вводу с клавиатуры), причем конкретно способ считывания зависит от типа этой переменной и указанного спецификатора (одно должно соответствовать другому, иначе возможны ошибки). Спецификатор — как правило, буква, реже — несколько, указывающая на тип считываемого значения. Например, если в указанном фрагменте используется целая переменная, то спецификатор должен быть d, и в этом случае с клавиатуры считывается целое число. Но если переменная имеет тип char, то спецификатор должен быть c, и в этом случае считывается один символ и его код записывается в переменную.

Что касается второго параметра scanf, это — указатель на ту переменную, куда надо записывать результат. Он получается из имени этой переменной при помощи операции взятия адреса &. Ограничимся здесь пока этим объяснением, в дальнейшем будем обсуждать указатели гораздо подробнее.

Для ввода нескольких переменных можно использовать один вызов функции scanf с несколькими спецификаторами, по одному для каждого считываемого значения:

```
scanf("%d%d%d", &x, &y, &z);
```

причем это работает даже в том случае, если переменные имеют разные типы (в этом случае должны быть указаны разные спецификаторы для разных типов считываемых переменных).

11 Арифметические операции

Почти всякое выражение, кроме чисел и переменных, содержит еще арифметические операции, такие как сложение, вычитание, умножение, деление и т. д. Для умножения правила несколько отличаются от математических: в качестве значка для умножения используется звездочка, и пропускать его нельзя. Деление имеет одну особенность: если оно используется для целых чисел, т. е. оба его операнда имеют целый тип, результат будет целым (дробная часть отбрасывается), так что выражение $1/3$ имеет достаточно странный результат 0. Если нам нужно получить вещественный результат, нужно явно преобразовать один из операндов к вещественному типу, например так: $1.0/3$. Если в некоторой операции участвуют операнды разных типов, то один из них преобразуется к типу другого (целый и вещественный — к вещественному, занимающие разный объем памяти — к занимающему больший объем, знаковый и беззнаковый — к беззнаковому). Можно, однако, и явно преобразовать результат выражения к другому типу, если это необходимо, при помощи операции преобразования типов. Синтаксис ее таков: (новый_тип)выражение. Смысл этой операции в том, чтобы превратить результат выражения в значение нового_типа, по возможности изменив само значение минимальным образом. Например, результат выражения (double)5 будет 5.0. Эту операцию можно использовать при делении значения одной переменной целого типа на значение другой, тоже целого типа, чтобы результат получился вещественным:

```
int a,b;
scanf("%d%d", &a, &b);
printf("%f", (double)a/b);
```

Здесь нужно помнить, что сначала a преобразуется к типу double, и уже затем, как вещественное число, делится на b, а не результат деления целых чисел (с отброшенной дробной частью) преобразуется к вещественному типу, что, в сущности, бессмысленно.

А вот обратное преобразование типов (вещественное значение к целому типу) в большинстве случаев уже меняет значение (происходит округление в сторону 0, т. е. то, что идет после точки, отбрасывается; для отрицательных чисел этот способ округления отличается от взятия целой части на 1).

Здесь же уместно сказать о том, что операция ^ в C++ также имеется, но означает она вовсе не возведение в степень, а что она означает, будет сказано несколько позже.

Кроме того, для целых чисел имеется дополнительная операция % (остаток от деления), которая дает целый результат. Также, разумеется, можно использовать скобки, но только круглые (смысл квадратных скобок будет ясен несколько позже).

Примеры.

1. Вычислить выражение

$$\frac{3+5(6+3)-8\cdot 3+1}{21+50/(3+4(1+2))}$$

и вывести его результат на экран.

Решение:

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

int main()
{
    printf("%f\n", 3+5*(6+3)-8*3+1)/(21+50.0/(3+4*(1+2)));
    return EXIT_SUCCESS;
}
```

2. Вычислить выражение

$$\frac{(0,25-0,12)0,81+0,13^2-2,7/3,1}{3/7+2,97(8,05-8,1\cdot 6,07)}$$

и вывести его результат на экран.

Решение:

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

int main()
{
    printf("%f\n", (0.25-0.12)*0.81+0.13*0.13-2.7/3.1)/
          (3.0/7+2.97*(8.05-8.1*6.07));
    return EXIT_SUCCESS;
}
```

3. Ввести с клавиатуры 3 вещественных числа a, b и c и вывести значение выражения

$$a+b-c\frac{3ab+a^2}{bc}-(c+\frac{ab}{c})^2$$

на экран.

Решение:

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

int main()
{
    double a = 0, b = 0, c = 0;
    printf("a = "); scanf("%f", &a);
    printf("b = "); scanf("%f", &b);
    printf("c = "); scanf("%f", &c);
    printf("Результат = %f\n",
```

```
        a+b-c*(3*a*b+a*a)/(b*c)-(c+a*b/c)*(c+a*b/c));
    return EXIT_SUCCESS;
}
```

Задачи.

1. Вычислить выражение

$$\frac{3,1}{5,23^3+1}+\frac{4.1^3}{2,72^2+1}$$

и вывести его результат на экран.

2. Ввести с клавиатуры 3 вещественных числа a, b и c и вывести значение выражения

$$\frac{(ab+7c)^3-(a^2+b^2+c^2)}{b-ac-a(b+c))}$$

на экран.

3. Грузовик может увезти 32 коробки. Ввести число коробок, которые необходимо увезти и вывести минимальное достаточное для этого число грузовиков.

4. Вычислить 20!. Объяснить результат программы.

Одной из особенностей C является то обстоятельство, что тип переменных для хранения одиночных символов (он называется char) является целочисленным (соответствующая переменная на самом деле хранит код символа, т. е. его номер в специальной таблице символов), и потому символьные переменные вполне могут участвовать в арифметических выражениях, а также имеется возможность превращать целые числа в те символы, которые имеют соответствующий номер.

Пример. Ввести целое число от 1 до 26 и вывести большую латинскую букву, имеющую данный номер в алфавите.

Решение:

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

int main()
{
    int n;
    printf("Input the number: "); scanf("%d", &n);
    printf("Numbered letter = %c\n", 'A'-1+n);
    return EXIT_SUCCESS;
}
```

Задача.

Ввести оценку по 99-балльной шкале (от 0 до 99) и вывести разряд, соответствующий этой оценке (оценкам от 90 до 99 соответствует разряд 'A', от 80 до 89 — 'B', от 70 до 79 — 'C' и т. д.).

Также в C имеются и менее стандартные арифметические операции. К ним относятся операции инкремента и декремента, префиксные и постфиксные. Эти операции имеют один аргумент и могут применяться только к переменным, т. е. так называемым левым выражениям — выражениям, которые могут встречаться в левой части операции присваивания. Это могут быть как обычные переменные из одного из предыдущих разделов, так и ряд других выражений — какие именно, будет сказано несколько позже.

Операции инкремента и декремента, в отличие от обычных операций, рассмотренных ранее, имеют так называемый побочный эффект — они изменяют значение той переменной, к которой применяются.

В следующей таблице показано, как нужно использовать эти операции:

Операция	Выражение	Результат	Значение переменной n после вычисления операции
Префиксный инкремент	++n	$n+1$	$n+1$
Постфиксный инкремент	n++	n	$n+1$
Префиксный декремент	−−n	$n-1$	$n-1$
Постфиксный декремент	n−−	n	$n-1$

Здесь n — переменная, а n — ее значение до начала вычисления соответствующего выражения.

На использование этих операций в выражениях имеется одно ограничение: нельзя в одном и том же выражении, к одной и той же переменной, применять какие-либо из этих операций более одного раза.

Пример.

Следующая программа нарушает вышеприведенное правило и потому ее результат достаточно неожиданный.

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

int main()
{
    int n = 1;
    printf("result = %d %d %d\n", n++, n++, n++);
    return EXIT_SUCCESS;
}
```

12 Функции из стандартной математической библиотеки.

Для вещественных чисел в выражениях также можно использовать элементарные функции, названия некоторых из них отличаются от принятых в математике:

Функция	Запись	
	математика	язык C++
модуль	$\|x\|$	fabs(x)
синус	$\sin x$	sin(x)
косинус	$\cos x$	cos(x)
тангенс	$\mathrm{tg}\, x$	tan(x)
арксинус	$\arcsin x$	asin(x)
арккосинус	$\arccos x$	acos(x)
арктангенс	$\mathrm{arctg}\, x$	atan(x)
гиперболический синус	$\mathrm{sh}\, x$	sinh(x)
гиперболический косинус	$\mathrm{ch}\, x$	cosh(x)
гиперболический тангенс	$\mathrm{th}\, x$	tanh(x)
гиперболический арксинус	$\mathrm{arcsh}\, x$	asinh(x)
гиперболический арккосинус	$\mathrm{arcch}\, x$	acosh(x)
гиперболический арктангенс	$\mathrm{arcth}\, x$	atanh(x)
экспонента	e^x	exp(x)
натуральный логарифм	$\ln x$	log(x)
степень	x^y	pow(x,y)
квадратный корень	$\sqrt{x}$	sqrt(x)
корень	$\sqrt[y]{x}$	pow(x,1.0/y)

Все тригонометрические функции, прямые и обратные, работают с углами в радианах.

Для использования этих функций необходимо включить файл math.h при помощи директивы #include (т. е. в начале программы нужно написать #include <math.h>).

Примеры.

1. Написать программу, которая вычисляет выражение

$$\frac{\sin 5 + 1,75^2}{3e^{\cos 7}}$$

и выводит его результат на экран.

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

int main()
{
    printf("%f\n", (sin(5.0)+pow(1.75,2.0))/(3*exp(cos(7.0))));
    return EXIT_SUCCESS;
}
```

2. Написать программу, вводящую длины двух сторон треугольника и величину угла между ними и печатающую длину оставшейся стороны.

Решение:

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

int main()
{
    double a,b,gamma;
    printf("Input the lengths: "); scanf("%f %f", &a, &b);
    printf("Input the angle: "); scanf("%f", &gamma);
    printf("The length of the remaining side=%f\n",
        sqrt(a*a+b*b-2*a*b*cos(gamma)));
    return EXIT_SUCCESS;
}
```

Задачи.

1. Ввести координаты двух разных точек на плоскости и вывести коэффициенты уравнения прямой, проходящей через эти точки.

2. Ввести длины сторон треугольника и вывести его площадь.

3*. Ввести начальную сумму вклада и вывести число, через сколько лет сумма вклада превысит миллион (ежегодно начисляется 3%, которые добавляются к сумме вклада), без циклов.

4*. Ввести целое положительное число и вывести число цифр в нем в десятичной системе счисления (для тех, кто знает, что это такое: без циклов, массивов и строк!).

13 Параметры вывода

Вообще говоря, вид простейшей функции вывода нам уже известен. Однако, имеется ряд полезных возможностей, которые вполне могут пригодиться; о них и пойдет здесь речь.

Иногда бывает удобно, особенно при выводе таблиц, задавать ширину поля (т. е. места на экране, куда будет выведено число). Если выводимое число занимает меньше места, оно будет дополнено пробелами спереди. Это делается так:

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

int main()
{
    printf("Result=%15f\n", (sin(5)+pow(1.75,2))/(3*exp(cos(7))));
    return EXIT_SUCCESS;
}
```

В данном случае ширина поля равна 15 символам.

Для вещественных чисел, кроме того, удобно задавать точность, с которой выводится результат. Для этого можно задать требуемое число цифр после «.». Например, если в предыдущем примере нам требуется 10 цифр после «.», нужно написать

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

int main()
{
    printf("Result=%15.10f\n", (sin(5)+pow(1.75,2))/(3*exp(cos(7))));
    return EXIT_SUCCESS;
}
```

Задачи.

1. Вычислите в вещественных числах значение выражения

$$\frac{\sqrt[3]{\sin^2(1.28)+1}-26+\operatorname{arctg}(1.17+2.95)}{(2.01-\cos^2 3.86)^{5.84}+25.362}.$$

Выведите его значение на экран с подсказкой, шириной поля 20 и 15 значащими цифрами.

2. Написать программу, выводящую на экран таблицу значений функции $f(x) = \sin(x^2)\ln x$ для значений параметра x от 10 до 15 с шагом 0.5. При этом ширина поля для аргумента должна быть 7, с точностью 3 цифры после «.», для значения — ширина поля 17, с точностью 12 цифр после «.».

С помощью разных спецификаторов можно решать и другие задачи, например, выводить целые числа в восьмеричной (%o) или шестнадцатеричной (%x) системах счисления.

Задача.

Написать программу, переводящую целое число из десятичной в восьмеричную систему счисления.

14 Операции сравнения

Операций сравнения в C, как и в других языках, тоже шесть, и выглядят они так: < (меньше), > (больше), <= (меньше либо равно), >= (больше либо равно), == (равно) и != (не равно). Особенно нужно подчеркнуть, что = в C — это присваивание, а проверка на равенство — это == (источник многих ошибок в программах). Результаты этих операций: 0 для ложного результата и 1 — для истинного. В отличие от математики, в C, как и в большинстве языков программирования, неравенства нельзя соединять в цепочки: хотя выражение $10 > 7 > 2$ и является в C законным (программа успешно скомпилируется), но результатом его будет неожиданное число 0 (потому что это выражение понимается как (10>7)>2; выражение в скобках истинно, поэтому его значением будет 1, а получившееся таким образом неравенство 1>2 неверно — общий итог всего этого выражения будет ложь, т. е. 0). Как правильно записать на C подобного рода неравенства, мы обсудим чуть позже, когда речь пойдет о логических операциях.

Также уместно сказать здесь и о сравнении вещественных чисел на равенство. Дело в том, что вещественные числа представляются в компьютере лишь приближенно, с небольшой, но определенно ненулевой погрешностью, и тем более после арифметических операций, результат будет неточным из-за ошибок округления. Поэтому сравнивать вещественные числа на точное равенство практически бессмысленно, особенно если они получаются как результаты выражений. Результат всегда будет «ложь», даже если математически он должен быть истинным. Поэтому обычно задается некоторая небольшая положительная константа, для каждого из вещественных типов своя, в соответствии с точностью, обеспечиваемой данным типом, и вместо точного равенства проверяется, что модуль разности сравниваемых чисел не превосходит этой константы. При этом, никаких точных указаний насчет выбора конкретного значения этой константы нет, и при разных ее значениях результаты, вообще говоря, могут быть разными, так что ее значение надо подбирать экспериментально.

15 Логические операции

Еще в C имеются логические операции. В C в качестве логических значений используются все те же целые числа (0 — ложь, все остальное — истина).

Для логических операций используются следующие обозначения: «и» — &&, «или» — ||, «не» — ! (перед операндом). Логические операции трактуют свои операнды, как уже говорилось (ноль — ложь, все остальное — истина) и всегда возвращают 0 для ложного результата и 1 — для истинного. Таблица значений этих операций (вместо 1 операнд может иметь любое ненулевое значение) дана ниже:

x	y	x && y	x || y	!y
0	0	0	0	1
0	1	0	1	0
1	0	0	1	1
1	1	1	1	0

Например, первая строка этой таблицы (не заголовок) означает, в частности, что 0 && 0 будет 0 (при x==0 и y==0 имеем (x&&y)==0). Вообще, результат операции «и» будет истина (true) только в том случае, если оба выражения, которые она соединяет, имеют значение истина. Результат логического «или» будет истина (true), если хотя бы один операнд — истина, а логического «не» — истина, если операнд имеет значение ложь (false).

Логические операции позволяют обойти ограничение, запрещающее соединять неравенства в цепочки: вместо 10>7>2 можно и нужно писать 10>7 && 7>2.

Также, при вычислении логических выражений всегда используется сокращенное вычисление, т. е. сначала вычисляется первый операнд, и если этого достаточно для определения значения всего выражения, второй операнд не вычисляется. Например, если первый операнд логического «и» имеет значение ложь, то результатом всего выражения также будет ложь, независимо от значения второго операнда (который в этом случае даже не будет вычислен). Сокращенное вычисление позволяет писать логические выражения, например, из двух частей, такие, что вторая часть вообще не имеет смысла, если значение первой достаточно для определения значения всего выражения. Самый простой

пример — выражение x != 3 && 1/(x-3)<2. В этом выражении, если первый операнд имеет значение ложь, то второй вообще не может быть вычислен. Забегая вперед, можно сказать, что такого рода условия часто встречаются при работе с массивами (первая часть условия проверяет допустимость значения индекса, а вторая проверяет содержимое элемента массива с указанным индексом).

Задачи.

1. Ввести длины трех отрезков и вывести 1, если из них можно составить треугольник, и 0, если нельзя.

2. Ввести координаты трех точек на плоскости и вывести 1, если они лежат на одной прямой, и 0, если не лежат.

16 Побитовые операции

Побитовые операции по смыслу похожи на логические, но применяются в каждой позиции двоичного разложения отдельно. Например, результат битового «и» в младшем бите будет иметь результат логического «и» младших битов аргументов, в следующем бите — результат логического «и» вторых битов с конца, и так каждый бит результата побитового «и» будет результатом логического «и» битов в той же позиции аргументов. Побитовые операции используются для манипуляций отдельными битами в составе целых чисел, для экономии памяти или работы с регистрами аппаратуры. Битовые операции работают очень быстро по двум причинам: во-первых, целые числа хранятся в памяти компьютера в двоичной системе счисления, и во-вторых, битовые операции могут выполняться параллельно, в отличие от арифметических операций, где нужно в старших разрядах учитывать возможный перенос из младших.

Для побитового «и» в C используется двухместная операция & (а есть еще и одноместная операция с таким же значком — взятие адреса, она уже встречалась нам в функции ввода scanf, подробнее о ней позже), для «или» — |, для «не» — ~ (знак этой операции пишется перед операндом). Также имеется побитовая операция исключающего или ^ (этот значок означает отнюдь не возведение в степень, о чем уже говорилось раньше!).

Кроме операций над битами, происходящих из логических операций, применяемых в каждом двоичном разряде чисел-аргументов отдельно, для манипуляций двоичным представлением чисел имеются еще так называемые операции сдвига. Результатом их применения будет число, полученное при помощи сдвига составляющих его битов на определенное число разрядов влево или вправо. Важное замечание состоит в том, что операции сдвига возвращают результат сдвига, но операнды при этом не меняются. Например, если написать i«3, то результатом будет сдвинутое влево на три позиции значение переменной i, но значение «внутри» самой переменной i при этом не меняется.

Операция сдвига влево выглядит как <<. Левым операндом будет то число, результат сдвига которого нужно получить, правым — число позиций для сдвига. Например, значение 5<<2 будет равно 20. Арифметически сдвиг влево — это просто умножение числа на соответствующую степень двойки. При этом старшие биты, выдвигаемые за пределы числа, пропадают, а освободившиеся младшие биты просто заполняются нулями.

Что касается сдвигов вправо, здесь имеются альтернативы. В любом случае, выдвигаемые за пределы числа младшие биты исчезают, а вот чем заполняются освободившиеся старшие биты, возможны варианты. Логический сдвиг вправо заполняет освободившиеся слева разряды нулями, а арифметический — копиями знакового бита. Какой именно из этих сдвигов обозначается значком >>, зависит от реализации, так что лучше не пользоваться сдвигами для отрицательных чисел.

Пример. Написать выражение, дающее i-ю двоичную цифру числа n (0-я цифра — младшая).

Решение:

```
(n>>i) & 1
```

Задачи.

1. Ввести целое число n и вывести 2^n (без циклов и рекурсии).

2. Поменять местами два бита с заданными номерами в двоичном представлении целого положительного числа.

3. Вычеркнуть i-й бит из двоичного представления целого положительного числа (младшие i-го биты остаются на месте, старшие сдвигаются на один разряд вправо).

4. Написать программу, вводящую три целых числа и выводящую новое число, каждый бит которого равен тому значению, которое встречается чаще среди битов с таким же номером у введенных чисел (при помощи побитовых операций; без циклов).

5*. Ввести целое число (32-битное) и вывести число единиц в его двоичном представлении (без циклов).

6*. Описать словами результат следующего выражения: $x \;\&\; (x-1)$.

7*. Придумать, каким образом можно резко ускорить вычисления в игре «Жизнь».

8**. Написать фрагмент программы, по заданному числу n выдающий следующее за ним по порядку число, в двоичной записи которого столько же единиц, сколько в двоичной записи числа n.

17 Условная операция

Следующая удобная операция в C называется условной операцией (иногда она же называется тернарной операцией, поскольку это единственная операция языка C, имеющая три аргумента). Синтаксис ее таков:

```
условие? выражение1 : выражение2
```

При вычислении этого выражения сначала вычисляется условие, и если оно истинно, т. е. результат отличен от 0, то вычисляется выражение1 и его результат становится результатом всего выражения. Иначе вычисляется выражение2 и его результат становится результатом всего выражения. Эта операция удобна для анализа случаев прямо внутри выражения, поскольку синтаксис C не разрешает использовать условный оператор (как и любой другой оператор тоже) внутри выражения.

Пример. Ввести два целых числа и вывести наименьшее из них.

Решение:

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

int main()
{
    int a,b;
    printf("Input the numbers: "); scanf("%d%d", &a, &b);
    printf("Minimum=%d\n", (a>b? b : a));
    return EXIT_SUCCESS;
}
```

Задачи.

1. Ввести три вещественных числа и вывести наименьшее из них.
2. Ввести коэффициенты квадратного уравнения — три вещественных числа, и вывести число вещественных корней этого уравнения.
3. Ввести целое число n и вывести $(-1)^n$ (без использования функции pow и циклов).
4. Ввести целое число n и вывести n-ую производную функции cos.

18 Операция последования

Наконец, последняя из операций, которые мы рассмотрим здесь — операция последования; для ее обозначения используется запятая. Смысл этой операции в том, чтобы вычислить последовательно ее операнды. Значение всех операндов, кроме последнего, пропадает (они используются только ради их побочных эффектов), значение последнего становится значением всего выражения в целом. Эта операция, естественно, не распознается в декларациях и вызовах функций (где, как и следовало ожидать, запятая разделяет описания разных переменных или параметры функции). Операция последования удобна там, где нужно вычислить несколько выражений в том месте программы, где по правилам должно быть выражение (а не оператор, так что составной оператор туда не поставить). Например, если перед вычислением условия в условном операторе нужно выполнить присваивание, вполне можно воспользоваться операцией последования прямо в условии.

19 Оператор, сделанный из выражения

Одним из самых часто встречающихся операторов в программе на C является оператор, сделанный из выражения. Его синтаксис достаточно прост — выражение и затем точка с запятой. Действие этого оператора состоит в вычислении входящего в его состав выражения. Результат этого выражения исчезает, поэтому такого рода оператор используется только ради побочного эффекта своего выражения. В большинстве случаев это выражение представляет собой присваивание или ввод/вывод.

Одним из часто встречающихся частных случаев является пустой оператор. Он состоит только из точки с запятой (его выражение пусто) и ничего не делает. Используется он там, где по правилам синтаксиса должен стоять оператор, но делать ничего не нужно. Например (забегая вперед), тело цикла с условием, вся работа которого выполняется в самом условии. Также, такая конструкция позволяет поставить метку (подробнее — в разделе про оператор goto) там, где тоже делать ничего не нужно — например, в самом конце тела цикла за последним его оператором.

Важно понять, что в C++ точка с запятой не разделяет операторы, как в некоторых других языках программирования, например, паскале, а завершает оператор, сделанный из выражения, и ряд других операторов.

20 Составной оператор, локальные и глобальные переменные

Во всех управляющих структурах, где нужны вложенные операторы, таких, как, например, условный оператор или операторы цикла, разрешается использование в качестве вложенного лишь одного оператора. Однако часто нужно выполнить в случае истинности некоторого условия несколько операторов, или в цикле выполнить более одного оператора. Для этих случаев в C предусмотрена конструкция, называющаяся «составной оператор». Она состоит из последовательности операторов, заключенной в фигурные скобки. Действие такого оператора состоит в последовательном выполнении входящих в его состав операторов.

Операторы в составном операторе не разделяются ничем (еще раз напомним, что в C символ «;» не разделяет операторы, как в Pascal, а завершает оператор, сделанный из выражения, и ряд других операторов). В частности, это означает, что обе «;» в следующем фрагменте необходимы:

```
{ i = 0; j = 1; }
```

Правда, используя операцию последования, этот фрагмент можно переписать эквивалентным образом и без составного оператора:

```
i = 0, j = 1;
```

В отличие от Pascal, в C можно определять переменные в любом составном операторе (почти в любом месте), а не только в теле какой-либо функции.

Так определенные переменные видимы (на них можно сослаться) только в том составном операторе, где они объявлены, почему они и называются локальными.

Локальные переменные бывают двух видов:

а) Автоматические (по умолчанию) — такие переменные создаются (и инициализируются) в начале выполнения того составного оператора, в котором они определены, и уничтожаются, когда этот составной оператор закончит свою работу.

б) Статические (в декларации перед типом стоит ключевое слово static) — такие переменные создаются в начале работы программы; они инициализируются, когда первый раз выполняется их декларация (когда она выполняется второй и более раз, инициализация игнорируется); они уничтожаются, когда программа завершит свою работу. В частности, они сохраняют свои значения между периодами работы такого составного оператора. Так что при, скажем, повторном выполнении их составного оператора в начале их значения будут такими, какими они были в конце работы первого его выполнения.

Если один составной оператор вложен в другой, то вполне возможна ситуация, когда в каждом из них определена локальная переменная с одним и тем же именем. В этом случае, во внутреннем составном операторе имя этой переменной будет означать ту из этих переменных, которая определена во внутреннем составном операторе и скрывает ту, которая определена во внешнем.

Здесь уместно сказать и о так называемых глобальных переменных. Под глобальными переменными понимаются переменные, определенные в одном из исходных файлов программы, вне любого составного оператора. Обычно, такие переменные доступны во всей программе, но чтобы пользоваться ими в других исходных файлах, в них должно встречаться объявление такой переменной (та же декларация, но перед ней стоит ключевое слово extern и нет инициализации — обычно помещается в соответствующий заголовочный файл).

Наконец, скажем здесь и о внутреннем и внешнем связывании. Это понятие применяется к именам глобальных переменных и функций, определенных в программе. Если какое-то имя глобальной переменной подлежит внешнему связыванию, то такая переменная будет доступна из других файлов программы (конечно, если она объявлена в них). Но если у нас есть две одноименные глобальные переменные, каждая из которых подлежит внешнему связыванию, даже из разных исходных файлов, компоновщик выдаст ошибку — конфликт имен. Чтобы этого не происходило, можно сделать так, чтобы по крайней мере одна из конфликтующих переменных подлежала внутреннему связыванию, поставив перед ее определением ключевое слово static (это ключевое слово для локальных и глобальных переменных означает совсем разные вещи). Такое решение проблемы конфликта имен, правда, приведет к тому, что та глобальная переменная, которая подлежит внутреннему связыванию, будет видна только в том файле, в котором она определена.

21 Условный оператор

Синтаксис условного оператора выглядит так:

```
if(условие) оператор1 else оператор2
```

При выполнении условного оператора проверяется условие, и если оно истинно, выполняется оператор1, а если ложно — оператор2.

Обратим внимание на следующие важные моменты:

1) условие всегда заключается в скобки, и условием может быть любое выражение, имеющее результат (как оно трактуется в смысле истинности или ложности, уже говорилось ранее);

2) ключевое слово then отсутствует;

3) Ключевое слово else и отрицательная альтернатива (стоящий после него оператор) могут отсутствовать. В этом случае, если условие ложно, ничего не происходит.

Также, если один условный оператор вкладывается в другой, else относится к последнему if, еще не имеющему else. Если нам нужно, чтобы if без else был вложен в другой if, имеющий else, есть два варианта:

а) Заключить вложенный if без else в фигурные скобки.

б) Добавить к вложенному if вариант else; (с пустым оператором).

Задачи.

1. Часы работы магазина — с 9:00 до 18:00, перерыв на обед — с 13:30 до 14:30. Ввести момент времени (числа часов и минут) и напечатать текст «Открыто», если магазин в данный момент работает, и «Закрыто», если не работает.

2. Ввести координаты трех точек на плоскости и вывести слово «да», если они лежат на одной прямой, и «нет», если не лежат (при этом равенство вещественных чисел проверяется с точностью до 10^{-10}).

3. Ввести длины сторон двух треугольников и вывести текст «да», если треугольники подобны, и «нет», если не подобны.

22 Оператор выбора, оператор break

Синтаксис оператора выбора выглядит так:

```
switch(выражение_с_целочисленным_результатом)
{
case вариант1:
    операторы
case вариант2:
    операторы
...
default:
    операторы
}
```

Выполняется этот оператор так: вычисляется выражение, и ищется тот вариант, который совпадает с этим значением. Если такой есть, управление передается на соответствующий подходящему варианту case оператор. Если совпадения нет, но есть вариант default, управление передается на него. Наконец, если и варианта default нет, ничего не происходит.

Обратим внимание на следующие важные моменты:

1) в качестве вариантов могут выступать только константы (или константные выражения), значения которых известны на момент компиляции. Ни наборы значений (через «,»), ни диапазоны не допускаются;

2) после обнаружения соответствия значения выражения и какого-то из вариантов case выполняются все операторы, стоящие после соответствующего case, до тех пор, пока не встретится либо оператор «break;», либо конец тела оператора выбора (встречающиеся по дороге другие варианты case просто игнорируются, что называется провалом управления). Это правило позволяет выполнять некоторые операторы в случае нескольких вариантов, просто поставив несколько меток case подряд одна за другой перед нужными операторами: case 1: case 2: printf(“Неудовлетворительно“); break; (эти операторы будут выполнены в двух случаях, когда значение выражения под switch будет равно 1 или 2).

3) вариант default может стоять где угодно (в том числе и первым, хотя он все равно будет выбран, только если нет подходящих вариантов case).

4) Оператор выбора может анализировать символы и варианты могут содержать символьные константы, а вот строки он анализировать не умеет и ставить под case строковые константы нельзя: case “poor“: return 2; // ошибка!

Задачи.

1. Ввести вещественное число и первую букву единицы измерения (граммы, килограммы, центнеры, тонны). Ввести еще раз первую букву единицы измерения и преобразовать значение из одних единиц измерения (первая введенная буква) в другие (вторая введенная буква).

2. Ввести вещественное число, знак арифметической операции и еще одно вещественное число. Вычислить это выражение и вывести результат.

3*. Ввести сумму денег (целое число копеек до рубля включительно) и вывести эту сумму словами. Например, если вводится число 23, надо вывести «двадцать три копейки».

23 Операторы цикла, операторы break и continue

В C имеются 3 разновидности операторов цикла. Первая — цикл while. Синтаксис этого цикла таков:

```
while(условие) оператор
```

Смысл этой конструкции таков: вычисляется условие, и если оно истинно, выполняется оператор, затем снова вычисляется условие, и так до тех пор, пока условие не станет ложным; что касается условия, здесь справедливо все то, что было сказано об условии оператора if. Если условие сразу ложно, оператор не выполняется ни разу. Оператор в этой конструкции называется телом цикла. Однократное выполнение тела цикла называется итерацией цикла. Количество необходимых итераций определяется условием и телом цикла.

Следующий оператор цикла — цикл, в котором условие проверяется после выполнения его тела. Синтаксис его таков:

```
do оператор while(условие);
```

В теле этого цикла (между do и while) допустим только один оператор; если тело этого цикла должно содержать больше одного оператора, нужно использовать составной оператор. Здесь тело цикла всегда выполняется по крайней мере один раз.

Наконец, последний вариант цикла в C — это цикл for. Синтаксис его таков:

```
for(начало; условие; приращение) оператор
```

Здесь начало, условие и приращение — выражения. Смысл этой конструкции следующий: сначала вычисляется начало. Затем проверяется условие, и если оно истинно, выполняется оператор (тело цикла), после чего вычисляется приращение. Затем опять проверяется условие и т. д. до тех пор, пока условие не станет ложным. В частности, если условие сразу после вычисления начала ложно, тело цикла не выполняется ни разу.

Кроме вышеперечисленных операторов, в C имеются еще два оператора для управления выполнением циклов — break; и continue;. Первый из них вызывает немедленное прекращение самого внутреннего цикла, в котором находится (впрочем, если он стоит в теле switch, который вложен в этот цикл — он прерывает switch). Второй вызывает немедленное прекращение текущей итерации такого цикла и переход к следующей. Для циклов while и do ... while это означает переход к проверке условия, для for — переход к вычислению приращения.

К сожалению, эти операторы умеют прерывать только самый внутренний цикл, в котором они находятся. Для прерывания нескольких вложенных циклов одновременно нужно использовать оператор goto.

Задачи.

1. Вычислить выражение, состоящее из вещественных чисел и арифметических операций, считая, что операции выполняются слева направо (без учета приоритетов; конец выражения — символ =).

2. Ввести вещественное число x и вычислить приближение к e^x, пользуясь формулой

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Закончить вычисления, когда очередное слагаемое станет меньше 10^{-6}.

3. Напечатать наибольшее и следующее по величине число из 10 введенных. Предполагается, что все введенные числа различны.

4. Ввести с клавиатуры последовательность символов (конец — символ `"."`) и вывести максимальную длину подпоследовательности вида «abcabc...» (между соседними символами подпоследовательности могут встречаться любые другие символы в любом количестве, и подпоследовательность может оканчиваться любым символом из «a», «b» и «c», т. е. не только «c»).

5. Напечтать в терминальном окне окружность из символов «*» (при помощи циклов).

6. Ввести 20 символов и вывести максимальное число, содержащееся в этой строке (под числом понимается любая последовательность подряд идущих цифр, можно предполагать, что это число всегда влезает в unsigned long long int).

24 Метки и оператор goto

Последний оператор, который мы рассмотрим здесь (остальные — позже) — это goto. Синтаксис его таков:

```
goto метка;
```

Метка — это идентификатор, т. е. последовательность букв, цифр и знаков подчеркивания, не начинающаяся с цифры. Метка должна быть определена в той же функции, в которой стоит оператор goto. Синтаксис определения метки:

```
метка: оператор;
```

Так пометить можно любой оператор; выполнение goto приведет к выполнению этого оператора, и далее по порядку. Единственное ограничение состоит в том, что оператор goto позволяет выходить из составного оператора, но не позволяет входить в составной оператор или внутрь другой управляющей конструкции, например, цикла.

Оператор goto очень удобен для одновременного прерывания нескольких вложенных операторов цикла. Он также очень удобен для записи на языке С алгоритмов, представленных достаточно сложными блок-схемами.

Задача.

Переписать на язык С алгоритмы, предоставленные блок-схемами, которые Вы придумали в начале этого курса. Добиться того, чтобы Ваши программы компилировались и выдавали правильные ответы на нескольких тестовых примерах.

25 Необходимость массивов. Их объявления и определения, доступ к элементам.

Для объяснения того, что такое массивы и зачем они нужны, опишем здесь задачу, которую я почерпнул в одной из книжек по программированию.

Задача состоит в следующем: пусть у нас есть некоторая программа, и в ней имеется целая переменная x, в определенной точке программы принимающая значения от 0 до 500 (этот факт мы считаем известным). Нам требуется выяснить, сколько раз эта переменная принимает те или иные свои значения в этой точке, т. е. сколько раз (в указанной точке программы) она будет иметь значение 0, сколько раз она будет иметь значение 1, и т. д. до 500.

Самый тупой и очевидный способ решить эту задачу — завести в нашей программе 501 дополнительную целую переменную с начальным значением 0, примерно так:

```
int count000 = 0;
int count001 = 0;
.................
int count500 = 0;
```

а в интересующей нас точке программы проанализировать значение переменной x и подправить соответствующий счетчик, примерно так:

```
switch(x)
{
    case   0: count000++; break;
    case   1: count001++; break;
         .................
    case 500: count500++; break;
    default: exit(1); //такого быть не может
}
```

Для того, чтобы исправить это безобразие, и были изобретены массивы. Массив есть средство для объединения некоторого количества однотипных переменных под одним именем, доступных по номеру. Главное достоинство массива по сравнению с отдельными переменными состоит в том, что номер, выбирающий конкретную переменную из всей совокупности, не обязан быть константой, а может быть результатом любого выражения, возвращающего целое число.

Одномерный (элементы которого доступны по одному номеру) массив в С определяется следующим образом:

```
тип_элемента имя_массива[число_элементов];
```

Например, для задачи из начала данного файла нам потребуется массив из 501 целой переменной; его можно определить так:

```
int counters[501];
```

В С, однако, бывают и многомерные (элементы которых доступны по нескольким номерам, значения которых независимы один от другого) массивы. Их определение отличается от вышеприведенного только тем, что за именем массива ставятся несколько чисел, каждое в своей паре квадратных скобок. Эти числа означают количества возможных индексов по соответствующему измерению. Например, следующее определение создает двумерный массив (матрицу) из вещественных чисел:

```
double m[10][7];
```

Обычно, у матриц первый индекс называется номером строки, а второй — столбца, так что матрица m имеет 10 строк и 7 столбцов. Увы, эту запись нельзя сократить до

```
double m[10, 7];
```

как в Pascal.

Вообще говоря, массивы встречаются и большей размерности (трех-, четырех- и более мерные, но, поскольку с ростом числа измерений число элементов массива растет экспоненциально, массивы большого числа измерений встречаются очень редко).

В одной декларации можно сочетать определение нескольких массивов с одинаковым типом элементов, и даже сочетать в одной декларации определение массивов с определением обычных переменных указанного типа. Например, декларация

```
int A[10], B[20], C;
```

определяет два массива A и B из 10 и 20 целых чисел соответственно, а также обычную целую переменную C.

Довольно долго при определении массива число его элементов должно было быть константным выражением, т. е. таким, в которое входят только константы и стандартные операции, и его значение должно было быть известно на момент компиляции программы. Сейчас уже это требование во многих компиляторах осаблено: локальные по отношению к функциям (т. е. определенные в функциях) массивы вполне могут иметь число элементов, которое определяется только во время выполнения программы, например так:

```
void f(int n)
{
    int A[n];
    .........
}
```

Впрочем, при определении массива число его элементов фиксируется и в дальнейшем меняться не может.

Кроме определения, глобальные массивы допускают и объявления — для этого нужно поставить перед декларацией ключевое слово extern.

Как и обычные переменные, массивы в C можно инициализировать в определении, т. е. прямо в определении указывать начальные значения их элементов. Делается это так:

```
тип_элемента имя_массива[число_элементов] = { элементы_через_запятую };
```

При этом число начальных значений элементов должно быть не больше указанного в квадратных скобках числа элементов массива. Меньше оно может быть; в этом случае недостающие начальные значения считаются нулями. Если начальные значения указаны для всех элементов, число элементов массива в квадратных скобках можно не писать — компилятор может посчитать число начальных значений сам, и выделить под них массив соответствующего размера.

Например, для массива счетчиков из рассмотренной в начале файла задачи, следующая декларация инициализирует массив целых чисел нулями

```
int counters[501] = { 0 };
```

А следующая — числами по порядку, при этом размер массива определяется автоматически:

```
int A[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

В случае, когда размер массива определяется автоматически, он может быть вычислен при помощи выражения

```
sizeof(имя_массива)/sizeof(тип_элемента)
```

Например, для последней декларации массива A значением выражения sizeof(A)/sizeof(int) будет 10.

Для удобства инициализации массивов символов, если тип элемента массива — char, то можно инициализировать такой массив при помощи строковой константы. Составляющие ее символы в этом случае записываются в массив, начиная с первого элемента по порядку, и после последнего символа строковой константы записывается еще один символ с кодом 0. Это нужно для того, чтобы строка, записанная таким образом в массив, правильно обрабатывалась функциями для работы со строками из стандартной библиотеки языка C. К только что описанному способу инициализации массивов символов относятся те же замечания, что и к обычному (насчет количества начальных значений).

Например, можно написать следующее:

```
char s[10] = "Hello!";
```

Такая декларация не только определит массив s из 10 символов, но и запишет в него строку «Hello!» с завершающим символом с кодом 0. А вот следующая декларация

```
char s[6] = "Hello!";
```

ошибочна, поскольку в такой массив завершающий нулевой символ не влезет.

Инициализировать двумерный массив можно двумя способами: либо перечислить все его элементы подряд по строкам, как и для одномерного массива, либо указав элементы каждой строки отдельно, заключив их в {}. Например, каждая из следующих деклараций вполне законна и обе они присваивают элементам массива M одинаковые начальные значения:

```
int M[3][2] = { 1, 2, 3, 4, 5, 6 };
int M[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
```

Напоследок скажем здесь о том, как можно несколько упростить задачу переделки программы, если нужно изменить используемые в ней размеры массивов. Для этого обычно объявляют заранее именованные константы, которые и используются затем в качестве таких размеров везде, где соответствующий размер требуется по логике программы. Преимущество такого подхода очевидно — если требуется изменить размер какого-либо массива в программе, достаточно поменять значение всего одной именованной константы и перекомпилировать программу. В противном случае нужно было бы просмотреть всю программу, и изменить указанные в ней числа, причем вопрос о том, нужно ли менять то или иное число, и если да, то как именно, может быть достаточно сложен, так что «find and replace» из состава текстового редактора в таком случае обычно вовсе непригоден.

Доступ к элементу одномерного массива может осуществляться при помощи конструкции

```
имя_массива[индекс]
```

причем эту конструкцию можно использовать как в правой части операции присваивания (тогда берется значение указанного элемента массива), так и в левой (тогда указанному элементу массива присваивается значение выражения, стоящего в правой части этой операции присваивания).

Однако, нужно помнить следующие правила доступа к элементам массивов в C:

1) индексы всех массивов всегда начинаются с 0, и должны быть целыми числами;

2) сам по себе C не проверяет правильность индекса, т. е. условие, что индекс лежит в допустимых пределах (от 0 до числа элементов массива−1).

Поскольку тип char считается в C целочисленным, допустимо использовать переменные и константы такого типа в качестве индексов, однако нужно помнить, что на некоторых компиляторах для символов, не входящих в набор ASCII, эти значения могут быть отрицательными.

Пример.

Для задачи из введения, если вместо отдельных счетчиков завести массив, как указано выше, вместо более чем громоздкого оператора выбора можно написать следующий фрагмент:

```
if(x<0 || x>500) exit(1);//так не бывает
else counters[x]++;
```

Доступ к элементу двумерного массива осуществляется следующим образом:

```
имя_массива[индекс_строки][индекс_столбца]
```

Как и в определении двумерного массива, каждый индекс должен быть заключен в свои квадратные скобки.

В программировании очень часто встречается задача перебора всех элементов некоторого массива. В C эта задача может быть решена при помощи обычного цикла for, перебрав соответствующий диапазон индексов.

Задачи.

1. Ввести с клавиатуры массив из 50 чисел от 0 до 20 и распечатать его как обычную столбцовую диаграмму (элементам массива соответствуют столбцы из символов «*»).

2. Диаметром множества точек называется максимальное расстояние между двумя точками из этого множества (если множество конечное, этот максимум всегда существует). Ввести координаты 15 точек на плоскости и вычислить диаметр этого множества.

3. Ввести элементы квадратной матрицы 5×5, коэффициенты многочлена степени 6 и посчитать значение многочлена от квадратной матрицы.

4. Дан двумерный массив 8x8, представляющий шахматную доску (элемент, равный 0 означает пустую клетку, 1 означает, что на этой клетке стоит ферзь). Определить, найдутся ли два ферзя, бьющие друг друга (линейное время, константная память).

26 Указатели

Исторически понятие массива оказалось тесно связано с понятием указателя. Что это такое и зачем оно нужно, обсудим в этом файле.

Начнем со следующей задачи. Предположим, в программе имеются 3 целые переменные x, y, z. Задача состоит в том, чтобы в одном месте программы выбрать одну из них, а в другом месте — присвоить выбранной переменной значение 1.

Самое простое решение этой задачи таково. Мы заводим еще одну целую переменную, например t, и договариваемся о том, что она может иметь значения от 1 до 3, причем ее значению 1 соответствует выбор x, значению 2 — y и 3 — z. В том месте программы, где нужно выбирать одну из трех переменных, мы присваиваем t соответствующее значение; например, выбор переменной y будет выглядеть как «t=2;». Ну а в том месте, где выбранной переменной нужно присвоить 1, будет стоять следующий фрагмент:

```
switch(t)
{
case 1: x=1; break;
case 2: y=1; break;
case 3: z=1;
}
```

Это решение при всей своей простоте имеет ряд недостатков. Отметим два из них. Первый состоит в том, что указанная конструкция усложняется с ростом числа переменных, из которых мы хотим выбрать одну. Второй связан с тем, что нам нужно помнить, какое значение специальной переменной какую переменную означает.

Для устранения этих недостатков и было изобретено понятие указателя. По-прежнему нужна дополнительная переменная, но теперь она имеет не целый тип, а некоторый специальный тип, который называется «тип указателя на целую переменную» (если мы хотим выбирать одну из целых переменных; если же нам нужно выбрать одну из вещественных переменных, нужен уже другой тип — тип указателя на вещественную переменную).

Переменная-указатель объявляется почти так же, как и обычная переменная, но с добавлением символа «*» перед именем переменной. Итак, специальная переменная (называемая указателем) заводится так:

```
int *t;
```

В этом определении * относится к t, а не к int. Это означает, что если я хочу в одной декларации определить несколько указателей, * должна ставиться перед каждым именем:

```
int *p, *q;
```

Как и любую переменную, указатель можно инициализировать при объявлении.

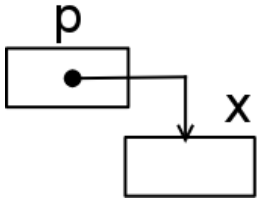
Теперь, чтобы выбрать одну из целых переменных (а этот способ позволяет выбирать любую из целых переменных, определенных в программе), например x, нужно написать:

```
t = &x;
```

В том же месте, где нужно присвоить выбранной переменной некоторое значение, например, 1, нужно написать:

```
*t = 1;
```

В дальнейшем, если переменная-указатель выбирает некоторую другую переменную, мы будем говорить, что указатель указывает на выбираемую переменную. Тот факт, что указатель p указывает на переменную x, обычно изображается следующим образом:



В C также имеется некоторый специальный тип указателя «неизвестно на что». Такие указатели определяются с ключевым словом void:

```
void *p;
```

Такому указателю можно присвоить любой указатель (т. е. указатель на любой тип), но пользоваться им непосредственно нельзя — сначала нужно явно привести его к типу указателя на конкретный тип, написав перед ним (тип *) (см. операцию преобразования типов).

Теперь скажем пару слов о реализации понятия указателя. Как известно, память компьютера состоит из пронумерованных от 0 и далее ячеек, каждая из которых может хранить небольшое целое число (более конкретно, от 0 до 255). Номер каждой такой ячейки называется ее адресом, он представляет собой обычное целое число. Так вот, в переменной типа указателя (независимо от того, на переменные какого типа он указывает) хранится просто адрес первой ячейки, которую занимает та переменная, на которую он указывает. Если целая переменная x занимает ячейки с адресами от 300 до 303, то после присваивания t = &x в переменной t окажется число 300.

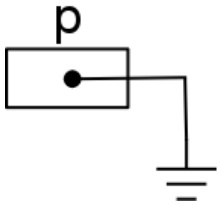
Однако, компилятор различает типы указателей по типу переменных, на которые они указывают; в частности, int* и double* (т. е. указатели на целые и указатели на вещественные переменные) считаются разными типами, и нельзя переменной одного типа указателя присвоить переменную другого типа указателя.

Первая из рассмотренных выше операций & называется операцией взятия адреса. Ее можно применять только к таким выражениям, которые могут стоять в левой части оператора присваивания, например, именам переменных или элементам массивов. Ее результатом будет указатель соответствующего типа, выбирающий (или указывающий на) ту переменную, к которой применяется эта операция.

Вторая операция, *, называется операцией разыменования. Она может применяться только к указателям и дает ту переменную, на которую этот указатель указывает. Результат этой операции может встречаться как в левой части оператора присваивания, что означает присваивание выбранной указателем переменной, так и в правой, что означает использование значения указываемой переменной.

Таким образом, если p — указатель на целую переменную, то *p обозначает ту переменную, на которую указывает p. К нулевому указателю (см. следующий раздел) эту операцию применять нельзя, однако эту ошибку замечает обычно не компилятор, а операционная система во время выполнения программы. К указателю неизвестно на что эту операцию тоже применять нельзя; если известно, на что на самом деле он указывает, нужно воспользоваться операцией преобразования типов для превращения его в обычный указатель. Например, если указатель p определен при помощи void *p; но в определенном месте программы известно, что он указывает на целое число, до этого целого числа можно добраться, написав *(int *)p. При этом, разумеется, компилятор никак не может проверить, что этот указатель действительно указывает на переменную типа целое число, а не вещественное; он не может проверить даже того, что этот указатель вообще указывает на какую-либо переменную, а не на пустую или даже недоступную программе область памяти.

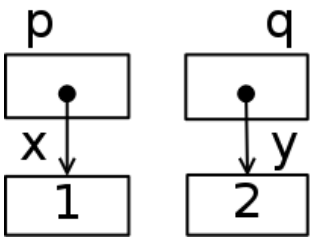
Оказалось очень удобным иметь некоторое специальное значение указателя, которое говорит о том, что указатель не указывает вообще ни на какую переменную. Такое значение указателя обозначается просто целым числом 0 и может быть присвоено указателю любого типа. Иногда, впрочем, вместо явного числа 0 используется макроопределение NULL,которое имеет значение 0. Тот факт, что указатель p не указывает ни на что, обычно изображается следующим образом:



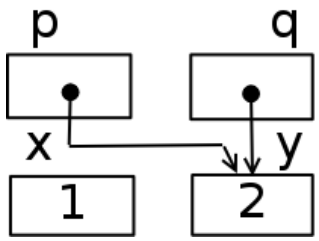
Указатели, как и переменные других типов, можно присваивать друг другу (только если они указывают на переменные одного и того же типа), сравнивать на равенство или отсутствие равенства. Равными считаются указатели, выбирающие одну и ту же переменную, т. е. такие, в которых хранится один и тот же адрес.

Присваивание указателей и присваивание переменных, на которые они указывают — совсем не одно и то же.

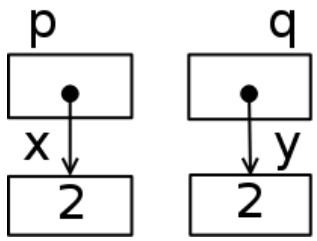
Пусть у нас есть 2 указателя p и q, каждый из которых указывает на свою переменную (предположим, что эти переменные, а следовательно, и указатели на них имеют одинаковый тип), эта ситуация изображена на следующем рисунке:



После «p=q;» оба указателя начнут указывать на одну и ту же переменную (на которую раньше указывал только q); при этом значения обоих переменных не изменятся (см. следующий рисунок).



Если же выполняется оператор «*p=*q;», то содержимое указателей не меняется, они по-прежнему указывают на те же переменные, что и раньше, но содержимое той переменной, на которую указывает q, записывается в ту переменную, на которую указывает p (см. следующий рисунок).



Задачи.

1. Объявить указатель на `double`, на `char`, два указателя на `float`, указатель неизвестно на что, тип указателя на `bool`, тип указателя на `unsigned int`.
2. Найти все ошибки в следующем фрагменте: `double *p = &3.14;`
3. Найти все ошибки в следующем фрагменте: `int x,y; int *p = &(x+y);`
4. Найти все ошибки в следующем фрагменте: `int *q = 0; *q = 7;`
5. Чему будет равно `y` после выполнения операторов `int y = 5; int *p = &y; *p = *p + 11;`
6. Попытаться ввести указатель; если получится, вывести его.
7. Определить размер указателя на целое; сравнить его с размерами указателей на `char` и на `double`.
8. Объявить локальную переменную типа `int` в функции `main` и вывести ее адрес. Проделать то же самое с целой переменной класса памяти `static` и с глобальной целой переменной.
9. Объявить указатель `p` на целое, инициализировать его значением 0 и вывести `*p`.
10. Объявить указатель `p` на целое, инициализировать его значением 0 и попытаться присвоить что-нибудь `*p`.

27 Арифметика указателей. Связь с массивами.

В C разрешены некоторые арифметические операции с указателями. Однако, все такие операции неявно предполагают, что все участвующие в них указатели (как исходные и промежуточные результаты, так и окончательные) указывают на элементы одного и того же массива. Однако, компилятор никак не проверяет истинность этого предположения и соблюдение его всецело на совести программиста.

Если указатель p указывает на элемент массива с индексом i, то p+j, где j — целое число, будет указывать на элемент того же массива с индексом i+j. При этом совершенно неважно, положительно j или отрицательно, лишь бы окончательный индекс лежал в допустимых пределах; аналогичным образом, p-j указывает на элемент с индексом i-j. По тем же правилам работают и операции инкремента и декремента. Также, если указатели p и q указывают на элементы с индексами i и j, то выражение p-q имеет целый результат и равно i-j (опять же неважно, положительно это значение, равно нулю или отрицательно).

Этим, однако, арифметика указателей и ограничивается. Несмотря на то, что указатели, с точки зрения внутреннего устройства, это просто адреса в памяти (целые числа), тем не менее их нельзя складывать или умножать друг на друга или даже на целые числа. Если такая нужда возникает (и Вы знаете, что делаете), можно явно превратить указатель в целое число при помощи операции преобразования типов, написав перед ним (int). Затем можно проделать необходимые манипуляции с полученным целым числом, и наконец, снова превратить теперь уже целое число обратно в указатель при помощи той же операции преобразования типов, написав перед ним (тип *), где тип — тип той переменной, в указатель на которую Вы хотите превратить полученное целое число.

Заметим здесь еще следующее. Во-первых, в большинстве выражений, в которых участвует имя массива, оно понимается как указатель на первый элемент этого массива (с индексом 0). Во-вторых, можно индексировать указатели

так же, как и массивы: запись p[i] означает то же самое, что и *(p+i). Однако, это работает только для одномерных массивов: имя двумерного массива понимается как указатель на первую строку (одномерный массив), и не приводится к типу «указатель на указатель». Также, из этого правила имеются очевидные исключения: в выражении sizeof M, если M — массив (неважно, одномерный или многомерный), вычисляется размер массива M, а не указателя.

Напоследок скажем только, что арифметика указателей не работает для указателей неизвестно на что (типа void *).

Задачи.

1. Объявить последовательно переменные типов `int`, `char`, `bool`, `short`, `long`, `float`, `double`, `long double`, `int`. Вывести их адреса; сравнить разности адресов с размерами переменных, выдаваемыми операцией `sizeof`.

2. Написать функцию, принимающую два указателя на `float` (указывающие на элементы одного и того же массива), и возвращающую указатель, делящий интервал между ними (приблизительно) в отношении 3:5.

3*. Написать функцию, принимающую 5 указателей на вещественные переменные (в пределах одного массива) и возвращающую такой указатель на элемент того же массива, что сумма модулей разностей между этим указателем и указателями-параметрами минимальна.

Мы уже говорили о том, что можно преобразовать тип указателя в целое число и обратно. Оказывается, принудительно можно преобразовать любой тип указателя в любой другой. Это приведет к тому, что содержимое памяти по этому адресу начнет трактоваться по-другому. Опреация преобразования типов указателей никак не меняет содержимого памяти по данному адресу; она лишь меняет способ трактовки этого содержимого, который тоже может быть важен. Например, и число типа int, и число типа float занимают по 4 байта. Но способ представления конкретных чисел, например 5 для int и 5.0 для float, существенно отличается один от другого. Поэтому следующий фрагмент выдаст совсем не 5.0, а нечто другое (что именно?):

```
int x = 5;
cout<<*(float*)&x<<endl;
```

Однако, такое преобразование переменных из одного типа в другой, сохраняющее не значение переменной, а содержимое памяти по данному адресу, иногда бывает важно. Например, таким образом можно превратить какое-либо значение в набор байтов, чтобы потом обрабатывать эти байты по-отдельности, скажем, для записи значения в файл в двоичном виде или передачи значения по сети.

Кроме обсуждаемого выше явного преобразования типов указателей имеется и неявное, которое происходит автоматически, без всяких усилий программиста. Это имеет место в следующих случаях:

1) Целое значение 0 (явно выписанное, и только оно) преобразуется в указатель любого типа, и означает указатель, никуда не указывающий.

2) Любой указатель может быть преобразован в тип void *, т. е. любой указатель можно присвоить переменной типа void * (обратное преобразование возможно только явно).

3) Любой указатель на один тип может быть преобразован в указатель на другой тип, который является синонимом первого, созданным при помощи конструкции typedef.

4) Наконец, при наследовании указатель на объект производного класса может быть преобразован в указатель на объект базового класса (обратное преобразование, как и в п. 2, возможно только явно).

Задачи.

1. Процессор называется Big Endian, если целые числа, занимающие несколько байтов, хранятся по следующему правилу: младший байт записан по наибольшему адресу, и Little Endian, если младший байт хранится по наименьшему адресу; если же порядок записи байтов в составе целого числа не совпадает ни с одним из упомянутых, назовем это Mixed Endian. Определить тип процессора, на котором мы работаем.

2. Написать макроопределение, выдающее значение последнего байта той переменной, на которую указывает параметр (это, естественно, указатель — больше о нем ничего не известно).

3. Написать функцию, принимающую указатель на double и 2 целых числа типа int, и записывающую в ту переменную, на которую указывает указатель-параметр, число типа double, составленное из двух целых чисел-параметров (число типа double занимает 8 байтов; первое целое число должно занимать байты с 0 по 3, второе — с 4 по 7).

4. Написать функцию, принимающую указатель неизвестно на что, и возвращающую другой указатель неизвесто на что, целочисленное значение которого равно $x^2 + 3x + 1$, где x — целочисленное значение указателя-параметра.

28 Понятие функции

В языке C предусмотрена возможность поименовать некоторый фрагмент программы, чтобы затем к нему обращаться многократно, не выписывая его каждый раз заново, а лишь указывая его имя. Такие именованные фрагменты программ называются функциями (в других языках программирования встречаются также термины «процедура» и «подпрограмма»). Они могут требовать для своей работы некоторой дополнительной информации (обычно называемой параметрами) и вырабатывать (как обычно говорится в программировании, возвращать) некоторое значение, которое можно использовать в дальнейших вычислениях.

К достоинствам понятия функции можно отнести следующее:

1) Если в отдельную функцию выносится достаточно длинный и часто используемый фрагмент программы, длина всей программы резко сокращается.

2) Если в отдельную функцию выносится законченный и замкнутый по смыслу фрагмент программы, решающий отдельную независимую от других подзадачу всей решаемой задачи, резко упрощается проектирование и реализация программы. Вообще, разбивка всей задачи на отдельные, независимые одна от другой подзадачи сильно упрощает процесс программирования, особенно тогда, когда разработка ведется командой разработчиков.

Пункт 2) позволяет применять при разработке программ метод проектирования сверху вниз, состоящий в следующем. Сначала вся задача, которую должна решать программа, разделяется на небольшое количество независимых подзадач, и пишется главная функция программы (как мы уже знаем, в C она всегда называется main), решающая поставленную задачу, с использованием функций (реализация которых пока отсутствует), решающих указанные подзадачи. Далее, к каждой из поставленных подзадач применяется та же самая технология, и так до тех пор, пока текущие подзадачи не окажутся настолько простыми, что их решение является очевидным или уже имеющимся в библиотеке (стандартной или другой доступной для использования).

В качестве простейшего примера рассмотрим программу, печатающую строку из 70 символов «*», затем строку «Title: Program 1», затем опять строку из 70 символов «*», потом строку «The end» и наконец опять строку из 70 символов «*»:

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

int main()
{
    int i;
    for(i = 0; i < 70; i++) printf("*"); printf("\n");
    printf("Title: Program 1\n");
    for(i = 0; i < 70; i++) printf("*"); printf("\n");
    printf("The end\n");
    for(i = 0; i < 70; i++) printf("*"); printf("\n");
    return EXIT_SUCCESS;
}
```

Нетрудно заметить, что в этой программе имеется фрагмент, повторяющийся 3 раза. Поэтому можно обозначить его, например, starline и определить один раз, а затем указывать только это имя:

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

void starline()
{
    int i;
    for(i = 0; i < 70; i++) printf("*"); printf("\n");
}

int main()
{
    starline();
    printf("Title: Program 1\n");
    starline();
    printf("The end\n");
    starline();
    return EXIT_SUCCESS;
}
```

В данном случае starline — это функция, не имеющая параметров и не возвращающая результата; следующий текст — определение этой функции:

```
void starline()
{
    int i;
    for(i = 0; i < 70; i++) printf("*"); printf("\n");
}
```

Наконец, текст

```
starline();
```

— вызов этой функции.

Определение функции — это назначение имени выделяемому фрагменту программы, оно состоит из заголовка (в нашем случае `void starline()`) и тела (в нашем случае

```
{
    int i;
    for(i = 0; i < 70; i++) printf("*"); printf("\n");
}
```

), а вызов — это использование поименованного фрагмента, т. е. выполнение входящих в тело функции операторов.

После определения функции starline наша исходная программа про строки из звездочек стала короче и лучше читается.

29 Параметры и результат функции

Параметры функции позволяют предоставлять поименованному фрагменту программы некоторую дополнительную информацию. Например, если в нашей программе часто используется операция возведения целого числа в квадрат, мы вполне можем захотеть выделить ее в отдельную функцию. Однако, чтобы возвести число в квадрат, нужно, во-первых, знать это число, и, во-вторых, иметь возможность возвратить результат удобным для его дальнейшего использования образом. Механизм передачи параметров решает первую из этих задач, а оператор return для возврата результата — вторую.

В итоге для возведения числа в квадрат мы можем написать определение функции

```
int sqr(int x)
{
    return x*x;
}
```

, вырабатывающей в качестве результата, т. е. возвращающей, квадрат своего параметра (в нашем случае параметр в определении функции обозначается x, и является целым числом; описание параметров похоже на определение переменных, с той разницей, что при описании соседних параметров с одним и тем же типом этот тип должен быть указан при имени каждого параметра отдельно; переменные, которые в определении функции обозначают ее параметры, называются формальными параметрами). Результат (целое число типа int, потому что заголовок функции начинается с указания типа возвращаемого значения, и в нашем случае указано int) возвращается оператором return, который указывает, что надо возвращать, и прекращает выполнение функции. Если функция возвращает какой-то результат, в ней всегда должен быть оператор return, и любой вариант выполнения тела такой функции должен завершаться оператором return.

Результат функции затем можно использовать в других выражениях, например, так:

```
printf("%d\n", sqr(25));
```

Такой фрагмент программы выведет на экран число 625. Здесь значение параметра, указанное в вызове, в нашем случае 25, называется фактическим параметром. При вызове функции с параметрами перед выполнением тела функции формальные параметры инициализируются значениями фактических, причем соответствие между ними определяется порядковым номером параметра, т. е. первый формальный параметр инициализируется значением первого фактического параметра, второй — второго, и т. д. Совпадение имен некоторых фактических параметров с именами формальных не имеет здесь вовсе никакого значения.

Язык C позволяет и игнорировать результат функции вовсе, например, оператор sqr(25); вполне законен, но бессмыслен, так как ничего не делает.

Продолжая пример из предыдущего файла про строки из звездочек, мы можем задаться вопросом, что делать, если первая строка должна содержать 50 символов *, вторая — 60 и третья — 70? Здесь нам на выручку приходит механизм передачи параметров:

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

void starline(int n)
{
```

```
    int i;
    for(i = 0; i < n; i++) printf("*"); printf("\n");
}

int main()
{
    starline(50);
    printf("Title: Program 1\n");
    starline(60);
    printf("The end\n");
    starline(70);
    return EXIT_SUCCESS;
}
```

Здесь n — формальный параметр функции starline, а 50, 60 и 70 — фактические параметры трех разных ее вызовов.

Если у функции должно быть более одного параметра, то нужно указывать тип у каждого параметра отдельно, даже если типы соседних параметров совпадают. Например, функция, вычисляющая определитель квадратной матрицы второго порядка с элементами a, b, c и d, вполне может выглядеть так:

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

double det(double a, double b, double c, double d)
{
    return a*d - b*c;
}
```

Задача.

Переделайте программу про строки из звездочек так, чтобы первая строка состояла из символов =, вторая — минусов и третья — звездочек.

Функция prime в следующей программе проверяет свой целый параметр на простоту и вырабатывает логическое значение, а вся программа, используя эту функцию, выводит все простые числа от 1 до 1000:

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

bool prime(int n)
{
    int i;
    if(n==1) return false;
    if(n<4)  return true;
    for(i = 2; i*i<=n; i++)
        if(n%i == 0) return false;
    return true;
}

int main()
{
    int i;
    for(i = 1; i<=1000; i++)
        if(prime(i)) printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

В этой программе мы видим две локальные переменные с именем i, определенные в двух разных функциях. В C это вполне допустимо, и между этими переменными нет ничего общего, кроме имени.

Задачи.

1*. Написать функцию с одним целым параметром, изображающую в текстовом окне терминала окружность указанного радиуса при помощи символов *.

2. Написать функцию, принимающую координаты вершин треугольника на плоскости и возвращающую его площадь.

3. Пусть дано число n. Будем заменять его на число $n/2$, если n четно, и на $3n+1$, если n нечетно. Существует гипотеза (не доказанная и не опровергнутая до сих пор), что с какого бы числа мы ни начали, в результате цепочки таких замен мы всегда можем прийти к числу 1. Ваша функция должна принимать n в качестве параметра и возвращать число замен, приводящих к 1.

4*. Написать функцию, вычисляющую функцию Эйлера $\varphi(n)$. Если n раскладывается на простые множители как $n = p_1^{k_1} p_2^{k_2} \dots p_l^{k_l}$, то

$$\varphi(n) = n\left(1 - \frac{1}{p_1}\right)\dots\left(1 - \frac{1}{p_l}\right).$$

Оказывается, передавать в функцию в качестве параметров можно не только числа, но и указатели, и массивы. Возвращать в качестве результатов массивы, правда, нельзя, но указатели можно, и можно возвратить указатель на первый элемент (динамического или глобального!) массива, если нужно вернуть массив.

30 Передача параметров по значению и по ссылке

До сих пор мы передавали в функции значения фактических параметров, и в теле функции формальные параметры были копией фактических. В частности, это означало, что 1) любое присваивание формальному параметру какого-либо значения никак не отражалось на фактическом параметре и 2) в качестве фактического параметра могло выступать любое выражение, возвращающее результат подходящего типа, в частности, явно указанная константа.

Однако, иногда возникает необходимость, чтобы в результате работы функции фактическому параметру присваивалось некоторое значение. Например, такая необходимость возникает, когда поименованный фрагмент кода вырабатывает не один результат, а несколько. Тогда один из результатов может возвращаться обычным для функции образом при помощи оператора return, а все остальные — через присваивания этих результатов фактическим параметрам. Также, бывает нужно, чтобы функция меняла значения своих фактических параметров (в этом случае, естественно, фактическими параметрами должны быть только такие выражения, которым можно что-либо присвоить, например, имена переменных).

Оказывается, эта задача вполне решаема. Чтобы, используя формальный параметр функции, можно было присваивать нечто внешним по отношению к функции переменным, достаточно передавать в функцию указатели на такие переменные. С таким явлением мы уже встречались, когда вводили значения переменных с помощью функции scanf. Знак &, стоящий перед именами вводимых переменных, был ничем иным, как операцией взятия адреса, и функция scanf просто получала в качестве параметров указатели на те переменные, куда надо было записать введенные значения.

Простейший пример происходящего представлен следующей программой:

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

void sp(int z, int *t)
{
    z = 7;
    *t = 5;
}

int main()
{
    int x = 1, y = 2;
    sp(x, &y);
    printf("x=%d\n", x); // 1
    printf("y=%d\n", y); // 5
    sp(1, y);  // OK
    sp(x, &2); // Ошибка!
    return EXIT_SUCCESS;
}
```

Задачи.

1. Написать функцию, возвращающую квадрат, куб и четвертую степень своего параметра, являющегося вещественным числом типа double.

2. Написать функцию, переставляющую значения своих трех вещественных параметров по циклу.

3. Написать функцию с тремя вещественными параметрами, заменяющую значение каждого параметра на среднее арифметическое значений двух других параметров.

4*. Изменить функцию для вычисления НОД так, чтобы она возвращала также коэффициенты его линейного разложения, т. е. если в качестве параметров заданы a и b и НОД$(a, b) = c$, то функция должна находить и возвращать такие числа k и l, что $ka + lb = c$.

Как уже говорилось в предыдущих разделах, в функцию можно передать массив. Однако, реально это означает всего лишь передачу в эту функцию указателя на первый элемент этого массива. Отсюда можно сделать следующие выводы:

1) Массив всегда передается по ссылке, т. е. любые манипуляции с его элементами внутри функции отражаются на фактическом параметре, т. е. том массиве, который был передан.

2) Можно передавать в одну и ту же функцию массивы из одного и того же типа элементов, но разной длины, так что при передаче одномерного массива можно не писать число его элементов. Обычно, это число передается как отдельный целый параметр, но могут быть и другие варианты, например, сигнальное значение (определенное значение элемента считается последним) и т. п. Какое-то соглашение, однако, должно быть, потому что вызываемая функция должна знать способ определения длины массива.

Можно передавать в функции и многомерные массивы, однако в этом случае все размеры кроме первого должны быть явно указаны, и передача, скажем, двумерного массива означает не передачу указателя на указатель, как можно было бы подумать, а передачу указателя на массив, представляющий собой одну строку передаваемой матрицы.

Также важно сделать замечание и о том, что писать в вызове при передаче массива в функцию. Если в функцию передается одномерный массив, и фактический параметр — тоже одномерный массив, в вызове указывается только имя фактического параметра, без каких бы то ни было индексов:

```
double V[100];
void f(double A[], int l); // прототип функции
f(V, 100); // вызов
```

Аналогично, если в функцию должен передаваться многомерный массив, и фактический параметр — весь массив той же размерности, в вызове функции указывается только имя массива-параметра. А вот если фактическим параметром должна быть часть какого-либо массива, то нужно и писать в вызове выражение, выдающее эту часть. Например, если формальный параметр — одномерный массив, а фактический — строка в матрице, то в вызове надо писать выражение, вырезающее нужную нам строку из матрицы:

```
double V[10][20]; // матрица
void f(double A[], int l); // прототип функции
f(V[1], 20); // вызов со второй строкой
```

31 Объявление и определение функции

С определениями функций мы уже познакомились. Оказывается, в С есть и еще один вид описания функций — объявление (как и для глобальных переменных). Кратко это уже обсуждалось в параграфе про раздельную компиляцию, теперь пришло время обсудить это подробнее.

Объявление функции (оно же иногда называется прототипом функции) состоит из заголовка функции и точки с запятой вместо ее тела. При этом, имена формальных параметров вообще могут быть опущены. Объявление функции говорит компилятору о том, что такая функция в программе есть и определяется где-то в другом месте (в другом файле или в том же файле, но позже).

Объявление функции нужно компилятору для того, чтобы правильно скомпилировать вызов этой функции. Собственно, для правильной компиляции определения какой-либо функции компилятору нужны определения или объявления всех вызываемых в ее теле функций. Обычно объявления функций группируются по смыслу или принадлежности определенной библиотеке в заголовочные файлы, которые затем включаются во все файлы реализации, в которых используются эти функции (вот зачем нужны заголовочные файлы!).

Когда компилятор, а затем ассемблер строят объектный модуль, в нем указываются имена тех функций, которые были объявлены, но не определены в исходном файле. На этапе компоновки результаты компиляции тел таких функций должны быть найдены в других объектных модулях или библиотеках и их адреса в окончательной программе подставлены в те места данного объектного модуля, где производился вызов такой функции. Если хотя бы одна из таких функций не будет найдена, компоновщик выдаст сообщение об ошибке, и окончательная программа не будет создана.

Такие правила обслуживают в основном нужды раздельной компиляции, позволяя разместить определения всех функций, из которых состоит программа, в нескольких исходных файлах, а в тех файлах, где некоторая функция не определена, но используется, поместив ее объявление (как правило, во включаемом заголовочном файле). Однако, забегая вперед, скажем, что нужда в прототипах функций бывает и без всякой раздельной компиляции, например, когда есть две функции, каждая из которых вызывает другую.

Задачи.

1. Написать функцию f с одним вещественным параметром x и вещественным результатом, вычисляющую 30*cos(x/4), и поместить ее в отдельный файл реализации. Написать функцию graph без параметров и результата, строящую график этой функции (ось X расположить сверху вниз, ось Y — слева направо, начало координат — в центре экрана), и поместить ее в другой файл. Написать функцию main, вызывающую функцию построения графика, и поместить ее в третий файл. Написать также все необходимые заголовочные файлы и включить их в файлы реализации директивой `#include`. Включить все файлы реализации и заголовочные файлы в проект, добиться того, чтобы он собирался и работал. После этого изменить функцию f так, чтобы теперь она вычисляла 10*tg(x/10), и пересобрать проект. Какие файлы будут компилироваться в этом случае?

2. Написать функцию f с одним вещественным параметром x и вещественным результатом, вычисляющую $e^x - 5\sin x$, и поместить ее в отдельный файл реализации. Написать функцию root, принимающую в качестве параметра два числа a и b, и возвращающую корень функции f на отрезке $[a, b]$ (корень находится методом половинного деления; если на концах этого отрезка значения функции имеют одинаковые знаки, выдается сообщение об ошибке), и поместить ее в другой файл. Написать функцию main, вычисляющую корень функции f на отрезке $[0, 1]$ при помощи функции root, и поместить ее в третий файл. Написать также все необходимые заголовочные файлы и включить их в файлы реализации директивой `#include`. Включить все файлы реализации и заголовочные файлы в проект, добиться того, чтобы он собирался и работал (проверить правильность ответа при помощи wolframalpha). После этого изменить функцию f так, чтобы теперь она вычисляла $\ln x - \sin x$, а в функции main — отрезок на $[1/2, \pi]$, и пересобрать проект. Какие файлы будут компилироваться в этом случае?

32 Статические переменные

Как мы уже видели, внутри функций могут быть определены свои так называемые автоматические локальные переменные, которыми можно пользоваться только внутри тех функций, в которых они определены. Такие переменные существуют (т. е. под них выделяется память) только пока содержащая их функция активна, т. е. ее тело выполняется в данный момент. После завершения выполнения функции все ее автоматические локальные переменные исчезают, и при повторном вызове они создаются заново, вполне возможно, в другом месте памяти. Таким образом их значения не сохраняются между разными вызовами содержащей их функции.

Однако, в C есть возможность объявить такую локальную (т. е. видимую только в этой функции) переменную, значение которой сохраняется между разными вызовами содержащей ее функции.

Такая переменная оформляется как «статическая», при помощи добавления в ее декларацию ключевого слова static до имени типа. Следующий пример показывает, как при помощи такой переменной написать функцию, при первом вызове печатающую строку «Hello!», а при всех последующих вызовах — «Hello again!»:

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

void greeting()
{
    static bool first = true;
    if(first)
    {
        printf("Hello!\n");
        first = false;
    }
    else
        printf("Hello again!\n");
}

int main()
{
    int i;
    for(i = 1; i<=10; i++)
        greeting();
    return EXIT_SUCCESS;
}
```

Здесь остается только добавить, что инициализация статической переменной выполняется всего один раз, когда до ее декларации в первый раз доходит процесс выполнения программы. Во все следующие разы эта инициализация просто игнорируется.

Задачи.

1. Написать функцию без параметров, возвращающую номер своего вызова (первый вызов возвращает 1, второй — 2, третий — 3, и т. д.).

2. Написать функцию с одним целым параметром, управляющую целочисленным значением следующим образом: сначала значение равно 0, и при каждом вызове устанавливается новое значение (равное значению параметра), и возвращается старое.

3. Написать функцию с целым параметром m, результат которой равен сумме параметров трех последних вызовов этой функции.

33 Возврат указателя в качестве результата

Оказывается, в C функция может не только получать указатели в качестве параметров, но и возвращать их в качестве результатов. Иногда это бывает очень удобно, однако может служить источником ошибок. В частности, никогда нельзя возвращать указатель на автоматическую локальную переменную, потому что она исчезает по выходе из функции, и результат функции указывает в этом случае на исчезнувшую переменную. Присваивание этой переменной может привести в лучшем случае к аварийному завершению работы программы, а в худшем — к неверному результату. Возвращать можно только указатели на глобальные, динамические или статические переменные, или те указатели, которые являются формальными параметрами.

Следующий пример показывает, как с помощью этой возможности моделируется массив из 5 целых элементов (доступ к переменной по номеру, который может быть результатом выражения; элементы хранятся в разных глобальных переменных):

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

int x1, x2, x3, x4, x5; // элементы

int *access(int n)
{
    switch(n)
    {
    case 1: return &x1;
    case 2: return &x2;
    case 3: return &x3;
    case 4: return &x4;
    case 5: return &x5;
    default: exit(EXIT_FAILURE); // закончить с ошибкой
    }
}

int main()
{
    int i;
    printf("%d\n", *access(2)); // печатаем x2
    *access(1) = 2;         // x1 = 2
    for(i = 1; i <= 5; i++)
        *access(i) = i*10;  // заполнить в цикле
    // ни xi ни x i работать не будет
    return EXIT_SUCCESS;
}
```

Задачи.

1. Написать функцию без параметров, имеющую 3 статические целые переменные и предоставляющую доступ (т. е. возвращающую адрес) той из них, которая имеет наименьшее значение, т. е. выражение вида printf("%d *f()); должно печатать наименьшее значение из этих трех переменных, и выражение *f()=... должно записывать значение выражения справа от = в ту из статических переменных, значение которой наименьшее (если таких несколько, годится любая из них; после такого присваивания, вообще говоря, значение этой переменной может и перестать быть наименьшим).

2. Описать три целые глобальные переменные x, y, z и функцию f без параметров, предоставляющую доступ к каждой из глобальных переменных по циклу, т. е. первый вызов должен возвращать адрес x, второй — y, третий — z, четвертый — опять x, и т. д.

3*. Написать функцию f, которая хранит положительное вещественное число (изначально 1) и обеспечивает доступ к его степеням по показателю степени, т. е. мы можем, в частности, присваивать значение как самому числу, так и любой его степени (разумеется, влезающей в разрядную сетку double), и функция должна автоматически поддерживать согласованность хранящихся в ней данных. Например, в результате выполнения фрагмента программы

```
*f(2)=25;
printf("%f", *f(3));
```

должно быть напечатано 125.

34 Динамические массивы, разреженные матрицы.

Обычные (автоматические) локальные переменные возникают в момент начала работы подпрограммы, живут, пока она работает, и исчезают по окончании ее работы. Однако иногда требуется в подпрограмме завести переменную, которая сохранилась бы и при выходе из подпрограммы. Если число таких переменных известно заранее (в момент компиляции программы), можно воспользоваться статическими переменными. Часто это не так, и в этом случае можно воспользоваться механизмом динамических переменных.

Динамическая переменная хранится в особой области памяти, называемой кучей. Особенность этой области памяти в том, что, в отличие от стека, где хранятся обычные локальные переменные и где уничтожать их можно только в порядке, обратном порядку их создания, в куче можно уничтожать переменные и освобождать занимаемую ими память в любом порядке. Платой за такое удобство является то, что выделение и освобождение памяти в куче происходит намного медленнее, чем в стеке.

Для того, чтобы завести динамическую переменную, нужно выделить под нее память, вызвав функцию malloc с указанием размера выделяемой под переменную памяти (в байтах). Если мы выделяем память под одиночную переменную, можно написать p=(тип*)malloc(sizeof(тип)) Это выражение заводит динамическую переменную указанного типа и возвращает указатель на нее. Если же нам нужен массив, то надо написать p=(тип*)malloc(число_элементов*sizeof(тип)) После такого присваивания указатель p будет указывать на вновь созданную динамическую переменную требуемого типа.

Динамические переменные не имеют имен, и доступ к ним возможен только через указатели на них. Если в результате ошибок в программе возникают динамические переменные, на которые никакой указатель не указывает (т. е. до которых невозможно добраться), это явление называется утечкой памяти, а сами недоступные динамические переменные называются мусором.

Еще одно отличие динамических переменных от автоматических локальных состоит в том, что если они перестали использоваться, их надо удалять вручную вызовом функции free, параметром которой является указатель на удаляемую переменную или первый элемент удаляемого массива. При этом размер удаляемого фрагмента памяти указывать не надо, он определяется автоматически по указателю на освобождаемую область памяти.

Динамический массив хорош тем, что число его элементов может быть любым выражением, возвращающим положительное целое число.

Здесь надо еще заметить, что в C нельзя менять размер массива на ходу. Если нам нужно расширить наш динамический массив, нужно явно завести массив большего размера, переписать в него элементы старого массива и наконец так же явно уничтожить старый массив.

В C есть возможность организовать не только одномерные динамические массивы, но и многомерные. Для организации двумерного динамического массива прямоугольной формы (матрица из m строк и n столбцов) нужно сначала завести динамический массив указателей на строки:

```
int **p = (int**)malloc(m*sizeof(int *));
```

Далее, нужно завести сами строки:

```
for(int k = 0; k < m; k++)
    p[k] = (int*)malloc(n*sizeof(int));
```

Доступ к такому массиву, как уже говорилось, будет иметь традиционный вид: p[i][j]. Для уничтожения такого массива нужно проделать следующее:

```
for(int k = 0; k < m; k++)
    free(p[k]);
free(p);
```

Этот подход позволяет иметь даже строки разной длины, что удобно для хранения так называемых разреженных матриц (т. е. таких, среди элементов которых много нулей), например, треугольных матриц.

Пример.

В качестве примера рассмотрим набор функций, позволяющий заводить, уничтожать и использовать элементы верхне-треугольной матрицы, порядок которой определяется только во время выполнения программы.

Традиционный подход к решению этой задачи, т. е. двумерный динамический массив n×n, нас не устраивает, поскольку хранит примерно в два раза больше элементов, чем надо (если примерно про половину элементов мы точно знаем, что они равны 0, зачем их хранить?).

Итак, сначала опишем удобную функцию для выделения памяти под верхнетреугольную матрицу. Эта функция принимает в качестве параметра порядок матрицы и возвращает указатель на первый элемент массива указателей на строки.

```
int **create(int n)
{
    int **p = (int**)malloc(n*sizeof(int *));
    for(int k = 0; k < n; k++)
        p[k] = (int*)malloc((n-k)*sizeof(int));
    return p;
}
```

Здесь сначала выделяется память под массив указателей на строки, а затем для каждого указателя на строку выделяется память под те элементы строки, которые надо хранить. При этом количество хранимых элементов каждой строки зависит от номера этой строки. Таким образом, мы выделяем память только под те элементы, которые могут отличаться от нуля.

Затем, мы рассмотрим функцию access, обеспечивающую доступ к элементам нашей матрицы по индексам строки и столбца. Она будет принимать в качестве параметров матрицу (указатель на первый элемент массива указателей на строки), индексы строки и столбца и порядок матрицы. Возвращает эта функция ссылку на (т. е. синоним) соответствующего элемента матрицы.

```
int *access(int **M, int i, int j, int n)
{
    static int dummy;
    dummy = 0;
    if(i<0 || i>=n || j<0 || j>=n)
        exit(1); // аварийная остановка программы
    if(i>j) return &dummy; // заведомо нулевой элемент
    return &M[i][j-i];
}
```

Данная функция проверяет, что индексы необходимого элемента попадают в допустимые пределы, и если это не так, то вызывает аварийное завершение программы. Далее, проверяется, что индексы попадают на заведомо нулевой элемент, и если это так, то возвращается статическая переменная данной функции, хранящая всегда нулевое значение. Нулевое значение присваивается этой переменной не в инициализации, потому что инициализация для статических переменных выполняется всего один раз, а эту переменную нужно обнулять каждый раз на тот случай, если ей было что-то присвоено, что вполне возможно. И наконец, если индексы попадают в нужный диапазон, т. е. на хранимый элемент, мы возвращаем его. Пересчет индексов (j-i) нужен потому, что хранимая часть строки матрицы начинается не с нулевого столбца, а с i-го, так что для преобразования индекса столбца в исходной матрице в индекс этого элемента в хранимой части строки нужно вычесть из него индекс столбца первого хранимого элемента в исходной матрице.

Задача.

Написать функцию create, заводящую блочно-диагональную матрицу заданного порядка (он определяется во время выполнения программы; порядок блоков равен 10; хранить в памяти, естественно, нужно только возможно ненулевые элементы) и возвращающую int** (указатель на первый элемент массива указателей на строки). Написать также функцию access, выдающую ссылку на заданный элемент этой матрицы по указателю на такую матрицу (типа int**) и номерам строки и столбца (номера в исходной квадратной матрице; если соответствующий элемент лежит в одном из блоков, нужно выдать ссылку на ту переменную в составе двумерного динамического массива, в которой этот элемент хранится, иначе выдается ссылка на локальную статическую переменную, которая обнуляется при каждом вызове этой функции). Также написать функцию destroy, освобождающую память, выделенную функцией create.

35 Рекурсия

Многие языки программирования, и C — не исключение, поддерживают рекурсию, т. е. функции, вызывающие сами себя, непосредственно или через другие функции. В этом случае, обычно, речь идет о функциях с параметрами, и все возможные наборы значений параметров делятся на два класса:

а) Базовые задачи, которые могут быть решены непосредственно (без вызовов той же функции).

б) Задачи общего случая, решение которых сводится к решению той же задачи, но с более простыми (как правило, это означает меньшими по величине) значениями параметров.

Чтобы такая схема работала, необходимо, чтобы для любого возможного набора параметров цепочка сведений одной задачи к набору других всегда была конечной, т. е. всегда приводила за конечное число сведений к набору базовых задач.

Рекурсивное решение задачи по программированию напоминает прием доказательства теорем, называющийся математической индукцией, и именно этот прием используется обычно для доказательства того, что рекурсивная функция работает правильно.

Рекурсия часто приводит к упрощению текстов функций. Она позволяет почти без изменений переписывать на соответствующий язык программирования (в нашем случае — C) рекуррентные определения.

Например, факториал целого числа может быть определен рекуррентно при помощи формулы

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0. \end{cases}$$

Пользуясь рекурсией, это определение факториала можно без труда почти дословно переписать на C, получив следующую функцию:

```
int fact(int n)
{
   if(n==0)
      return 1;
   return n*fact(n-1);
}
```

Также можно поступить и с рекуррентным определением чисел Фибоначчи.

```
int fib(int n)
{
   if(n<2)
      return 1;
   return fib(n-1)+fib(n-2);
}
```

Однако, здесь имеется один подвох. Если вычисление n-го числа Фибоначчи при помощи цикла требует количества действий, пропорционального номеру искомого числа в последовательности Фибоначчи, то рекурсивный алгоритм работает гораздо дольше, поскольку здесь почти каждый вызов порождает еще два других, в результате чего общее число вызовов растет экспоненциально в зависимости от значения параметра функции.

Так что рекурсивное решение задачи, как правило, гораздо проще написать, чем нерекурсивное, но оно вполне может быть чудовищно неэффективно.

Кроме того, здесь есть еще один подводный камень: при использовании рекурсии максимальная глубина вызовов функций обычно резко увеличивается и растет с ростом значений параметров задачи. Это увеличивает максимально необходимый объем стека (стек — это специальная область памяти программы, где хранятся локальные переменные функций, а также адреса возврата из них).

Также в C, как и во многих других языках программирования, возможна множественная рекурсия, когда имеется несколько функций, и они могут вызывать сами себя не только непосредственно, но и через вызовы других функций.

Поскольку каждая функция, которая вызывается в программе, должна быть объявлена прежде своего использования, возникает проблема: если есть две функции, каждая из которых вызывает другую, какая из них должна быть определена первой? Ответ дается введением в язык так называемых прототипов, когда указывается только заголовок функции и «;» вместо тела (и этого достаточно для правильной компиляции вызова), а тело описывается потом, когда будут указаны таким же образом (или определены полностью с указанием тел) все функции, вызываемые данной функцией. В качестве примера приведем следующую программу (правда, в отношении ее эффективности справедливы все замечания, касающиеся функции для вычисления чисел Фибоначчи):

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

int a(int); // прототип

int b(int n)
```

```
{
    if(n==1)
        return 1;
    return 7*a(n-1)+2*b(n-1);
}

int a(int n)
{
    if(n==1)
        return 2;
    return 3*a(n-1)-5*b(n-1);
}

int main()
{
    printf("a(2)=%d\n", a(2));
    printf("b(2)=%d\n", b(2));
    return EXIT_SUCCESS;
}
```

Как видно из этого примера, в прототипах можно не писать имена формальных параметров, а только их типы.

Задачи.

1. Написать функцию, печатающую цифры десятичного представления своего неотрицательного целого параметра, разделяя их пробелами: а) в обычном порядке; б) в обратном порядке (то и другое — без циклов).

2. Написать функцию, вычисляющую биномиальный коэффициент C_n^k, без использования операторов цикла.

3. (Ханойские башни) Имеется 3 стержня и набор колец разного размера, надетых на первый стержень так, чтобы больший диск всегда находился под меньшим. За один ход можно снять верхний диск с одного стержня и надеть его на другой, но запрещено класть больший диск на меньший. Требуется написать функцию, печатающую последовательность переноса дисков с одного стержня на другой (печатается, с какого на какой), в результате которого все диски переместятся с первого стержня на третий (при этом второй стержень играет роль вспомогательного).

4. Написать функцию, вычисляющую n^k за меньше, чем $2\log_2 k$ умножений.

5. Написать функцию, вычисляющую число представлений своего натурального параметра в виде суммы натуральных слагаемых, порядок которых существен, т. е. суммы 2+1 и 1+2 считаются разными (без циклов).

6. Написать рекурсивную функцию с целым параметром m, вычисляющую m-й член последовательности, начинающейся с 3, в которой каждый следующий член на 1 меньше произведения всех предыдущих (без циклов).

7*. То же, но порядок слагаемых несуществен, т. е. суммы 2+1 и 1+2 считаются одинаковыми (без циклов).

8*. Написать функцию, проверяющую правильность скобочной структуры, без циклов:

а) допускаются только символы «(», «)» и «.» (последний означает конец строки)

б) допускаются три вида скобок («()», «[]» и «{}»). Конец строки — «.»

в) вместо трехместной операции «что-то(что-то)что-то» используется четырехместная «что-то(что-то,что-то)что-то»; конец строки — как всегда «.»

9**. Написать функцию, принимающую в качестве параметров целое число $n \geq 1$ и возвращающую f_n, если последовательность f_n определена соотношениями $\sum_{i=1}^{n} f_i f_{n+1-i} = n^2$ при $n \geq 1$, $f_1 > 0$ (без циклов).

36 Функции, число и типы параметров которых заранее неизвестны

Оказывается, в C возможны объявления и определения такого рода функций. Более того, с примерами таких функций мы уже встречались — функции вывода printf и ввода scanf как раз и являются самыми яркими представителями этого семейства. Правда, пользоваться этой возможностью нужно как можно реже, поскольку в этом случае проверки типов на этапе компиляции невозможны, и передача неправильных параметров может быть источником весьма нетривиальных ошибок. Например, в этом случае 0 всегда передается как целое число, а если мы хотим передать такое же по значению вещественное число, нужно писать явно 0.0 (поскольку заранее неизвестно, что мы хотим получить, преобразования типов, даже стандартные, в этом случае не выполняются).

Объявляется такая функция так:

```
тип_результата имя_функции(список_параметров,...);
```

При этом сначала идет некоторое количество явно указанных параметров, типы которых известны заранее (не меньше одного такого параметра, их мы будем называть основными), а дальше в вызове могут использоваться параметры любых типов в любом количестве (мы будем называть такие параметры дополнительными).

Для написания тела такой функции нужно сначала подключить соответствующий заголовочный файл:

```
#include <stdarg.h>
```

В самом теле функции сначала нужно объявить переменную типа va_list:

```
va_list l;
```

Имя переменной может быть любым. Затем, перед началом извлечения дополнительных параметров, нужно написать следующий оператор:

```
va_start(l, x);
```

Здесь вместо l должно стоять имя переменной типа va_list (если оно у вас отличается), а вместо x — имя последнего основного параметра. После этого можно начать извлекать дополнительные параметры. Очередной дополнительный параметр извлекается вызовом

```
va_arg(l, тип)
```

Здесь тип — тип того параметра, который вы собираетесь извлечь (нужно его знать; обычно этот тип определяется типами и значениями основных и уже извлеченных к этому моменту дополнительных параметров). Надо заметить, что конструкция

```
va_arg(l, double)*va_arg(l, double)
```

не вернет вам квадрат очередного параметра типа double, а, скорее, вернет произведение двух очередных параметров типа double. Вместо этого, если нужно использовать один и тот же считанный параметр несколько раз, его нужно сохранить в локальной переменной.

Наконец, когда все дополнительные параметры считаны, перед выходом из функции (очевидно, до оператора return!) нужно вызвать

```
va_end(l);
```

Без этого программа, скорее всего, не будет работать, поскольку та память, через которую передаются параметры, будет сильно запорчена.

Пример. В качестве примера напишем функцию, принимающую целое число n (основной параметр) и столько дополнительных вещественных параметров, чему равно n, и вычисляющую среднее арифметическое своих вещественных параметров.

```
double average(int n, ...)
{
    va_list l;
    double sum = 0;
    va_start(l, n);
    for(int i = 0; i<n; i++)
        sum += va_arg(l, double);
    va_end(l);
    return sum/n;
}
```

Задача.

Написать функцию, принимающую в качестве параметров несколько вещественных чисел (признаком конца является нулевое значение) и вычисляющую среднее арифметическое своих параметров, кроме, разумеется, последнего нуля.

37 Сортировка

Под сортировкой в программировании понимается упорядочение элементов последовательности, например, массива. Для этого придумано несколько разных алгоритмов, каждый из которых имеет необходимый для работы объем дополнительной памяти и число требующихся операций как функции от числа элементов исходного массива.

Кроме того, число требуемых операций можно измерять по-разному. Можно интересоваться числом операций в среднем, по всем возможным перестановкам начальных элементов массива, а можно вычислять число операций в худшем случае, т. е. для того начального расположения элементов массива, для которого время работы алгоритма будет наибольшим.

По времени работы алгоритмы сортировки обычно делятся на простейшие (порядка CN^2 операций, где C — ни от чего не зависящий множитель) и быстрые (порядка $CN \log N$ операций).

Иногда время работы алгоритма может зависеть не только от начального расположения элементов, но и от их величины.

Задачи

1. Реализовать алгоритм сортировки пузырьком, состоящий в следующем. Делается n (размер массива) проходов, на каждом из которых последовательно просматриваются пары соседних элементов, и если их порядок неправилен, они меняются местами. Предложить и реализовать улучшения к этому алгоритму.

2. а) Реализовать алгоритм сортировки отбором, состоящий в следующем: в исходном массиве ищется наименьший элемент и он меняется местами с первым элементом массива. Затем процесс повторяется для подмассива, начинающегося со второго элемента, затем — с третьего, и т. д. Когда последние два элемента займут свои места, массив окажется отсортированным. б) Реализовать алгоритм п. а) в стиле «сети сортировки», т. е. как последовательность действий вида (посмотреть на пару элементов, и если их порядок неправилен, поменять их местами), причем последовательность пар элементов, на которые мы смотрим, должна зависеть только от количества элементов, а не от их начального порядка или величины.

3. а) Реализовать алгоритм сортировки вставками, состоящий в следующем. Последовательно просматривается сортируемый массив, и каждый очередной элемент ставится на свое место в уже отсортированной части массива (от начала до текущего элемента). б) Реализовать алгоритм п. а) в стиле «сети сортировки» (см. п. б) предыдущей задачи).

4. Реализовать алгоритм поразрядной сортировки, состоящий в следующем. Для сортировки массива из n неотрицательных целых чисел заводится дополнительный двумерный массив размера $10 \times n$, и сортировка состоит из последовательно чередующихся распределительного и собирательного проходов. Во время распределительного прохода последовательно просматривается исходный массив, и очередное число записывается в ту строку двумерного массива, индекс которой равен значению соответствующего разряда этого числа (на первом распределительном проходе анализируется разряд единиц, на втором — десятков, на третьем — сотен и т. д.). На собирательном проходе в исходный массив записывается сначала содержимое нулевой строки двумерного массива, затем — первой, и т. д. Пары из распределительного и собирательного проходов повторяются до тех пор, пока не будут исчерпаны разряды максимального элемента массива.

5. Реализовать алгоритм быстрой сортировки, состоящий в следующем. Берется последний элемент массива (его значение назовем x), и элементы массива переупорядочиваются так, чтобы элемент x занял свое окончательное место в отсортированном массиве, т. е. все элементы, меньшие x, стояли бы слева от x, а все большие — справа. После этого элементы слева от x сортируются отдельно при помощи того же алгоритма, и то же самое происходит с элементами, стоящими справа. Предложить и затем реализовать улучшения к этому алгоритму.

6. Реализовать алгоритм сортировки слиянием, состоящий в следующем. Сначала тем же алгоритмом сортируются две половины исходного массива (два рекурсивных вызова), затем заводится еще один вспомогательный массив той же длины, что и исходный, и в него переписываются элементы двух отсортированных частей исходного массива так, чтобы результат тоже был упорядоченным; наконец, упорядоченные элементы переписываются обратно в исходный массив.

7. Реализовать алгоритм пирамидальной сортировки, состоящий в следующем. Делается два прохода по исходному массиву. На первом проходе последовательно просматриваются элементы массива и по очереди добавляются в пирамиду. На втором проходе первый элемент пирамиды (самый большой в ней) меняется местами с очередным элементом в конце массива, длина пирамиды уменьшается на 1, и пирамида подправляется; после второго прохода массив оказывается отсортированным.

Пирамида — это набор элементов, занимающих начальный отрезок массива и удовлетворяющих условию: каждый элемент пирамиды с индексом i не меньше элементов с индексами $2i + 1$ и $2i + 2$ (здесь важно, что индексы массива начинаются с 0).

Когда мы хотим добавить в пирамиду элемент с индексом j (к этому времени пирамида состоит из элементов с индексами от 0 до $j - 1$), мы смотрим элемент с индексом $(j - 1)/2$ (в смысле C++), и если он меньше добавлямого, меняем их местами. Далее, мы сравниваем элементы с индексами $(j - 1)/2$ и $((j - 1)/2 - 1)/2$, и меняем их местами в случае необходимости, чтобы больший элемент был ближе к началу массива, и т. д. до тех пор, пока либо не встретится больший добавляемого элемент, либо добавляемый элемент не окажется первым элементом массива.

При извлечении элемента из пирамиды мы просто меняем местами первый элемент, очевидно не меньший всех оставшихся в пирамиде, с последним элементом пирамиды, считая при этом, что пирамида сократилась на один элемент, и подправляем ее так, чтобы сохранилось свойство пирамиды. Для этого элемент с индексом 0 сравнивается с элементами с индексами 1 и 2, если среди них есть больший его, наибольший из них меняется местами с первым. Затем, если мы поменяли элемент с индексом i, мы сравниваем его новое значение с элементами с индексами $2i + 1$ и $2i + 2$, и если оба они не больше его, процесс прекращается, а если среди них найдется больший, наибольший из них меняется с ним местами, до тех пор, пока свойство пирамиды не восстановится.

В стандартной библиотеке языка C имеется готовая функция qsort, реализующая алгоритм быстрой сортировки.

38 Указатели на функции.

Как и для обычных переменных, скалярных типов или массивов, в C можно иметь указатели, выбирающие одну из функций с конкретной сигнатурой, т. е. списком типов параметров и типом возвращаемого значения. Типы таких переменных называются указателями на функции.

Указатель на функцию заводится следующим образом:

```
тип_значения_функции (*p)(список_формальных_параметров);
```

Для работы с этим указателем ему нужно присвоить какое-нибудь значение. Если у нас есть функция f с тем же типом результата и набором типов параметров, что и указатель, мы можем присвоить ему указатель на эту функцию:

```
p = &f;
```

Более того, знак & в этом выражении можно опустить и написать просто p = f;

После того, как указателю на функцию присвоено значение, можно вызвать ту функцию, на которую он указывает, написав (*p)(); или даже проще p();

Понятие указателя на функцию используется очень часто, здесь мы приведем только два варианта его использования:

1) Создание массивов функций. Очень часто имеется набор одинаковых по типу возвращаемого значения и типам параметров функций, из которых нужно вызвать одну по номеру. Можно, конечно, написать большой оператор выбора и в зависимости от номера вызвать одну из функций, но проще построить массив из указателей на функции и затем проиндексировать его номером той функции, которая нам нужна:

```
void f1();
void f2();
void f3();
void (*f[3])() = { f1, f2, f3 };
...
int main()
{
...
   f[i](); //вызов
...
}
```

2) Передача функций в качестве параметров другим функциям:

```
double integral(double a, double b, double (*f)(double))
{
    int n = 100;
    double sum = 0;
    for(int i = 1; i<=n; i++)
        sum += f(a+(i-0.5)*(b-a)/n);
    return sum*(b-a)/n;
}
...
int main()
{
...
   printf("%f\n", integral(0,1,sin)); //вызов
...
}
```

В этом примере участвует функция integral, которая приближенно вычисляет определенный интеграл от функции, переданной ей в качестве параметра. В теле функции main эта функция используется для вычисления $\int_0^1 \sin(x)dx$.

C даже позволяет вернуть указатель на функцию из другой функции в качестве результата, однако это в любом случае должна быть одна из функций, определенных в программе; создать новую функцию в процессе работы программы невозможно.

Задачи.

1. Написать функцию, принимающую в качестве параметров две функции f и g (каждая из них имеет два параметра типа char и возвращает char), вычисляющие операции $\triangle$ и $\bigtriangledown$ (т. е. если мы хотим вычислить, например, выражение

$a \triangle (b \triangledown c)$, мы должны в программе написать f(a, g(b, c))), а также массив M из 80 элементов типа char, и вычисляющую выражение (((M[0] $\triangle$ M[1]) $\triangledown$ M[2]) $\triangle$ M[3]) $\triangle$ ... $\triangle$ (((M[76] $\triangle$ M[77]) $\triangledown$ M[78]) $\triangle$ M[79]).

2. Написать функцию, принимающую в качестве параметров массив целых чисел переменной длины, длину этого массива и указатель на функцию типа `int(int,int)` и сортирующую этот массив относительно порядка, определяемого функцией-параметром (т. е. мы считаем, что $x > y$, если функция-параметр с параметрами x и y возвращает положительное число, $x = y$, если она возвращает ноль, и $x < y$, если она возвращает отрицательное число). В функции `main` завести массив из 15 целых чисел, ввести его элементы, и отсортировать его при помощи написанной вначале функции так, чтобы сначала шли четные числа по возрастанию, а потом — нечетные по убыванию.

39 Поиск

Задача поиска в массиве элемента с заданным значением встречается в программировании очень часто. Существует три основных способа решения этой задачи.

Первое из них, самое простое, называется линейным поиском. Суть этого метода состоит в последовательном просмотре элементов массива и сравнении каждого из них с образцом для поиска (тем значением, которое мы ищем). Одним из достоинств этого метода, кроме простоты, является универсальность. Он гарантированно работает с любым массивом. Однако, у этого метода имеются и недостатки. Самым серьезным из них является достаточно долгое, в сравнении с другими методами поиска, время работы. Например, чтобы убедиться, что такого значения в массиве нет, нужно просмотреть его целиком.

Второе решение называется двоичным поиском. Суть его в том, что, если мы ищем значение в упорядоченном массиве, за небольшое постоянное (не зависящее от размеров массива) число операций можно сократить область поиска в два раза (отсюда и название этого метода — двоичный поиск). В самом деле, сравнив значение среднего элемента массива с образцом поиска, мы можем сразу понять, в какой из половин массива лежит наше значение, если оно вообще в массиве присутствует. Далее, поступая таким же образом с каждым диапазоном элементов, который нам встречается, мы либо находим нужный нам элемент, либо убеждаемся в том, что его в массиве нет, причем число необходимых для поиска операций пропорционально двоичному логарифму от количества элементов в массиве. Достоинствами этого алгоритма являются недостатки предыдущего, и наоборот. Например, на больших массивах этот алгоритм работает гораздо быстрее, чем линейный поиск, но он сложнее реализуется и требует, чтобы элементы массива были упорядочены (отсортированы). Поэтому, если изменения содержимого массива происходят относительно редко (гораздо реже операций поиска), то имеет смысл отсортировать массив, а для того, чтобы найти нужный элемент, использовать алгоритм двоичного поиска.

Наконец, третье решение — алгоритм интерполяционного поиска. Основная идея этого метода состоит в попытке сразу угадать место расположения нужного нам элемента, а если это положение не угадано, то использовать идею двоичного поиска для сокращения диапазона элементов, среди которых производится поиск (смотреть, правда, нужно уже не средний элемент, а тот, который стоит в угаданной нами позиции). Этот метод, как и предыдущий, требует упорядоченности массива. Кроме того, он требует, для того, чтобы мы могли приблизительно угадать положение нужного нам элемента, чтобы расположение элементов в массиве было более-менее предсказуемо с точки зрения теории вероятности, и этот закон был достаточно простым. Если наши предположения относительно этого закона верны, то интерполяционный поиск (в среднем) работает еще гораздо быстрее, чем двоичный, за время, пропорциональное логарифму от логарифма количества элементов массива.

Задачи

1. Реализовать алгоритм линейного поиска как функцию с параметрами массив целых чисел, его длина и целое число n. Функция должна последовательно просматривать массив-параметр и возвращать индекс первого найденного элемента, равного n. Если указанный элемент не найден, функция возвращает -1.

2. Реализовать алгоритм двоичного поиска как функцию с параметрами массив целых чисел, его длина и целое число n. Функция должна анализировать средний элемент массива, сравнивая его с n и сужая диапазон поиска соответствующим образом, до тех пор, пока либо не будет найден элемент, равный n (в этом случае возвращается его индекс), либо интервал поиска не станет вырожденным (в этом случае понятно, что такого элемента нет, и нужно возвратить -1).

3. Реализовать алгоритм интерполяционного поиска, который отличается от двоичного тем, что вместо среднего элемента массива рассматривается элемент, полученный в результате попытки «угадать» его положение в массиве, исходя из гипотезы о равномерном распределении элементов в массиве, т. е. из гипотезы о том, что «график» элементов массива приблизительно представляет собой отрезок.

4. Написать программы, позволяющие сравнить производительность различных алгоритмов поиска, следующим образом. Создать массив из 10 тысяч элементов и заполнить его случайными числами от 0 до 30 тысяч. Сделать в цикле 10 тысяч поисков и вывести число миллисекунд, затраченных на это (в качестве искомого числа использовать элемент массива со случайным индексом). Перед циклами двоичных и интерполяционных поисков отсортировать массив.

5. Написать функцию, принимающую двумерный массив целых чисел и целое значение и осуществляющую в массиве:

а) линейный поиск;

б) двоичный поиск (предполагая, что элементы массива упорядочены по строкам).

6. Написать функцию, принимающую вещественное число, массив упорядоченных вещественных чисел, его длину и определяющую пару соседних элементов массива, между которыми лежит данное число (методом, аналогичным двоичному поиску).

7. Написать функцию, принимающую отсортированный массив целых чисел, его длину и значение искомого элемента и возвращающую индекс первого элемента в массиве, имеющего искомое значение. Если в массиве такого элемента нет, нужно возвратить -1. Эта функция должна работать за время, пропорциональное логарифму от числа элементов массива, т. е. линейный поиск не годится. Также не годится метод, состоящий в том, чтобы найти какой-то элемент с заданным значением (не обязательно первый), а затем перебирать элементы с заданным значением, пока не встретим первый (таких элементов может быть много).

8. Написать функцию, принимающую двумерный массив целых чисел (считается, что его строки лексикографически, т. е. как слова в словаре, упорядочены) и одномерный массив, и возвращающую его номер как строки в двумерном массиве (если в двумерном массиве нет такой строки, возвращается -1).

40 Строки

40.1 Объявление переменной типа строка.

В языке C строка — это просто указатель на первый символ этой строки. Для того, чтобы данную строку можно было обрабатывать при помощи библиотечных функций, она должна заканчиваться нулевым байтом, т. е. символом с кодом 0.

Объявить строковую переменную в таком стиле можно двумя способами:

а) как указатель:

```
char *s;
```

В этом случае программист сам должен заботиться о выделении памяти под символы, составляющие строку. Например, можно сделать это так:

```
s = (char*) malloc(число_символов_строки+1);
```

Здесь $+1$ нужен для хранения завершающего строку нулевого символа. При использовании динамических массивов нужно заботиться о своевременном их освобождении функцией free.

Однако, такую переменную можно инициализировать строковой константой:

```
char *s = "текст_строки";
```

Такая возможность существует благодаря тому, что в C строковая константа — это просто указатель на первый символ строки, заключенной в кавычки (завершающий нулевой символ добавляется автоматически компилятором). Однако, обычно строковые константы хранятся в той части памяти, куда нельзя ничего записывать, так что у так определенной строки нельзя менять входящие в ее состав символы. Если же мы хотим, чтобы содержимое строки могло меняться, нужно пользоваться массивами (динамическими или обычными).

б) как массив:

```
char s[максимальное_число_символов+1];
```

Здесь память выделяется автоматически компилятором, и символы такой строки можно менять, но если этот массив — локальная переменная, то надо быть готовым к тому, что он исчезнет по окончании работы содержащей его функции. В частности, такие строки нельзя возвращать из той функции, в которой они определены (для этого нужно пользоваться динамическими массивами).

Такая строка тоже может быть инициализирована, как традиционным для массивов образом (в этом случае завершающий символ с кодом 0 надо писать явно)

```
char s[число] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

так и в сокращенной записи (завершающий нулевой символ добавляется автоматически компилятором)

```
char s[число] = "Hello";
```

Как и в случае массивов, указанное в квадратных скобках число элементов должно быть достаточным для записи всех символов строки, включая завершающий символ с кодом 0. Если мы хотим, чтобы размер массива совпадал с числом символов той строковой константы, которая в него записывается (включая завершающий 0), то число в квадратных скобках можно опустить:

```
char s[] = "Hello";
```

В этом примере будет определен массив из шести символов.

2. Типы символов

Заголовочный файл ctype.h содержит набор весьма удобных макроопределений для проверки символа на принадлежность определенному классу. Например, если c — символьная переменная, то условие isspace(c) будет истинно, если c — пробельный символ. Таких макроопределений достаточно много, можно упомянуть несколько самых востребованных: isdigit — цифра и isalpha — латинская буква.

Здесь будет уместно следующее замечание: все эти макроопределения в случае, если ответ — ложь, возвращают 0, но если ответ — истина, они возвращают не 1, как можно было бы ожидать, а некоторое ненулевое число. Так что вариант

```
if(isspace(c)) ...
```

будет работать, а вот такой код

```
if(isspace(c)==true) ...
```

— скорее всего, нет.

3. Доступ к отдельному символу.

Как и для любого массива, доступ к символу из строки s по индексу (начинается с 0) выглядит как s[индекс].

4. Операции над строками.

Для того, чтобы пользоваться библиотечными функциями для работы со строками, нужно подключить файл string.h .

а) длина строки s дается функцией strlen(s)

б) к сожалению, строки в стиле C нельзя ни присваивать, ни сравнивать при помощи обычных операций сравнения. Для копирования строк существует функция strcpy, принимающая два указателя на символы dest и src, и копирующая строку, на которую указывает src, в область памяти, на которую указывает dest. Завершающий символ с кодом 0 тоже копируется этой функцией автоматически.

Например, если нужно записать строку "Hello"в массив символов s, нужно написать

```
strcpy(s, "Hello");
```

Здесь нужно сделать два замечания. Во-первых, две строки, которыми оперирует strcpy, не должны перекрываться в памяти. Например, такая попытка сдвинуть строку, удалив ее первый символ, незаконна:

```
strcpy(s, s+1);
```

Вообще говоря, имеется аналогичная функция и для случая, когда строки перекрываются, она называется memmove (в том же заголовочном файле string.h), но она требует явного указания числа копируемых байтов).

Во-вторых, вся ответственность за то, чтобы в строке dest хватило памяти для записи строки src, всецело лежит на программисте. Иногда такие ошибки ловит операционная система, но возможен и худший вариант, когда программа молча выдает неверный ответ. Использование такой функции без проверки переполнения того буфера, куда записываются данные, представляет серьезную угрозу для безопасности того компьютера, на котором выполняется такая программа. Именно эту ошибку программы fingerd использовал первый интернет-червь.

Для надежности в этом плане существует функция strncpy, имеющая дополнительный параметр (он идет последним), определяющий число копируемых символов. Если строка src короче, оставшиеся до указанного числа символы заполняются нулями, но если она длиннее, то завершающий нулевой символ не пишется.

в) для сравнения строк используется функция strcmp. Ее параметрами будут сравниваемые строки, и она возвращает 0, если строки совпадают, отрицательное число, если первая строка меньше, и положительное, если она больше. Так что для проверки равенства двух строк a и b нужно написать

```
strcmp(a, b)==0
```

а никак не a==b (что, вообще говоря, допустимо, но проверяет не то, что совпадает содержимое строк, а то, что они находятся в памяти по одному и тому же адресу, т. е. занимают одно и то же место в памяти — далеко не одно и то же).

По аналогии с strcpy, существует также вариант strncmp, сравнивающий только определенное количество начальных символов.

г) имеется также функция strcat с двумя строками-параметрами, приписывающая к строке, заданной первым параметром, строку, заданную вторым. Как и в случае strcpy, строки-параметры не должны перекрываться, и наличие достаточного количества памяти для приписывания — забота программиста. Имеется и вариант strncat; в отличие от strncpy, нулевой завершающий символ приписывается к результату всегда, и если вторая строка короче указанного числа символов, копируются только символы до завершающего нулевого включительно.

д) превращение строки s в целое число дается выражением atoi(s), в вещественное — atof(s). Обе эти функции объявлены в файле stdlib.h . К сожалению, для обратного превращения числа в строку отдельных библиотечных функций нет, и нужно пользоваться функцией sprintf.

е) ввод строки a с клавиатуры выполняется вызовом scanf("%s a). Такой вызов не проверяет переполнения буфера. Есть и проверяющий вариант: если a — массив из 100 символов, нужно написать scanf("%99s a) (не забываем про завершающий символ с кодом 0). При таком способе чтения строки все пробельные символы считаются разделителями, так что строка читается до первого пробела. Если же нужно считать строку с возможными пробелами до символа перевода строки, нужно написать scanf("%[^\n] a) . Если же нужны обе возможности (ограничение числа символов и считывание пробелов), эти варианты очевидным образом комбинируются.

ж) для выделения подстроки следует использовать функцию strncpy со сдвинутым указателем src. Например, выделить в массив dest подстроку строки src, начиная с символа с индексом 5 из 20 символов, можно так:

```
strncpy(dest, src+5, 20); dest[20] = 0;
```

з) найти первое вхождение символа c в строку s можно, написав

```
strchr(s, c)
```

Функция strchr возвращает указатель на первое вхождение символа в строку, или NULL, если символ не входит в строку.

Аналогично, найти последнее вхождение можно при помощи функции strrchr.

и) найти указатель на первый символ первого вхождения строки s1 в строку s2 можно при помощи вызова strstr(s2, s1). Этот вызов имеет значение NULL, если вхождений нет.

Задачи.

1. Ввести имя файла, возможно, содержащее путь, и вывести:

а) Путь без имени файла (если пути не было — вывести пустую строку).

б) Имя файла с расширением, но без пути.

в) Расширение имени файла.

г) Имя файла без пути и без расширения.

д) Имя последней папки в пути.

е) Имя файла с путем, но расширение заменено на html.

2. Ввести две строки и вывести число вхождений второй строки в первую.

3. (Транслитерация) Ввести три строки, и в первой из них заменить символы, входящие во вторую, на символы с такими же индексами из третьей строки. Предполагается, что вторая и третья строки имеют равные длины; если вторая строка имеет значение «abc», а третья — «def», то в первой строке «a» заменяется на «d», «b» на «e», «c» на «f», а остальные символы остаются без изменений.

4. Написать функцию, которая принимает строку и возвращает количество содержащихся в этой строке чисел (под числом понимается последовательность идущих друг за другом цифр, ограниченная с двух сторон нецифровыми символами или началом/концом строки).

5. Написать функцию, которая принимает натуральное число и возвращает строку, содержащую запись этого числа римскими цифрами.

6. Написать функцию, которая принимает две строки, в которых записаны неотрицательные целые числа и возвращает новую строку, содержащую их сумму (перевести обе строки в числа, например, типа int или даже long long int нельзя, поскольку строки могут быть достаточно длинными).

7*. Аналогично предыдущей задаче, но произведение чисел.

8*. Написать функцию, которая принимает две строки, в первой из которых записан шаблон, а во второй — текст. Функция должна возвращать true, если текст соответствует шаблону, и false иначе. В шаблоне допускаются обычные символы, которые соответствуют самим себе, символ «?», который соответствует любому, но ровно одному символу, символ «*»,соответствующий любой строке из любых символов, наконец — конструкция «[строка]», соответствующая одному символу, но только из числа тех, которые имеются в строке, заключенной в квадратные скобки. Например, строка abcd соответствует шаблону a?c?. Строка axcf тоже соответствует этому шаблону, а строки acd и brct — нет.

41 Структурные типы данных

Одним из центральных понятий современных языков программирования является понятие абстракции данных. Оно позволяет программисту решать поставленную перед ним задачу в терминах самой задачи, а не в терминах компьютера, на котором задача решается. Например, если нам нужно написать программу для учета автомобилей на складе автомагазина, мы можем захотеть иметь описание всех тех автомобилей, которые у нас имеются в наличии. Если мы решаем задачу в терминах компьютера, нам необходимо завести несколько одинаковых по количеству элементов массивов, каждый из которых хранит один из аспектов наших автомобилей — один массив хранит цены (целые числа), другой — цвета (строки), третий — мощность двигателя (вещественные числа), и так далее. Если таких массивов потребуется достаточно много, неудобство этого подхода очевидно.

На практике тот факт, что язык программирования поддерживает абстракцию данных, выражается в возможности создания новых типов данных, значения которых способны более-менее точно (с точки зрения задачи) описывать объекты реального мира. Как правило, это структурные типы данных, т. е. наборы переменных с заданными программистом именами и типами из числа базовых типов языка или уже созданных к данному моменту пользовательских типов.

Вообще, как правило, в большинстве языков программирования переменная структурного типа представляет собой набор вложенных переменных, каждая из которых имеет имя и тип, структурный или базовый (такой, например, как целое или вещественное число). Каждая переменная структурного типа имеет свои значения всех вложенных переменных, т. е. одна и та же вложенная переменная в составе разных переменных структурного типа может иметь разные значения.

Сам структурный тип данных определяет, какие именно вложенные переменные содержатся в каждой переменной этого типа. Это значит, что любые две переменные одного структурного типа имеют один и тот же состав, т. е. состоят из одного и того же набора вложенных переменных (с теми же именами и типами), возможно, имеющих разные значения.

Сами вложенные переменные обычно называются полями большой структурной переменной, их содержащей. Поля могут иметь любые типы, в том числе структурные; наличие у одной структурной переменной полей другого структурного типа называется композицией. В языке C не допускается, чтобы структурный тип данных имел поле того же типа; более того, запрещены даже циклические зависимости такого рода, т. е. например, два структурных типа, каждый из которых имеет поле другого типа из них.

Продолжая пример с автомобилями, теперь мы можем определить структурный тип данных «автомобиль», содержащий поля, описывающие разные аспекты автомобиля, которые могут интересовать потенциальных клиентов: например, поле price (целого типа) может содержать цену, поле color (строка) — цвет, и т. д.

Поддержка языком программирования абстракции данных обычно не ограничивается возможностью создавать новые типы данных, но простирается дальше и обеспечивает возможность работать с этими новыми типами так же, как и с любыми другими, например встроенными. В частности, это означает возможность создавать в программе переменные таких типов, возможность передавать их в качестве параметров подпрограммам и возвращать в качестве результатов, возможность присваивать значение одной переменной структурного типа другой переменной того же типа и т. д.

Двигаясь далее в примере с автомобилями, мы можем теперь завести массив автомобилей, т. е. переменных, каждая из которых имеет тип «автомобиль». Кроме того, мы можем передавать автомобили в функции в качестве параметров, возвращать их в качестве результатов, присваивать одной переменной типа автомобиль значение другой переменной такого же типа. Это все и позволяет нам наконец писать нашу программу в терминах автомобилей, а не в терминах встроенных типов конкретного языка программирования — символьных, целых, вещественных, логических переменных и иногда строк.

Как и в большинстве императивных языков программирования, в C имеются структурные типы данных; обычно, для обозначения этого понятия в языке C используется термин «структуры». Структуры похожи на массивы тем, что позволяют объединять под одним именем переменной несколько переменных. На этом, однако, сходство с массивами и заканчивается. В отличие от массивов, объединяемые в одну структуру переменные могут иметь разные типы, и доступ к ним, в отличие от массивов, происходит с использованием специальных идентификаторов, а не номеров.

Как и для переменных или функций, существуют объявления и определения структур. Объявление структурного типа данных выглядит следующим образом:

```
struct имя;
```

Этого достаточно только для объявления или определения указателей на данный структурный тип. Для того, чтобы определять переменные данного структурного типа, нужно его определение, которое выглядит так:

```
struct имя { тело } переменные;
```

Тело структурного типа данных, в частности, может содержать определения полей, в том числе битовых. Под полями (термин из теории баз данных) структуры понимаются те переменные, которые объединяются в структурную переменную.Определение поля выглядит как обычная декларация переменной.

Если у нас есть переменная типа структуры, для доступа к ее полям используется символ «.», за которым следует имя интересующего нас поля. Кроме того, в C имеется удобная операция для доступа к полю той структуры, на которую указывает указатель. Если p — указатель на структуру, а f — поле этой структуры, то выражение p->f означает поле f той структурной переменной, на которую указывает p.

Например, следующая структура описывает комплексное число:

```
struct Complex { double re, im; };
```

Каждая переменная только что описанного типа состоит из двух вещественных переменных с именами re и im. Если мы заведем теперь переменную z типа Complex, написав

```
struct Complex z;
```

мы на самом деле получим две вещественные переменные с именами z.re и z.im. Работать с этими переменными можно вполне также, как и с обычными вещественными переменными, т. е. мы можем присваивать им вещественные числа, считывать их значения при помощи функций ввода, или использовать их значения в выражениях или функциях вывода.

Теперь уместен вопрос: где же удобство? На первый взгляд, ситуация только ухудшилась: мы написали несколько лишних строк кода и в результате получили те же переменные, но с длинными, составными именами. Ответ на этот вопрос состоит в следующем: удобство такого подхода начинается, когда мы начнем передавать наши переменные в функции в качестве параметров. Дело в том, что переменные структурных типов данных можно передавать в функции, равно как и возвращать их из функций, ровно также, как и обычные, не структурные (или, как еще говорят, скалярные) переменные, скажем, целого или вещественного типа. В итоге, полученный код резко упрощается. Для сравнения можно привести два варианта организации суммы трех комплексных чисел:

```
//  --- без структур ---
void add1(double r1, double i1, double r2, double i2,
          double &r, double &i)
{
   r = r1 + r2;
   i = i1 + i2;
}
...
/* нужно сложить три комплексных числа:
   a + b i,  c + d i  и  e + f i; результат
   записывается в  g + h i

   понадобятся две вспомогательные вещественные
   переменные x и y */
  add1(a, b, c, d, x, y);
  add1(x, y, e, f, g, h);

//  --- со структурами ---
struct Complex add2(struct Complex z1, struct Complex z2)
{
    struct Complex res;
    res.re = z1.re + z2.re;
    res.im = z1.im + z2.im;
    return res;
}

/* нужно сложить три комплексных числа:
   z1, z2 и z3; результат записывается в z4
   вспомогательные переменные не понадобятся */
   z4 = add2( add2(z1,z2), z3 );
```

Второй вариант незначительно труднее в реализации, но зато много проще в использовании. Конечно, можно было бы возразить, что трудности в первом варианте надуманные, для решения задачи достаточно написать

```
g = a+c+e;
h = b+d+f;
```

вообще без всяких функций, однако трудности такого подхода резко возрастают при усложнении той операции, которую нужно вычислять (например, во что превратится этот подход, если нужно вычислить не сумму, а, скажем, произведение трех комплексных чисел? А четырех?).

Задача.

Написать структурный тип данных, описывающий автомобиль (в нем удобно будет, кроме полей, описывающих разные свойства автомобиля, иметь одно дополнительное логическое поле — зачем оно, будет ясно дальше). Написать также функции, принимающие в качестве параметра массив автомобилей и число его элементов и делающие следующее:

а) Инициализация. У каждого автомобиля из массива дополнительное логическое поле устанавливается в false, что означает, что массив пока пуст (ни одна из его ячеек пока не используется).

б) Заполнение склада. Пробежаться по массиву и для каждой ячейки, логическое поле которой имеет значение false, сделать следующее: ввести с клавиатуры значения полей, и логическое поле установить в true.

в) Печать информации. Пробежаться по массиву и для каждой ячейки, логическое поле которой имеет значение true, напечатать номер в массиве и информацию о соответствующем автомобиле.

г) Продать автомобиль. Ввести с клавиатуры его номер в массиве и проверить, что автомобиль с указанным номером имеется на складе, т. е. у соответствующего элемента массива логическое поле имеет значение true. Если это так, то продать автомобиль, т. е. установить его логическое поле в false. Иначе выдать сообщение об ошибке.

В функции main завести массив из 10 автомобилей, и организовать цикл, на каждой итерации которого пользователю выдается меню из пяти пунктов (четыре вышеописанных функции и выход из программы), вводится выбор пользователя и вызывается выбранная пользователем функция, до тех пор, пока пользователь не выберет выход.

Кроме только что рассмотренных обычных полей, существуют также так называемые битовые поля.

Битовые поля — это поля, хранящие целое число (знаковое или нет) и занимающие указанное число битов. Например, следующая структура имеет три битовых поля, каждое из которых занимает два бита; поскольку они беззнаковые (для знаковых надо писать int), диапазон значений каждого из них — от 0 до 3:

```
struct bitfields { unsigned a:2, b:2, c:2; };
```

Битовые поля, чаще всего, используются для экономии памяти. Используя битовые поля, в одном байте можно хранить 8 логических переменных, тогда как если каждую из них определять с типом bool, они в совокупности займут 8 байтов. Также битовые поля используются для программирования аппаратуры на низком уровне (на уровне регистров), поскольку один регистр, хранящий небольшое целое число, может делиться на некоторое число полей, управляющих независимыми функциями аппаратуры. В этом случае может возникнуть потребность пропустить некоторое число битов между соседними битовыми полями; для этого можно воспользоваться безымянным битовым полем, определяемым также, как и обычное, но без имени:

```
struct bitfields { unsigned a:2, :3, c:2; };
```

Здесь между a и c будут пропущены 3 бита.

Битовым полям можно присваивать значения или использовать их значения в выражениях. Однако, применять к ним операцию взятия адреса & нельзя, поскольку у битового поля, вообще говоря, нет своего адреса: адрес имеется только у всей ячейки целиком, а не у ее частей.

42 Объединения

В языке C есть еще одно средство организации структурных типов данных, называемое объединением. Объявление и определение объединения похоже на обычные структурные типы данных, но вместо struct используется ключевое слово union. Отличие объединения от структуры в том, что все его поля разделяют одну и ту же область памяти, и следовательно, в каждый момент времени использоваться может только одно поле.

Объединения обычно используются для следующих целей:

а) Экономия памяти. Если одновременно используется не более одного поля структуры, то такую структуру можно заменить объединением.

б) Преобразование типов данных, сохраняющее битовое представление. Обычное преобразование типов, например, с помощью операции (тип), старается по возможности сохранить значение преобразуемого выражения. Например, значением выражения (double)5 будет вещественное число 5.0, хотя битовое представление целого числа 5 и вещественного 5.0 сильно отличаются одно от другого. Однако, иногда бывает нужно преобразовать тип переменной, сохранив неизменным именно битовое представление. Например, иногда бывает удобно трактовать вещественное число как восемь подряд идущих байтов.

Для этого можно воспользоваться объединением с двумя полями, первое из которых имеет тип исходного выражения, а второе — тип, к которому его надо привести. Далее мы присваиваем первому полю интересующее нас значение, а из второго поля считываем результат преобразования.

Того же эффекта можно добиться при помощи преобразования типов указателей. Нужно просто взять указатель на интересующее нас значение и явно преобразовать его в указатель на интересующий нас тип.

в) Полиморфизм. Полиморфизм — это возможность производить одни и те же по смыслу действия над операндами разных типов при помощи одних и тех же выражений. Объединения можно использовать для того, чтобы один и тот же структурный тип данных мог представлять различные по смыслу объекты. Обычно для этого в структурном типе данных используются поля, определяющие, что представляет собой объект в данный момент, и объединение, поля которого представляют разные возможные варианты объекта.

Задача.

Написать структурный тип данных, который может представлять вещественное или целое число. Должны быть реализованы функции, обеспечивающие арифметические операции и операции сравнения

43 Библиотека ввода-вывода

Типы, глобальные переменные и прототипы функций для чтения или записи информации в файлы содержатся в заголовочном файле stdio.h . Прежде, чем обсуждать их использование, расскажем вкратце о том, как вообще устроены файлы с точки зрения программ на C.

С точки зрения прикладного программиста файл сам по себе представляет собой просто поименованную последовательность байтов, длина и имя которой могут меняться в процессе работы программы. Файлы нужны для того, чтобы хранить в них информацию, которая должна сохраняться между запусками использующих ее программ, примерно так же, как значение статических переменных в какой-либо функции сохраняется от одного запуска этой функции до другого.

Вообще говоря, файл на реальном устройстве внешней памяти организован, как правило, достаточно нетривиальным образом, но эта нетривиальность обычно скрыта от прикладных программистов средствами операционной системы, и они видят только поименованную последовательность байтов.

Кроме имени и содержимого, у файлов обычно имеется указатель позиции в файле, который показывает, куда будет производиться следующая запись информации или откуда она будет считана. Для удобства программистов, при записи или чтении этот указатель автоматически передвигается вперед по файлу, и поэтому при последовательном чтении или записи информации в файл о значении этого указателя можно вообще не беспокоиться (при чтении — до тех пор, пока этот указатель не упрется в конец файла). В C имеются функции для принудительного получения и изменения позиции этого указателя.

Обычно, у файлов имеется также набор атрибутов вроде времени последнего доступа или флагов разрешения доступа, показывающих, какие операции над файлом и кому можно выполнять. Состав таких атрибутов и их смысл обычно определяются типом файловой системы, на которой находится данный файл. Также, файловая система (способ хранения файлов на конкретном устройстве) может накладывать определенные ограничения на, скажем, максимальную длину файла или символы, из которых может состоять его имя. Набор типов файловых систем, которые можно использовать, определяется операционной системой.

Обычно, в достаточно распространенных операционых системах способы работы с файлами из программ на C примерно похожи.

Для работы с содержимым файла, для начала нужно завести в программе специальную переменную, которая будет представлять этот файл. Такая переменная передается как параметр во все функции, в которых производится чтение или запись в файл.

В C, такая переменная должна иметь тип FILE* (указатель на переменную типа FILE, который определен в заголовочном файле stdio.h).

После этого, если мы хотим записать что-то в файл или считать что-нибудь из файла, нам нужно произвести действие, которое называется открытием файла, и связать открытый файл с представляющей его переменной. Оба этих действия в C производятся с помощью одной функции fopen. Она принимает имя файла (возможно, с путем, в котором нужно удваивать символ \по правилам записи строковых констант в C) и режим его открытия, показывающий, что (и как) мы собираемся с ним делать, и возвращает указатель на FILE в качестве результата, который нужно просто присвоить заведенной ранее переменной.

Более подробно говоря о режиме открытия файла, этот режим представляет собой строку из максимум трех символов. Первый из них — обязательный, и он может быть «r» (файл открывается на чтение), «w» — на запись и «a» — на дописывание информации в конец файла. Следующий (необязательный) символ «+» означает, что для этого файла возможны обе операции (как чтение, так и запись; в этом случае «r» в первом символе означает, что файл должен существовать, а «w» — что он будет создан), и наконец, последний символ (тоже необязательный) «t» означает, что файл текстовый, или «b» — двоичный.

Различие между текстовым и двоичным файлами проявляется в основном в следующем:

1) В двоичных файлах при чтении и записи информации она не претерпевает никаких изменений, тогда как в текстовых файлах читаемая или записываемая в файл информация может быть подвергнута определенным модифи-

кациям. Например, в разных операционных системах разные правила относительно того, как разделяются строки в текстовых файлах. В Linux, как и в других клонах Unix, строки разделяются единственным символом перевода строки LF с кодом 10 (в C '\n'), а в Windows строки разделяются идущими подряд двумя символами CR с кодом 13 (в C '\r') и LF. Для более легкого переноса текстовых программ с одной ОС на другие при чтении текстового файла в Windows два подряд идущих символа, разделяющие строки, превращаются в один LF, а при записи — наоборот. В итоге, если открыть программу .exe или любой архив как текстовый файл, эти преобразования приведут к существенной порче информации. Не стоит думать, что, поскольку преобразования кажутся взаимообратными, в общем итоге даже при простом копировании информация сохранится. Это будет так только в тех редких случаях, когда в исходном файле перед каждым LF стоит свой CR, что, конечно, весьма маловероятно для произвольного двоичного файла.

2) В двоичных файлах конец файла определяется атрибутом длины файла, поэтому в них могут содержаться абсолютно любые символы. В текстовых файлах ОС Windows по историческим причинам символ с кодом 26 (^Z) означает конец файла, так что если открыть по-настоящему двоичный файл как текстовый, скорее всего, он будет считан не до конца, а до первого байта со значением 26.

Эти различия важны, потому что в Windows файлы по умолчанию, т. е. если не писать в строке режима ни t, ни b, открываются как текстовые.

После того, как файл будет открыт, возможны дальнейшие варианты. Обсудим собственно операции ввода и вывода информации в поток, однако, перед этим скажем пару слов о двух способах ввода и вывода информации: текстовом и двоичном.

Чтобы обсуждать текстовый способ ввода и вывода информации, сначала скажем пару слов о том, как вообще происходит обработка текстов в компьютере.

Начнем с того, что компьютер в принципе может обрабатывать только числа. Что бы мы ни хотели обработать с помощью компьютера (текст, звук, изображения, видео), сначала нужно закодировать, т. е. превратить в набор чисел. Затем этот набор чисел можно обработать на компьютере, и, наконец, результирующий набор чисел должен быть раскодирован обратно в текст, звук и т. д.

Что касается текста, он кодируется так: все символы, которые могут входить в текст, перенумеровываются определенным образом, и далее вместо текста в компьютере обрабатывается последовательность таких номеров. Кодировка текста — это и есть таблица, сопоставляющая символам (рассматриваемым как картинки) их номера и наоборот. Поэтому, если для ввода текста в компьютер использовалась одна кодировка, а для вывода будет применяться другая, получится абракадабра (студенты, работающие на Visual Studio, могли убедиться в этом лично при попытке вывести на экран строки из русских букв — дело в том, что в окне ввода программы для русских букв используется кодировка CP1251, а в окне терминала для их вывода используется кодировка CP866).

Так вот, пусть у нас есть число 513, и мы хотим вывести его в файл. Если мы применяем текстовый способ (функция printf), то это число из внутреннего представления (в памяти компьютера) в двоичной системе превращается в набор цифр в десятичной системе счисления (в нашем случае 5, 1, 3), и эти цифры как символы (с использованием кодировки символов) выводятся в поток. На тех компьютерах, на которых мы работаем, для цифр используется кодировка ASCII, согласно которой код символа 5 равен 48+5=53, код символа 1 равен 49 и код символа 3 — 51. В итоге, в поток попадут три байта со значениями 53, 49 и 51. Такой способ хорош тем, что так записанное в файл число можно прочесть в текстовом редакторе (например, блокнот в Windows или gedit в Linux). Это происходит потому, что текстовый редактор просто отображает те символы, номера (коды) которых он читает из файла. Редактор Word тут не годится, поскольку он пытается кроме кодов символов читать из своего файла еще много дополнительной информации о форматировании.

У текстового способа вывода в поток есть и недостатки: он медленно работает, и число попадающих в поток байтов зависит от величины числа.

Если нам нужно считывать из файла информацию, представленную там в текстовом виде, или, наоборот, писать ее туда, то для этого существуют функции fscanf и fprintf (они отличаются от их обычных аналогов без первой буквы f только наличием дополнительного первого параметра типа FILE*, означающего тот файл, с которым мы будем работать).

Что же касается двоичного вывода в поток нашего числа, то при таком способе число записывается в поток так, как оно хранится в памяти компьютера. Обычно целое число занимает четыре байта, и в нашем случае они будут содержать числа 1, 2, 0, 0 (поскольку $513 = 0 \cdot 256^3 + 0 \cdot 256^2 + 2 \cdot 256^1 + 1 \cdot 256^0$). Итак, в поток попадут четыре байта со значениями 1, 2, 0, 0. Если теперь открыть этот файл в блокноте, то вместо нашего числа мы увидим в лучшем случае странные символы, в худшем — вовсе ничего, поскольку 0, 1 и 2 — это номера в ASCII управляющих символов, т. е. смысл которых состоит не в отображении на экране определенной картинки, а в некоторых других действиях, которые раньше должен был предпринимать терминал, получая такие символы. Теперь, правда, иногда эти символы отображаются как рожицы.

Двоичный вывод работает быстрее текстового, и как правило, требует меньше памяти в потоке, но и у него есть свои недостатки. Если файл будет записан на компьютере с одним порядком байтов в целом числе (например, little endian — см. файл про указатели), а считан на компьютере с другим порядком байтов (например, big endian) — число будет считано совсем другое.

Для двоичного ввода-вывода в C имеются функции fread и fwrite.

При чтении файлов важно также иметь возможность проверить, не дошли ли мы до конца файла. Это делает функция feof (параметр — переменная, представляющая файл).

Как уже говорилось, в C есть функции для получения и изменения позиции в файле: fgetpos/fsetpos, fseek, ftell, rewind.

Наконец, когда все манипуляции с файлом закончены, чтобы информация, которую мы писали в файл, не пропала, файл должен быть закрыт функцией fclose.

Есть в C и функция, позволяющая связать переменную, уже связанную с одним потоком, с другим (т. е. изменить файл, с которым связана та или иная файловая переменная). Она называется freopen.

В библиотеке функций ввода-вывода также имеется три глобальных переменных, представляющих уже открытые файлы. Это stdin, называющийся стандартным потоком ввода, по умолчанию связанный с вводом терминала, куда попадает информация, вводимая с клавиатуры, и stdout (стандартный поток вывода) и stderr (поток вывода сообщений об ошибках), оба которых по умолчанию связаны с выводом терминала, т. е. помещаемая туда информация попадает на экран (в окне терминала, в котором запущена программа). Эти потоки используются теми функциями, которые не принимают параметров, определяющих поток, например, printf и scanf. Функция freopen позволяет связать стандартные потоки с файлами, после чего printf и scanf начнут использовать новые файлы.

Далее, рассмотрим еще один вариант ввода-вывода — строковый. Он похож на файловый, с той только разницей, что информация хранится не в файле на внешнем устройстве, а в строке в памяти. Функции, осуществляющие такие действия, называются sprintf и sscanf, и отличаются от их файловых аналогов только тем, что их первый параметр — не файл, а указатель на первый символ соответствующей строки. При этом вся забота о том, чтобы при выводе информации в строку она не переполнилась, ложится на плечи программиста.

Напоследок скажем и о функциях, позволяющих управлять файлами в целом:

1) remove — удалить файл, имя которого передается как параметр.

2) rename — переименовать файл.

Задачи.

1. Написать программу, которая выводит в текстовый файл "f.txt" таблицу значений sin(x) для x, меняющегося от 0 до 1 с шагом 0.1. Первая строка этой таблицы должна быть заголовком вида "x sin(x)".

2. Написать программу, считывающую из текстового файла "g.txt" фамилии абитуриентов и их оценки по математике и физике и выдающую на экран таблицу, содержащую фамилии и сумму баллов за оба экзамена. Рассмотреть два случая:

а) формат данных во входном файле "g.txt" представлен следующим примером:

```
Иванов   5 5
Петров   4 5
Алексеев 5 4
...
```

б) Кроме данных в формате пункта а) входной файл содержит первую строку заголовка, имеющую вид "Фамилия математика физика".

3. Написать функцию, принимающую две строки и копирующую файл с именем, заданным первой строкой, в файл, имя которого задано второй строкой. Старое содержимое последнего файла, если такой был, удаляется.

4. Аналогично предыдущей задаче, но первый файл дописывается в конец второго.

5. Написать функцию, принимающую имя файла в качестве параметра и подсчитывающую и печатающую частоты вхождения отдельных символов в этот файл. Рассмотреть два случая:

а) ASCII файл, т. е. коды всех записанных в нем символов находятся в пределах от 0 до 127.

б) Общий случай.

6. Написать функцию, вычисляющую сумму целых чисел, записанных в файле (их число заранее неизвестно).

7. Написать функцию, читающую текстовый файл и пишущую в другой файл суммы целых чисел из каждой строки первого файла (в нем записано несколько строк, в каждой из которых — несколько целых чисел, разделенных пробелами).

8. Написать программу, открывающую двоичный файл f.dat, в котором хранятся 10 целых чисел, и заменяющую в этом файле все значения 0 на 5, а также печатающую на экран все 10 чисел после замены.

9. Написать две функции save и load, принимающие в качестве параметров массив из 15 вещественных чисел и строку, содержащую имя двоичного файла. Функция save должна записывать массив-параметр в файл, а load — загружать массив из файла.

10. Написать функцию, принимающую имя двоичного файла и возвращающую сумму вещественных чисел типа double, записанных в этом файле (числа записаны в файле подряд, в том виде, в котором они записываются в памяти компьютера).

11. Написать функцию, принимающую имя двоичного файла и заменяющую в нем каждое число на сумму его и всех предыдущих (в том же файле).

12. Написать функцию, принимающую в качестве параметров два положительнных целых числа и возвращающую квадрат результата их сцепления. Например, если переданы числа 1 и 2, результатом их сцепления будет 12, и функция должна возвращать 144. Написать программу, тестирующую данную функцию.

13. Написать программу, создающую 10 текстовых файлов, каждый из которых содержит таблицу значений функции sin на промежутке от 0 до 20 с шагом $\frac{i}{10}$, где i — номер таблицы (меняется от 1 до 10). Имя файла, содержащего конкретную таблицу, начинается со строки "sin_ к которой приписывается значение шага. Напирмер, файл, содержащий таблицу с шагом $\frac{1}{2}$, должен называться "sin_0.5".

14. Имеется текстовый файл, в котором записаны в произвольном порядке числа от 1 до 1000000, причем каждое число может встречаться не более одного раза. Требуется записать новый файл, в котором встречаются те же числа, но в порядке возрастания, наибыстрейшим образом.

15. Написать функцию, читающую текстовый файл и пишущую в другой файл суммы целых чисел из первого файла, в которые входит наибольшее количество последовательных чисел так, чтобы эти суммы были меньше 1000 (можно считать, что все числа из первого файла неотрицательны).

44 Связные списки

Ранее мы уже познакомились с массивами — средством объединения нескольких однотипных переменных, доступных по номеру, который может быть результатом выражения. Как мы знаем, этот способ хранения последовательности однотипных переменных имеет ряд недостатков — трудно вставить новый элемент в середину массива и даже просто расширить массив, добавив один или несколько элементов в конец. Расширить обычный (не динамический) массив в принципе невозможно, а для расширения динамического массива нужно, как правило, выделить память под новый (больший) массив, переписать туда имевшиеся в старом массиве элементы, и удалить старый. Это весьма дорогостоящая по времени операция, и после нее все указатели на элементы старого массива становятся висячими.

Рассмотрим теперь другое решение задачи хранения последовательности однотипных переменных, не обладающее указанными в предыдущем абзаце недостатками. Оно называется «связный список» и состоит в том, чтобы хранить элементы из нужной нам последовательности в так называемых узлах, каждый из которых содержит один элемент данных и указатель на следующий узел; в узле, содержащем последний элемент данных, указатель на следующий узел имеет значение 0.

Чтобы использовать так устроенную структуру данных, нам нужно завести новый структурный тип данных (тип узла), содержащий два поля — данные (те элементы последовательности, которые нам надо хранить) и указатель на этот же тип узла. К счастью, C позволяет в структурном типе данных иметь указатель на тот же тип, который мы определяем (это происходит потому, что размер памяти, отводимой под указатель, не зависит от размера памяти, отводимой под значение того типа, на который он указывает).

Если нам нужен список из элементов конкретного типа, например, целых чисел, тип узла может выглядеть так:

```
struct Node
{
    int d;     // элемент данных
    struct Node *n;   // указатель на следующий узел
};
```

Если же в нашей программе встречаются списки для разных типов элементов (например, список целых чисел, список вещественных чисел, список строк и др.), можно завести макроопределение с параметром, в котором передается нужный нам тип элемента данных в узле.

```
#define NODE(TYPE)  \
struct Node_##TYPE  \
{                   \
    TYPE d;         \
    struct Node_##TYPE *n; \
};
```

Это макроопределение, к сожалению, работает не для всех типов, а только для тех, имя которых состоит из одного слова. Чтобы пользоваться им для других типов (составные имена типов, указатели, массивы, структуры), нужно воспользоваться ключевым словом typedef и создать синоним нужного нам типа, имя которого состоит из одного слова, и уже это имя подставлять в макроопределение.

Список представляется в программе как указатель на первый его узел (список, вообще говоря, может быть и вовсе пустым, т. е. не содержащим ни одного узла; такой список представляется указателем 0). Вообще говоря, есть и другой подход к реализации связного списка, когда он представляется в программе при помощи переменной, содержащей

первый узел. Такой подход принят в языках программирования, в которых нет явного понятия указателя, например Java. В C такой подход неудобен, потому что непонятно, как представлять пустой список (в Java этот недостаток не проявляется, потому что каждая структурная переменная может иметь специальное значение null, говорящее о том, что в этой переменной нет никакого значения).

Список отличается от традиционного для хранения последовательности элементов массива тем, что, с одной стороны, легко (и быстро) расширяется и позволяет вставлять элементы в любой точке последовательности, но зато страдает очень медленным доступом к элементу по номеру (для доступа к конкретному элементу по номеру нужно просмотреть все предыдущие элементы, и чтобы выяснить, что список короче, чем надо, необходимо просмотреть его весь).

Наличия типа узла достаточно для написания программы, использующей связные списки. Для примера рассмотрим функцию, вычисляющую длину списка, на первый узел которого указывает ее параметр:

```
int length(struct Node *f)
{
    struct Node *p = f; // указатель p пробегает все элементы
                        // списка, начиная с того, на который
                        // указывает f
    int i = 0;
    while(p != 0)
    {
        ++i;      // подсчет числа элементов
        p = p->n; // переход к следующему элементу
    }
    return i;
}
```

Задачи.

1. Написать макроопределение, дающее функцию, вычисляющую длину списка для любого типа его элементов (параметр макропределения).

2. Написать функцию, возвращающую указатель на второй элемент списка.

3. Написать функцию, возвращающую указатель на последний элемент списка.

4. Написать функцию, возвращающую указатель на элемент, находящийся в заданной позиции списка (добавляется один целый параметр — номер необходимой позиции; 0 — первый элемент списка, 1 — второй элемент, ..., длина списка−1 — последний элемент): а) для списка целых чисел; б) для списка из любого типа элементов.

5. Написать функцию, печатающую все элементы списка на экран.

6. Написать функцию, подсчитывающую элементы списка, удовлетворяющие определенному условию (добавляется один параметр — указатель на функцию, принимающую элемент данных и возвращающую логическое значение; считать надо те элементы, на которых эта функция возвращает true).

7. Написать функцию, возвращающую указатель на: а) первый; б) последний элемент списка с указанными данными (второй параметр функции).

Следующий пример — функция, добавляющая в начало списка новый элемент с данными, указанными в ее втором параметре. Первый параметр здесь будет уже другой — указатель на указатель на первый узел того списка, куда мя хотим добавить элемент. Нам нужен указатель на указатель потому, что в результате работы этой функции указатель на первый элемент списка изменится; раньше он указывал на бывший первый элемент, а после будет указывать на новый первый элемент этого списка. Возможен и другой вариант: мы по-прежнему принимаем в качестве первого параметра указатель на бывший первый узел, но возвращаем указатель на новый первый узел в качестве результата.

```
void add_first(struct Node **f, int d)
{
    struct Node *p;
    p = (struct Node *)malloc(sizeof(struct Node));
    p->d = d;
    p->n = *f;
    *f = p;
}
```

Использование в функции malloc операции sizeof важно, потому что размер структуры далеко не всегда равен сумме размеров ее элементов. Иногда компилятор вставляет между полями структуры дополнительные промежутки. Это явление называется выравниванием, и имеет целью ускорение работы программы за счет увеличения объема используемой памяти.

Задачи.

1. Написать макроопределение, дающее функцию, вставляющую первый элемент в список для любого типа его элементов.

2. Написать макроопределение, дающее функцию, вставляющую последний элемент в список для любого типа его элементов.

3. Написать функцию, вставляющую элемент в заданную позицию списка (добавляется один целый параметр — номер необходимой позиции; 0 — вставка в начало списка, 1 — после первого элемента, ..., длина списка — вставка в конец): а) для списка целых чисел; б) для списка из любого типа элементов, при помощи а) обычных указателей, с разбором случаев: вставляется первый или не первый элемент списка; б) указателей на указатели, без разбора таких случаев.

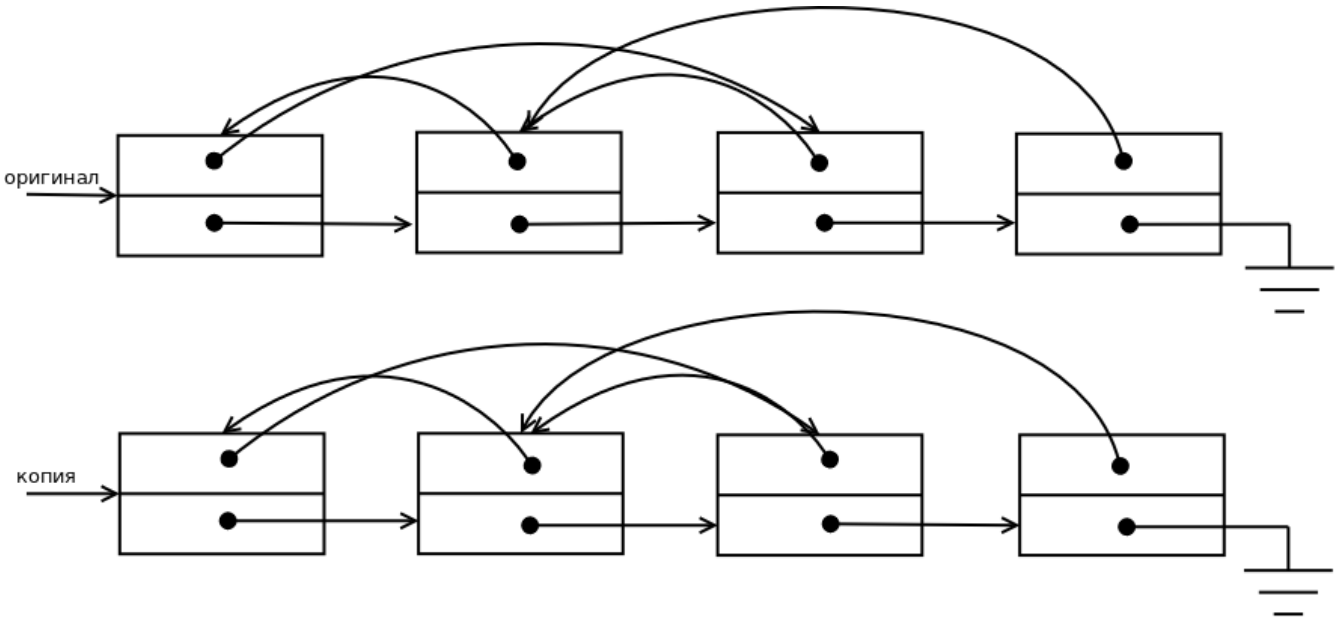
4. Написать функцию без параметров, считывающую элементы списка с клавиатуры до тех пор, пока элементы не кончатся (конец потока) и возвращающую указатель на первый элемент созданного списка: а) при помощи операции вставки элемента в конец списка (порядка n^2 действий, где n — длина списка); б) более быстрый вариант — порядка n действий (используя вспомогательный указатель на текущий узел), в) рекурсивно.

5. Написать функцию, строящую копию списка (другой список (т. е. при его построении узлы заводятся при помощи malloc, а не заимствования из того списка, который копируется), содержащий ту же самую последовательность данных в узлах): а) рекурсивно; б) с разбором случаев (см. задачу 3); в) используя указатели на указатели.

6. Построить копию списка, но в обратном порядке.

7. Поменять порядок узлов в списке на обратный (только путем изменения указателей на первый элемент и в узлах, без переприсваивания данных в узлах).

8*. Дан список, каждый узел которого в качестве данных имеет другой указатель на узел, причем известно, что он указывает на какой-то элемент того же списка. Требуется построить копию данного списка, в которой дополнительный указатель на узел («данные») указывает на узел копии с тем же номером в копии, с каким указатель в оригинале для данного узла указывал на элемент исходного списка (за линейное время, на константной памяти, кроме заведения копий узлов). Пример:



Следующий пример — функция, удаляющая первый элемент списка:

```
void del_first(struct Node **f)
{
    struct Node *p;
    if(*f == 0) return;
    p = *f;
    *f = (*f)->n;
    free(p);
}
```

Эта операция тоже меняет указатель на первый элемент списка, поэтому и здесь надо передавать указатель на него. Также, нам нужно перед выполнением этой операции проверить, что список не пуст.

Задачи.

1. Написать функцию, удаляющую первый элемент из списка для любого типа его элементов.

2. Написать функцию, удаляющую последний элемент из списка для любого типа его элементов.

3. Написать функцию, удаляющую элемент из заданной позиции списка (добавляется один целый параметр — номер необходимой позиции; 0 — удаление первого элемента списка, 1 — второго элемента, ..., длина списка−1 — удаление последнего элемента): а) для списка целых чисел; б) для списка из любого типа элементов.

4. Написать функцию, удаляющую все элементы списка.

5. Написать функцию, удаляющую элементы списка, удовлетворяющие определенному условию (добавляется один параметр — указатель на функцию, принимающую элемент данных и возвращающую логическое значение; удалять из списка надо те элементы, на которых эта функция возвращает true): а) с разбором случаев; б) через указатель на указатель.

6. Найти все простые числа от 2 до 1000, используя решето Эратосфена: сначала строится список, содержащий все числа из указанного диапазона; затем берется первый элемент, и из списка выкидываются все узлы, содержащие числа, кратные ему; затем аналогичная процедура применяется для второго элемента списка, затем для третьего, и т. д. до предпоследнего.

7. (Задача Иосифа Флавия, подробности см. в википедии) Построить кольцевой список (в котором последний элемент указывает на первый), узлы которого хранят последовательные числа от 1 до 40. Затем, начиная с узла, содержащего 1, выкидывать каждый пятый узел по порядку, печатая числа, содержащиеся в удаляемых узлах (в конце, когда число узлов станет меньше пяти, при определении пятого узла некоторые узлы придется считать несколько раз), до тех пор, пока не останется один узел. Каков будет его номер?

8. Написать функцию слияния, принимающую в качестве параметров два упорядоченных по возрастанию списка и из их элементов строящую новый упорядоченный список, меняя только значения указателей в узлах. Функция должна возвращать указатель на первый элемент полученного списка.

9. Используя функцию предыдущей задачи, написать функцию, сортирующую список за время порядка $n \ln n$, где n — длина исходного списка.

Только что рассмотренные связные списки обладают одним очень существенным недостатком. Он состоит в том, что, имея указатель на какой-то элемент списка, мы не можем получить указатель на предыдущий элемент. Практическим следствием этого недостатка будет, например, невозможность удалить из списка элемент, на который указывает известный нам указатель, поскольку для удаления из списка данного элемента мы должны подправить указатель на следующий элемент, расположенный в предыдущем узле, а указатель на него, т. е. предыдущий узел, мы получить не можем.

Для исправления этой ситуации было изобретено понятие двусвязного списка. Узел такого списка отличается от узла обычного списка тем, что у него есть дополнительное поле, указывающее на предыдущий узел списка (у первого узла списка этот указатель, очевидно, равен 0). Так что описание узла двусвязного списка выглядит следующим образом:

```
struct DNode
{
    int d; // элемент данных
    struct DNode *p, *n; // указатели на предыдущий и следующий узлы
};
```

Кроме того, хотя для доступа к любому узлу такого списка достаточно иметь указатель на любой из его узлов (в том числе на первый узел, как у обычных списков), часто для двусвязного списка хранят как указатель на первый узел, так и указатель на последний.

В качестве примера разберем реализацию функции swap_, меняющей местами два первых элемента двусвязного списка, для написания которой применим технологию опорных указателей. Эта технология состоит в следующем: сначала нужно завести несколько переменных-указателей на узлы, указывающих на все те узлы, указатели на которые нам понадобятся в процессе работы. В данном случае, для перемены мест первых двух узлов списка нам понадобятся указатели на первые три узла нашего списка (если в списке меньше двух узлов, то менять местами нечего, а если ровно два — третий указатель будет 0). Затем, пользуясь только что заведенными указателями, мы проставляем заново значения всех тех указателей в узлах, которые должны измениться в результате нашей операции.

Эта технология использует обычно больше переменных, чем нужно; в большинстве случаев можно обойтись меньшим числом вспомогательных переменных, но использование этой технологии резко упрощает решение задач, состоящих в изменении структуры данных, связанном с изменением значений указателей.

```
void swap_(struct DNode **f)
{
    struct DNode *a, *b, *c;
    if(*f==0 || (*f)->n==0) // в списке меньше двух узлов
        return;
```

```
    a = *f; b = (*f)->n; c = (*f)->n->n; // вспомогательные указатели
    *f = b;
    a->p = b; a->n = c; // бывший первый узел станет вторым
    b->p = 0; b->n = a; // бывший второй узел станет первым
    if(c!=0) c->p = a;  // если было больше двух узлов
}
```

Задачи.

1. Написать функцию, возвращающую указатель на предпоследний элемент двусвязного списка с данными 1 перед третьим элементом с данными 0.

2. Построить копию двусвязного списка.

3. Написать функцию, удаляющую элемент двусвязного списка по указателю на него (и указателю на первый элемент списка).

4. Написать функцию, меняющую порядок узлов двусвязного списка на обратный (и меняющую указатель на первый элемент соответствующим образом).

45 Стеки и очереди

Для решения очень многих задач программирования используются такие понятия, как стек и очередь. Мы обсудим в этом разделе их реализацию и некоторые приложения.

45.1 Стек

Стек — структура данных, предназначенная для хранения последовательности элементов и умеющая выполнять следующие операции:

1) empty — проверка, пуст ли стек.

2) push — положить в стек элемент.

3) pop — удалить из стека последний положенный туда элемент и вернуть его в качестве результата.

Часто стек называют LIFO (сокращение от Last In, First Out — последним пришел, первым ушел).

Задачи.

1. Реализовать стек с использованием массива.

2. Реализовать стек с использованием связного списка.

3. Используя стек, реализовать алгоритм вычисления выражения, записанного в обратной польской записи (см. статью «Обратная польская запись» в википедии).

4. Используя стек, реализовать алгоритм перевода выражения из обычной математической в обратную польскую запись (см. статью «Алгоритм сортировочной станции» в википедии).

5. Используя решения предыдущих двух задач, написать функцию, принимающую строку, в которой записано выражение, и вычисляющую его.

6*. Используя решения предыдущих трех задач, написать функцию, принимающую строку, в которой записано выражение, и возвращающую его частную производную по переменной x.

45.2 Очередь

Очередь — тоже структура данных для хранения последовательности элементов, однако, добавление элементов в очередь происходит с одного конца, а извлечение — с другого. Для очереди используется также название FIFO (сокращение от First in, First Out — первым пришел, первым ушел).

Задачи.

1. Реализовать очередь с использованием массива.

2. Реализовать очередь с использованием связного списка.

3. (Кушниренко-Шень) Реализовать очередь, имея два стека (как черные ящики, с доступом только при помощи имеющихся у стеков операций).

45.3 Очередь по приоритетам

Очередь по приоритетам — структура данных, предназначенная для хранения набора элементов и умеющая выполнять следующие действия:

1) empty — проверка, пуста ли очередь.

2) push — положить в очередь элемент; в отличие от обычной очереди без всяких приоритетов, в этой операции, кроме самого элемента, указывается его приоритет — целое или вещественное число.

3) pop — удалить из очереди элемент с наибольшим приоритетом и вернуть его в качестве результата.

На самом деле, с этой структурой данных мы уже встречались в теме «сортировки», а именно при описании пирамидальной сортировки. Так что в качестве задач предложим лишь извлечь оттуда реализацию такой структуры данных (с использованием массива), и оформить это в качестве отдельной программы.

46 Деревья

Мы будем понимать под деревом неориентированный связный граф без циклов. Если убрать требование связности, получится понятие, представляющее собой набор из нескольких не связанных между собой деревьев. Это понятие называется лесом.

Кроме того, в программировании обычно рассматриваются деревья с двумя дополнительными требованиями:

1) Одна из вершин дерева выделена и называется корнем. В математике такие деревья называются корневыми. В дальнейшем мы будем рассматривать только корневые деревья и называть их просто деревьями. Обычно на рисунках корень изображается сверху, а все остальные вершины — ниже, и чем дальше они от корня, тем ниже на рисунке они изображаются.

Еще один важный для нас термин из области деревьев — лист. Так называется вершина, у которой имеется только одно ребро (в сторону корня).

Другим источником терминов, описывающих отношения вершин в дереве, являются родственные связи. Если две вершины соединены ребром, то та из них, которая находится ближе к корню, называется отцом для другой. Аналогично, если a — отец b, то b — сын a. Сыновья одного отца называются братьями. Также можно ввести понятия внука и дедушки, потомка и предка (вершина a — потомок b, а b — предок a, если путь из a в корень, а такой путь, как известно, единственен, проходит через b). Таким образом, лист — это вершина без детей, а все вообще вершины дерева — потомки корня.

Теперь можно сформулировать второе требование к деревьям, которое тоже предполагается всегда выполненным:

2) Для любой вершины ее сыновья всегда упорядочены, т. е. для любых двух братьев всегда можно сказать, какой из них левее, а какой правее.

46.1 Двоичные деревья

Эти деревья характеризуются двумя моментами: во-первых, у каждой вершины может быть не более двух сыновей (меньше — пожалуйста). Во-вторых, упорядочивание, имеющееся для братьев, распространяется на все дерево: считается, что для любой вершины левый ее сын со всеми своими потомками левее ее, а правый — правее. Даже если у вершины всего один сын, считается, что при создании дерева (в компьютере) должно быть указано, левый он или правый (соответственно, он левее или правее своего отца).

Как и для связных списков, для представления деревьев в компьютере используется тип узла; переменная такого типа хранит одну вершину. Мы будем хранить в ней поля, содержащие указатели на левого и правого сыновей соответственно. Если какого-то (или обоих) сына нет, соответствующий указатель (или оба) равны 0.

Кроме этого, в узле дерева мы также храним данные. Если нам нужно иметь в программе много разных деревьев (с разными типами данных в узлах), тип узла дерева удобно сделать макроопределением, чтобы его можно было использовать для любого типа хранящихся в узле данных.

Тип узла с целыми данными можно описать следующим образом:

```
struct BNode
{
    int d;
    struct BNode *l, *r;
};
```

В программе двоичное дерево будет представлено просто указателем на его корень, т. е. переменной типа struct BNode *. Как и связный список, дерево может быть пустым, т. е. не имеющим ни одной вершины. Такое дерево представляется указателем, равным 0.

Для создания конкретных деревьев в программе удобно использовать функцию, создающую узел дерева:

```
struct BNode *f(int d, struct BNode *l, struct BNode *r)
{
    struct BNode *p = (struct BNode *) malloc(sizeof(struct BNode));
    p->d = d;
    p->l = l;
    p->r = r;
```

```
    return p;
}

struct BNode *tree1 = f(1, f(2, f(4,0,0), f(5,0,0)), f(3, f(6,0,0), 0));
```

Это выражение построит нам дерево, изображенное ниже:

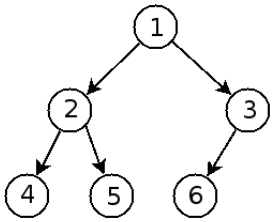


Рис. 1

Забегая несколько вперед (подробнее происходящее здесь будет обсуждаться в теме «Обход дерева»), выпишем функцию печати дерева. Дерево будет печататься, повернутое на 90° против часовой стрелки.

```
void print(struct BNode *r, int offset)
{
    if(r==0) return;
    print(r->r, offset+3);
    for(int i = 0; i<offset; i++)
        printf(" ");
    printf("%d\n", r->d);
    print(r->l, offset+3);
}
```

Эта функция для нашего дерева (на рис. 1) выведет следующее:

```
    3
        6
1
        5
    2
        4
```

В качестве примеров решим следующие задачи:

1. Написать функцию, принимающую в качестве параметра дерево (т. е. указатель на его корень) и возвращающую число сыновей корня (если дерево пустое, вернуть 0).

```
int count1(struct BNode *r)
{
    int c = 0;
    if(r==0) return 0;
    if(r->l != 0) c++;
    if(r->r != 0) c++;
    return c;
}
```

2. Написать функцию, удаляющую левого сына корня, в предположении, что он является листом.

```
void dell(struct BNode *r)
{
    if(r==0 || r->l==0) return;
    free(r->l);
    r->l = 0;
}
```

Задачи.

1. Написать функцию, принимающую в качестве параметра дерево и возвращающую указатель на самого правого внука корня (если у корня нет внуков, вернуть 0).

2. Написать функцию, принимающую в качестве параметра дерево и возвращающую число внуков корня (если дерево пустое или у корня нет внуков, вернуть 0).

3. Написать функцию, переставляющую корневой узел и его левого сына (не перестановкой данных, а изменением указателей; переставляются только эти узлы, их поддеревья должны остаться, где и были).

4. Написать функцию, осуществляющую левый поворот в корне (см. рис. 2).

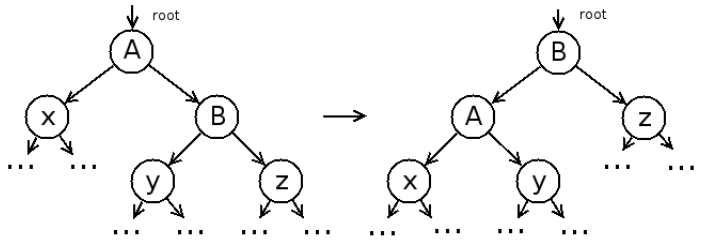


Рис. 2

5. Написать функцию, принимающую в качестве параметра дерево и возвращающую указатель на самый левый узел.

6. Написать функцию, принимающую в качестве параметра дерево и возвращающую указатель на самый левый лист.

7. Написать функцию, принимающую в качестве параметра дерево и возвращающую указатель на второй лист слева.

8. Написать функцию, вставляющую в дерево левого сына у самого левого узла. Данные нового узла передаются в качестве параметра.

9. Написать функцию, удаляющую самый левый узел (его правый сын становится левым сыном его отца).

10. Написать функцию, удаляющую самый левый лист.

Иногда бывает удобно в узле дерева, кроме указателей на сыновей, иметь указатель на отца (у корня этот указатель равен 0). Такие деревья мы будем называть U-деревьями. Если проводить аналогию со связными списками, U-дерево соответствует двусвязному списку.

11. Написать тип узла U-дерева и решить для него задачи 3, 4 и 8–10.

12. Написать функцию, принимающую U-дерево и указатель на какой-то его узел и удаляющую узел, на который он указывает. Можно предполагать, что у удаляемого узла нет одного из сыновей, а другой должен встать на место удаляемого узла.

46.2 Деревья с произвольным числом сыновей у каждого узла

Наряду с двоичными деревьями в программировании рассматриваются и произвольные деревья, каждая вершина которых может иметь произвольное число сыновей.

Такие деревья обычно представляются теми же по структуре узлами, что и двоичные деревья, но смысл указателей меняется. Теперь один указатель у узла указывает на его первого (самого левого) сына, он называется указатель вниз, а другой — на соседнего брата справа, и он называется указатель вправо.

Приведем необходимые описания:

```
struct ANode
{
    int d;
    struct ANode *s, *b;
};
```

Таким образом, набор братьев представлен по-существу связным списком; отличие от связного списка состоит в наличии дополнительного указателя вниз.

Такой способ трактовки указателей, содержащихся в узле, позволяет представлять не только деревья, но и (упорядоченные) леса, т. е. наборы деревьев (их корни считаются братьями).

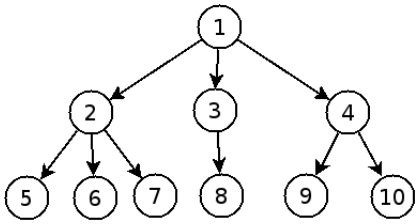


Рис. 1

Если рисовать узлы дерева как прямоугольники, внутри которых содержатся поля, то дерево на рис. 1 будет представлено структурой на рис. 2.

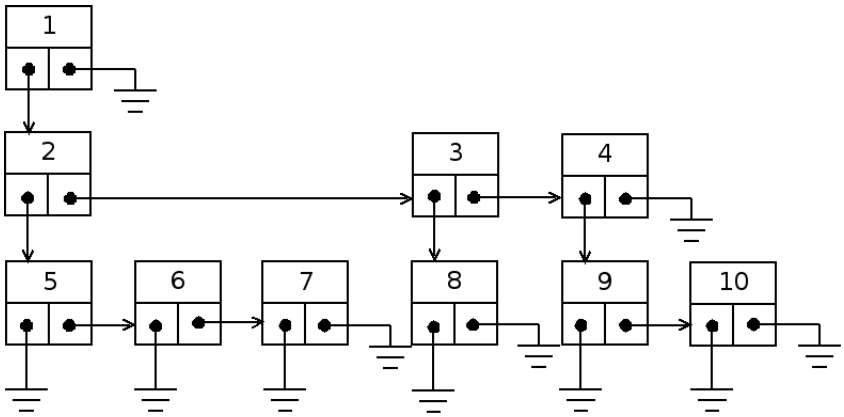


Рис. 2

Создать новое произвольное дерево можно, как и двоичное, при помощи специальной функции, создающей узел произвольного дерева:

```
struct ANode *g(int d, struct ANode *s, struct ANode *b)
{
    struct ANode *p = (struct ANode *) malloc(sizeof(struct ANode));
    p->d = d;
    p->s = s;
    p->b = b;
    return p;
}
```

Дерево на рис. 1 может быть создано при помощи выражения

```
g(1, g(2, g(5, 0, g(6, 0, g(7,0,0))), g(3, g(8,0,0), g(4, g(9, 0, g(10,0,0)), 0))), 0)
```

Дерево с произвольным числом потомков у каждого узла удобно печатать в том виде, в котором Windows показывает дерево папок и файлов в программах с графическим интерфейсом. Приведем здесь функцию, которая печатает такие деревья.

```
void print(struct ANode *r, int offset)
{
    if(r==0) return;
    for(int i = 0; i<offset; i++)
        printf(" ");
    printf("%d\n", r->d);
    print(r->s, offset+3);
    print(r->b, offset);
}
```

Если применить эту функцию к дереву, изображенному на рис. 1, будет напечатано следующее:

```
1
    2
        5
```

```
        6
        7
    3
        8
    4
        9
        10
```

В качестве примера рассмотрим задачу: написать функцию, возвращающую число сыновей корня.

```
int count(struct ANode *r)
{
    int c = 0;
    if(r==0) return 0;
    r = r->s;
    while(r!=nullptr)
    {
            c++;
            r = r->b;
    }
    return c;
}
```

Задачи.

1. Написать функцию, возвращающую число внуков корня.

2. Написать функцию, возвращающую указатель на первого сына корня с данными d1, у которого есть сын с данными d2 (d1 и d2 — параметры функции).

3. Написать функцию, возвращающую указатель на узел, полученный из корня последовательными перемещениями вниз в сына с заданным номером(последовательность номеров хранится в массиве, который передается в качестве параметра).

4. Написать функцию, производящую один шаг уплощения (удаление всех сыновей корня; внуки корня становятся его сыновьями, причем их порядок сохраняется).

46.3 Обход дерева

В программировании часто возникает потребность перебрать один за другим все узлы некоторого дерева и предпринять определенные действия для каждого из них. Такой перебор и называется обходом дерева. Пример — распечатка узлов дерева, которая уже встречалась нам на прошлых занятиях. Также обход дерева встречается в задачах поиска.

В зависимости от порядка перебора узлов дерева различают обход в глубину и обход в ширину. Обход в глубину лучше всего описывается рекурсивно: сначала мы обходим левое поддерево, затем посещаем корень, и, наконец, обходим правое поддерево, причем обход поддеревьев выполняется рекурсивно с помощью того же самого алгоритма. Такой алгоритм начинает обработку узлов дерева с самого левого узла, и затем обрабатывает их слева направо, т. е. если есть два узла в дереве, то тот из них, который расположен левее, будет обработан раньше.

Номера вершин дерева при таком способе обхода даны на рис. 1:

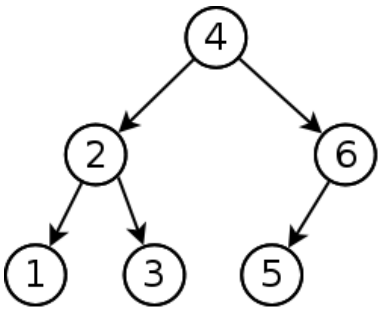


Рис. 1

Иногда рассматриваются и другие варианты обхода дерева в глубину, например, сначала обходим левое поддерево, затем правое, затем посещаем корень. Номера вершин дерева при этом способе обхода даны на рис. 2:

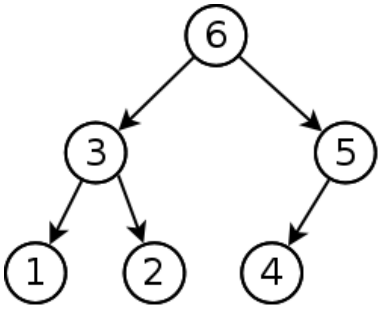


Рис. 2

Наиболее просто обойти дерево в глубину, используя рекурсию. Например, функция, подсчитывающая число узлов дерева, может выглядеть так:

```
int count(struct BNode *p)
{
    if(p==0) return 0;
    return 1+count(p->l)+count(p->r);
}
```

Однако, существуют способы, позволяющие обходить дерево в глубину и без использования рекурсии. Правда, для этого нам потребуется стек.

В нашем случае стек должен хранить последовательность указателей на узлы дерева.

Тогда та же самая функция, подсчитывающая число узлов дерева, в нерекурсивном варианте будет выглядеть так:

```
int count1(struct BNode *p)
{
    int c = 0;
    struct Stack S;
    struct BNode *t;
    if(p==0) return 0;
    push(&S, p);
    while(!empty(&S))
    {
        c++;
        t = pop(&S);
        if(t->l != 0) push(&S, t->l);
        if(t->r != 0) push(&S, t->r);
    }
    return c;
}
```

Как уже говорилось выше, существует также способ обхода дерева в ширину. В этом случае узлы дерева просматриваются по уровням от корня к листьям, причем на каждом уровне узлы перебираются слева направо. Номера вершин дерева при обходе в ширину даны на рисунке 3:

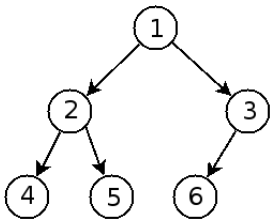


Рис. 3

Для обхода дерева в ширину рекурсия совершенно непригодна. Нерекурсивный же метод очень похож на обход в глубину, только стек заменяется на очередь — другую структуру данных, которая, как и стек, хранит последовательность элементов, но вставляются они с одного конца, а берутся с другого.

Чтобы использовать обход в ширину вместо обхода в глубину, достаточно в функции count1 заменить Stack на Queue. Обычно обход в ширину используется тогда, когда нужно найти узел дерева с заданными свойствами, ближайший к корню.

Обход дерева в глубину тесно связан с методом поиска, который называется «перебор с откатом».

Предположим, нам нужно найти решение некоторой задачи, которое сводится к последовательному выбору одного варианта из набора имеющихся, т. е. нам надо осуществить выбор одной возможности из некоторого набора; в зависимости от сделанного нами выбора на первом шаге, нам предоставляется новый набор возможностей, из которых мы опять должны сделать выбор одной, и так далее, пока набор сделанных нами выборов не будет удовлетворять условию задачи.

Типичным примером такого рода задачи поиска может служить задача поиска выхода из лабиринта. Изначально мы стоим на некотором перекрестке и должны выбрать направление движения. В зависимости от нашего выбора мы попадаем на новый перекресток и там опять должны выбирать направление, и так до тех пор, пока не найдем выход или не убедимся, что его нет.

По такой системе возможностей можно построить следующее дерево: сыновьями корня будут те возможности, которые имеются на первом шаге, а далее каждая вершина представляет собой последовательность актов выбора, которые нужно совершить, чтобы в нее попасть, и ее сыновьями будут те возможности, которые имеются в соответствующей ей ситуации.

Метод поиска, называемый «перебор с откатом», и состоит в обходе полученного дерева в глубину, до тех пор, пока мы не найдем интересующую нас вершину.

Задачи.

Написать функции, принимающие указатель на корень дерева:

1. total, возвращающую сумму всех данных в узлах дерева.

2. height, возвращающую высоту дерева.

3. min, возвращающую минимальное значение данных в узлах дерева.

4. eval, вычисляющую выражение, заданное данным двоичным деревом (числа в листьях рассматриваются как операнды, а в промежуточных узлах — как коды операций 1 +, 2 −, 3 *, 4 /).

5. find, принимающую второй параметр d (целое число), и возвращающую указатель на любой узел, содержащий данные d, если такие узлы в дереве есть, и nullptr, если таких узлов нет.

6. destroy, уничтожающую все узлы дерева, т. е. освобождающую занимаемую ими память.

7. copy, строящую копию дерева и возвращающую указатель на корень копии. Написать также функцию ucopy, копирующую аналогичным образом U-деревья.

8. reflect, меняющий дерево на его зеркальное отражение (правые и левые сыновья каждого узла меняются местами).

9. alt_sum, возвращающую сумму, в которую данные левых сыновей своих отцов входят с плюсом, а правых — с минусом (данные корня в эту сумму не входят вообще).

10. Написать функцию input, считывающую двоичное дерево с клавиатуры и возвращающую указатель на его корневой узел. Дерево должно быть записано в виде <данные корня>(<левое поддерево>, <правое поддерево>) Отсутствующее дерево обозначается символом «.».

Реализовать метод поиска «перебор с откатом» для следующих задач:

1. Лабиринт задан массивом maze типа int[50][50] (элемент 0 означает проход, 1 — стенка). Написать процедуру, принимающую исходную позицию в лабиринте (два целых числа) и печатающую путь, состоящий из соседних пустых клеток (имеющих общую сторону), начинающийся с заданной позиции и оканчивающийся на границе лабиринта (т. е. в клетке, одна из координат которой равна 0 или 49).

2. Написать программу, ищущую обход конем шахматной доски. Конь должен начинать свое движение в поле a1, переходить с одной клетки на другую по обычным правилам и посещать каждую клетку ровно один раз.

3. Написать программу, ищущую такую расстановку восьми ферзей на шахматной доске, при которой никакие два из них не бьют друг друга.

47 Деревья поиска

Очень часто в программировании возникает задача построения ассоциативного массива, т. е. массива, индексами которого являются либо вовсе не целые числа (например, строки или вещественные числа), либо целые числа из очень широкого диапазона, при том, что используется реально лишь очень небольшая часть индексов, так что заводить отдельную ячейку под каждое возможное значение индекса нецелесообразно.

В дальнейшем индекс в ассоциативном массиве, по которому ищется элемент, будем называть ключом, а сам элемент — данными. Термин «ключ» взят из теории баз данных.

Самыми простыми решениями задачи построения ассоциативного массива являются списки и массивы пар (ключ, данные). Однако эти решения обладают рядом недостатков.

Списки обеспечивают быструю вставку и удаление элементов из произвольной позиции (при условии, что мы знаем указатель на элемент в требуемой позиции), но поддерживают только линейный поиск, что в большинстве случаев неприемлемо медленно.

Массивы могут обеспечить быстрый (двоичный или интерполяционный) поиск, но для этого массив должен быть упорядочен, и вставка в упорядоченный массив требует сдвига большого числа элементов, что также неприемлемо медленно.

Дальнейшее развитие технологии предлагает использовать для построения ассоциативных массивов деревья. В общем существует два основных способа использования деревьев для организации ассоциативных массивов.

48 Деревья поразрядного поиска

Первый способ — так называемые деревья поразрядного поиска. В основе данной технологии лежит следующая идея. Предположим, у нас имеется функция, которая по любому допустимому значению x ключа и неотрицательному целому номеру n выдает значение n-го бита x, причем если все биты у двух ключей совпадают, то такие ключи также совпадают. Нумерация битов начинается с 0, причем 0-й — это младший бит (для целых чисел). Если ключи представляют собой целые числа, эта функция устроена очевидным образом; если ключи представляют собой вещественные числа или строки, она устроена чуть сложнее; на самом деле, такая функция существует для любого типа ключа, который можно использовать в компьютере, поскольку если ключ хранится в памяти, то он (с точностью до некоторой процедуры, называемой сериализацией — нужно только для связных структур вроде списков и т. д.) представляет собой последовательность битов.

Дерево поразрядного поиска — двоичное дерево, каждый узел которого содержит пару из ключа и данных, а также указатели на сыновей, в рамках данной технологии называющихся не «правый» и «левый», а «нулевой» и «единичный». В корневом узле может храниться любой ключ. В «нулевом» поддереве корневого узла содержатся только ключи, нулевой по номеру бит которых (в смысле функции из предыдущего абзаца, т. е. младший бит) равен 0, в «единичном» — 1.

Далее, в соответствии с вышесказанным, в «нулевом» сыне корня может храниться любой ключ, младший бит которого равен 0. В его «нулевом» поддереве встречаются только ключи, у которых как младший, так и следующий за ним по старшинству биты имеют значение 0, а в «первом» поддереве — только те ключи, у которых последние два бита имеют значение 1 и 0.

Итак, на глубине i (корень имеет глубину 0) идет разделение ключей по значению i-го бита (в соответствии с функцией, выдающей значение битов по номеру). Никаких других ограничений на ключи нет; таким образом, у ключа, стоящего на глубине i, фиксированы только i младших битов, а все остальные могут иметь любые значения.

Это свойство позволяет перенести на такие деревья алгоритм двоичного поиска в несколько видоизмененном варианте (по значению битов, начиная с младших), и время, затрачиваемое на основные операции с таким деревом (вставка/поиск/удаление пары) определяется не количеством элементов в дереве, а его высотой, которая, в свою очередь, ограничена максимальной длиной ключа в битах.

Для ключей потенциально небольшой длины (например, целых или вещественных чисел, или не очень длинных строк) такое решение задачи о построении ассоциативного массива вполне приемлемо.

Задачи.

0. Написать функцию digit с двумя целыми параметрами x и n, возвращающую значение n-го бита числа x.

1. Написать структуру DSNode узла дерева поразрядного поиска. В ней должны быть следующие поля: ключ (в данном разделе всегда будет целое число типа int), данные (тоже целое число) и указатели на сыновей.

Все последующие задачи требуют написания функций, первый параметр которых — указатель на корневой узел дерева (или указатель на указатель, если указатель на корневой узел дерева может меняться в результате работы данной функции). В тексте задач оговариваются только остальные параметры, если они есть.

2. print, печатающую дерево (как обычно, но для каждого узла печатаются ключ и данные).

3. search, принимающую ключ в качестве параметра и возвращающую указатель на узел дерева, содержащий этот ключ. Если такого узла в дереве нет, метод должен вернуть 0.

4. insert, вставляющий новую пару (ключ, данные). Подсказка: проще всего сделать это при помощи рекурсии.

5. del, удаляющую узел с заданным ключом.

6. Написать программу, которая вводит с клавиатуры имя файла, в котором записана последовательность целых чисел через пробел, и выводит таблицу, в которой указано, сколько раз в файле встречается каждое число (не обязательно выводить встречающиеся в файле числа по порядку).

7. searchMask, принимающую ключ и маску в качестве параметров. Эта функция должна осуществлять поиск по ключу с маской. Считаются известными только те биты ключа, для которых соответствующие биты маски имеют значение 1. Значения таких битов берутся из ключа-параметра функции. Значения остальных битов ключей в узлах могут быть любыми. Этот метод должен печатать на экран все имеющиеся в дереве пары (ключ, данные) с подходящими ключами.

8*. check, проверяющую дерево поразрядного поиска на правильность.

49 Деревья двоичного поиска

Наряду с деревьями поразрядного поиска, задача хранения и поиска информации имеет и другое решение, также построенное с использованием деревьев, но несколько на другом принципе. Речь идет о деревьях двоичного поиска.

Такие деревья точнее соответствуют идее двоичного поиска. Как и у деревьев поразрядного поиска, в каждом узле таких деревьев имеется пара из ключа и данных; однако принцип записи этих пар в узлы дерева несколько отличается.

Как для построения дерева поразрядного поиска была необходима функция, возвращающая значение бита ключа с определенным номером, так для построения двоичного дерева поиска необходимо, чтобы ключи можно было сравнивать. Как правило, для традиционных наборов ключей (целые, вещественные числа, строки) это так.

Однако, ключами в дереве двоичного поиска могут быть значения любого типа, лишь бы для них была определена функция, которая их сравнивает, т. е. принимает два параметра такого типа и выдает логическое значение (true, если первый параметр меньше второго, и false иначе). От этой функции требуется только, чтобы задаваемое ею отношение действительно являлось отношением линейного порядка. Если f(x,y) — такая функция, то от нее требуется выполнение следующих условий:

1) Для любого x значение f(x,x) ложно.

2) Для любых x и y f(x,y) и f(y,x) не могут одновременно быть истинными.

3) Транзитивность: если f(x,y) и f(y,z) оба истинны, то f(x,z) тоже истинно.

4) Линейность: для любых различных x и y, всегда f(x,y) или f(y,x) истинно.

Здесь имеет место удивительное явление: смысл задаваемого этой функцией отношения порядка не играет никакой роли. Важно только, чтобы были выполнены вышеприведенные 4 условия, и чтобы функцию f можно было относительно просто вычислить; имеет ли при этом задаваемое такой функцией отношение порядка вообще какой-либо смысл для тех объектов, которые мы беремся сравнивать, совершенно не важно! Например, если мы разрабатываем класс множества и хотим его объекты использовать в качестве индексов ассоциативного массива, первый вариант упорядочения, приходящий в голову — упорядочение множеств по включению. Однако, такой способ упорядочивания множеств, хотя и является естественным для множеств, обеспечивает лишь частичный (не линейный) порядок. А вот если определить порядок на множествах, например, как лексикографический порядок для упорядоченных по возрастанию наборов элементов множества, то он будет линейным, и следовательно, подходит для построения ассоциативного массива с индексами-множествами, несмотря на то, что такой порядок естественным для множеств не является, т. е. его смысл для двух сравниваемых множеств кажется очень странным.

Все дальнейшее будет сказано в предположении, что для интересующего нас типа ключей у нас уже имеется такая сравнивающая функция (удовлетворяющая всем 4 условиям). Мы будем обозначать ключи латинскими буквами и писать между ними неравенства, имея в виду, что либо они принадлежат традиционному набору (числа или строки), тогда имеется в виду обычное отношение порядка; либо для них определена некоторая сравнивающая функция, и отношение порядка определяется ею (хотя мы и пишем $x < y$ вместо «f(x,y) истинно»).

Двоичное дерево, в каждом узле которого стоит пара из ключа и данных, называется двоичным деревом поиска, если в каждом его узле выполняется следующее условие: все ключи в левом поддереве меньше, а в правом — больше ключа в данном узле (повторяющиеся ключи не допускаются по очевидным причинам).

Свойство дерева двоичного поиска позволяет организовать поиск в таком дереве по тому же принципу, как и двоичный поиск в отсортированном массиве (почему собственно такое дерево и называется деревом двоичного поиска). Проще всего описать такой поиск рекурсивно. Будет два параметра: указатель на узел, в поддереве которого мы ищем нужный ключ, и само значение искомого ключа. Если указатель на узел равен 0, надо вернуть 0 — ничего не найдено; если же искомый ключ равен значению ключа в узле, на который указывает указатель — вернуть этот указатель; иначе, если искомый ключ меньше значения ключа в узле, на который указывает указатель — искать (при помощи той же функции поиска) в левом поддереве, больше — в правом.

Вставка информации, т. е. новой пары (ключ, данные), в двоичное дерево поиска выполняется похожим образом, только для указателя на узел нужно передавать его адрес (в итоге, первый параметр функции будет иметь тип указатель на указатель на узел), и будет один дополнительный параметр — значение данных. Если указатель на узел равен 0 — присваиваем ему указатель на новый узел, построенный с помощью malloc и заполняем поля надлежащими значениями — это и есть вставка. Если же нет — сравниваем ключ, который хотим вставить, со значением ключа в том узле, на который указывает указатель-параметр, и в зависимости от результатов этого сравнения вставляем новую пару в правое или левое поддерево при помощи рекурсивного вызова той же функции. В итоге, как и в дереве поразрядного поиска, новые пары (ключ, данные) вставляются только в листья нашего дерева. Правда, последующие вставки могут привести к тому, что эти конкретные узлы перестанут быть листьями.

Удаление узла по значению ключа — значительно более сложная функция. Здесь уместно выделить несколько случаев. Первый случай — когда у удаляемого узла имеется не более одного сына. В этом случае все просто — узел удаляется, а его единственный сын (если такой был) вместе со всем своим поддеревом подтягивается наверх, становясь сыном отца удаляемого узла с той стороны, с которой удаляемый узел был прикреплен к своему отцу.

Второй, более сложный случай — когда у удаляемого узла есть оба сына. В этом случае мы ищем самого левого потомка правого сына удаляемого узла. Этот потомок обладает двумя важными свойствами: во-первых, он содержит

следующее по величине значение ключа за удаляемым, и, во-вторых, у него нет левого сына. Мы переписываем его ключ и данные в тот узел, который собирались удалять, и удаляем его (так как у него нет левого сына, см. первый случай).

Кроме того, важно также рассматривать случаи, удаляем ли мы корень или нет.

Есть и другие способы удаления узла с двумя сыновьями из двоичного дерева поиска, но по соображениям, которые мы изучим в дальнейшем, этот способ предпочтительнее.

Задачи.

1. Написать структуру узла дерева двоичного поиска.
2. Написать функцию поиска узла по значению ключа.
3. Написать функцию вставки пары ключ, значение.
4. Написать функцию удаления по значению ключа.
5*. Написать функцию проверки правильности дерева двоичного поиска.

50 АВЛ-деревья

Определение дерева двоичного поиска позволяет перенести на такие деревья алгоритм двоичного поиска (откуда и происходит название таких деревьев), и время, затрачиваемое на основные операции с таким деревом (вставка/поиск/удаление пары) определяется не количеством элементов в дереве, а его высотой. В лучшем случае высота примерно равна $\log_2 N$, где N — число узлов дерева, т. е. число хранящихся в ассоциативном массиве элементов.

Однако, дерево двоичного поиска также имеет один существенный недостаток. Дело в том, что при фиксированном наборе ключей основное свойство дерева двоичного поиска не определяет структуру дерева однозначно, и в худшем случае такое дерево может вырождаться в список (например, у каждого узла нет левого сына). Так происходит, если мы добавляем элементы в ассоциативный массив в порядке возрастания ключей. В этом случае дерево двоичного поиска ничем не отличается от обычного связного списка со всеми его недостатками.

Для исправления такой ситуации было предложено понятие сбалансированного дерева. В самом жестком варианте дерево сбалансировано, если число узлов в левом и правом поддереве каждого узла отличается максимум на 1. Очевидно, высота такого дерева всегда приблизительно равна $\log_2 N$.

Однако, такой подход обладает недостатками упорядоченного массива — в дерево с такими требованиями очень трудно добавить новый узел; точнее, добавить просто, но это может привести к нарушению условия сбалансированности, а вот восстановить такую сбалансированность — очень трудно и долго.

Поэтому в итоге решение проблемы состоит в том, чтобы ослабить требование сбалансированности. Известно два варианта такого ослабления.

Первый вариант был предложен в 1962 году Г. М. Адельсоном-Вельским и Е. М. Ландисом, по первым буквам фамилий которых и получили свое название АВЛ-деревья. Их предложение состояло в том, чтобы ослабить требование сбалансированности, заменив его АВЛ-условием: высоты поддеревьев любого узла должны отличаться не более, чем на 1.

Оказывается, во-первых, этого достаточно, чтобы высота дерева не превосходила $C \log_2 N$, где N — число узлов дерева и C — ни от чего не зависящая константа. Во-вторых, после добавления или удаления из дерева одного узла справедливость АВЛ-условия может быть восстановлена за время, ограниченное другой константой, умноженной на высоту дерева.

Задачи.

1. Написать структуру AVLNode узла АВЛ-дерева. В ней должны быть, кроме ключа, данных (для простоты и то, и другое — целые числа) и указателей на сыновей, следующие поля: указатель на родительский узел и высота (целое число). Также реализовать функции (единственный параметр — указатель на узел): correct_h, вычисляющую высоту узла на основании высот сыновей и записывающую результат в соответствующее поле, defect_h, возвращающий разность высот левого и правого сыновей, и print, печатающий данные узла (указатель на него и значения полей).

2. Написать функцию print_all, печатающую дерево.

3. Написать функции rotate_left (левый поворот) и rotate_right (правый поворот). Параметры — указатель на указатель на корень дерева и указатель на указатель на узел, в котором производится поворот (чтобы менять значение этих указателей в соответствии с преобразованием поворота). Не забыть также поменять правильным образом указатели в дереве. Рассмотреть случаи, когда указатель, переданный в качестве параметра, является частью дерева (т. е. это сам root или указатель, являющийся полем какого-либо узла дерева) или нет.

4. Написать функции correctm2 и correct2, принимающие в качестве параметров указатель на указатель на корень дерева и указатель на указатель на узел дерева и меняющие поддерево этого узла при помощи поворотов так, чтобы оно удовлетворяло АВЛ-условию. Можно предполагать, что все нижестоящие узлы удовлетворяют АВЛ-условию, а в самом узле, на который указывает параметр, разность высот левого и правого поддеревьев равна -2 и 2 соответственно. Подсказка: имеется всего два случая, и в одном из них нужен один дополнительный поворот, и затем поворот, нужный в обоих случаях.

5. Написать функцию rebalance, исправляющую АВЛ-свойство дерева, нарушенное вследствие изменения высоты одного из узлов на 1. Параметрами являются указатель на указатель на корень дерева и указатель на указатель на самый глубокий узел, высота которого изменилась.

6. Написать функцию insert, вставляющую новую пару ключ-данные.

7. Написать функции для удаления корня/правого/левого сына своего отца, не имеющего правого/левого сына (всего 6 вариантов).

8. Написать функцию del, удаляющую узел с заданным ключом. Подсказка: если у удаляемого узла имеются оба сына, мы находим узел x, содержащий следующее по величине (за удаляемым) значение ключа, и ключ и данные этого узла переписываем в тот узел, который хотели удалить, и вместо него удаляем узел x (почти как в деревьях двоичного поиска, только потом еще вызвать rebalance).

51 B-деревья

B-деревья — еще один вид сбалансированных деревьев. B-деревья изобрели Байер (Bayer) и МакКрейт (McCreight) в 1970 году.

51.1 Назначение B-деревьев.

Известно, что доступ к жесткому диску осуществляется гораздо медленнее, чем к оперативной памяти. Кроме того, за одну операцию чтения или записи на жесткий диск можно считать или записать только целое число блоков (блок — последовательность байтов определенной фиксированной длины, зависящей от операционной системы, объема жесткого диска и типа файловой системы, т. е. способа организации хранения файлов, на нем).

Поэтому при разработке способа хранения информации на жестком диске обычно минимизируют количество раз доступа к нему, исходя из того, что практически любая обработка полученного с диска блока выполняется гораздо быстрее, чем само считывание этого блока.

Поскольку малый размер узла дерева не дает никаких преимуществ в скорости работы (все равно меньше блока за один раз не считать), у B-деревьев размер одного узла примерно равен одному блоку. Это позволяет хранить в одном узле большое количество ключей и указателей на сыновей, что обеспечивает высокую степень ветвления; в свою очередь, это приводит к очень маленькой высоте дерева даже при огромном числе узлов. Но скорость работы деревьев поиска пропорциональна высоте дерева, и в итоге B-дерево для стандартных операций по его обработке (добавление новой пары ключ/данные, поиск, удаление) требует считывания или изменения очень небольшого числа блоков на диске, что существенно повышает производительность B-деревьев по сравнению с другими деревьями поиска, если дерево хранится на диске.

При всем том, B-дерево остается деревом поиска, т. е. в каждом его узле для каждого его сына хранится минимальное значение ключей соответствующего поддерева. Это позволяет при поиске информации в B-дереве во время анализа конкретного узла определить, в поддереве какого сына лежит (если он вообще имеется в дереве) конкретный ключ.

Однако, для того, чтобы B-дерево не вырождалось, на него накладывается дополнительное требование: глубина всех листьев должна быть одинакова. Чтобы выполнить это требование, и вместе с тем, иметь возможность быстро производить над деревом стандартные операции, вводится еще одно условие: число сыновей каждой вершины должно лежать в пределах от $N/2$ (включая) до N (не включая), где N — максимально возможное число записей, описывающих одного сына, которое может быть записано в один узел. Таким образом, в каждом узле имеется массив для описания его сыновей, заполненный не менее, чем наполовину, но не полностью. Единственным исключением из этого правила является корневой узел, в котором должно быть не менее двух записей, если в дереве кроме корневого узла имеются другие узлы. Удобно, чтобы N было четным.

В B-деревьях, которые используются в реальности, данные хранятся только в листьях; в промежуточных узлах хранятся только ключи и указатели на сыновей. Однако, для упрощения ситуации, мы рассмотрим вариант B-дерева с одинаковым типом узла, не зависящим от его положения в дереве, т. е. от того, является он листом или нет.

Пример B-дерева с $N = 4$ высоты 3 (в реальности N велико — порядка сотен или даже тысяч).

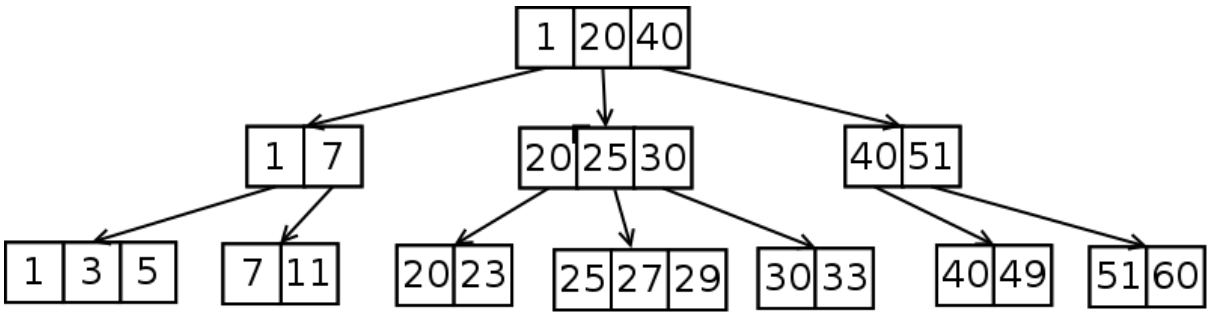


Рис. 1

Узел B-дерева описывается структурой BTNode (см. ниже).

```
#define N 4 /* размер массива записей о сыновьях в каждом узле */

struct BTNode; /* тип узла дерева */

struct Item /* тип записи об одном сыне */
{
    int key;
    int data;
    struct BTNode *down;
};

struct BTNode
{
    int n; /* реальное число сыновей */
    struct Item ar[N];
};

struct BTTree /* собственно тип дерева */
{
    struct BTNode *root; /* корневой узел */
    int height; /* высота дерева */
};
```

Задачи.

Во всех этих задачах B-дерево представлено в программе переменной структурного типа BTTree. Соответственно, все функции, работающие с B-деревьями (а не только с отдельными узлами), получают в качестве первого параметра указатель на такую переменную.

1. Написать функцию, строящую пустое дерево (в отличие от деревьев двоичного поиска и деревьев поразрядного поиска, в пустом B-дереве имеется один (корневой) узел без сыновей, поле n в этом узле равно 0).

2. Написать функцию print, печатающую дерево (форма вывода B-дерева такая же, как и произвольного дерева, т. е. дерева с произвольным числом сыновей у каждого узла).

3. Написать функцию search с параметром-искомым ключом и возвращающий указатель на элемент данных, соответствующий указанному ключу (если такого нет, возвращается нулевой указатель). Для поиска заданного ключа нужно начать с корня и определить, в каком из поддеревьев его сыновей лежит ключ (это можно сделать, найдя последний ключ в массиве данного узла, который не превосходит искомого; если таких нет, значит ключ меньше минимального ключа в дереве). Затем перейти в это поддерево, пользуясь указателем на сына в найденной записи, и т. д. до тех пор, пока не окажемся в листе, а там уже можно искать сам ключ — если его там нет, значит, его вообще нет в дереве.

4. Написать функцию insertB, принимающую в качестве параметров указатель p на узел, номер (целое число) i и запись о сыне x (параметр типа Item), и вставляющую элемент x в массив ar записей узла, на который указывает p, в позицию i (все элементы с номерами i и более сдвигаются на одну позицию вправо, общее число сыновей увеличивается на 1). Можно предполагать, что до вставки число сыновей узла $*p$ было меньше N.

5. Написать функцию split, которая принимает в качестве параметра указатель p на узел, заводит новый узел и переписывает в него вторую половину элементов массива ar из узла, на который указывает p. Можно предполагать, что узел $*p$ был заполнен до конца, т. е. в нем было N записей о сыновьях. Эта функция должна устанавливать у обоих узлов (нового и старого) правильное число сыновей и возвращать указатель на новый узел.

6. Написать функцию insertR, принимающую в качестве параметров указатель на узел (в дальнейшем этот узел называется текущим), ключ, данные и высоту текущего узла и вставляющую указанную пару (ключ, данные) в поддерево указанного узла. Этот метод должен возвращать 0, если вставка в текущий узел не привела к его заполнению до конца (N сыновей), и указатель на новый узел, если после вставки текущий узел был заполнен до конца; в этом случае insertR должна расщеплять текущий узел на два при помощи split из предыдущей задачи и вернуть указатель на новый узел, содержащий вторую половину записей из текущего узла. Существует два случая (распознаются по значению высоты текущего узла, переданной как параметр): вставка в лист и в промежуточный узел. Если мы вставляем новую пару из ключа и данных в лист, мы должны найти место для нее (а ключи в записях о сыновьях должны быть упорядочены), осуществить собственно вставку при помощи insertB (ключ и данные берутся из параметров, указатель на сына — 0), и проверить, до конца ли заполнен тот лист, куда мы осуществляли вставку. Если нет, вернуть 0, иначе вызвать split и вернуть ее результат. Если же мы вставляем пару не в лист, нужно сначала определить того сына, в поддерево которого будет осуществляться вставка (а это первый сын, если вставляемый ключ меньше ключа первого сына, или последний сын из тех, ключи которых не больше вставляемого; в первом из этих двух случаев нужно записать вставляемый ключ на место ключа первого сына). Далее, нужно рекурсивно вызвать insertR для найденного поддерева для вставки туда новой пары. Если этот вызов вернул 0, на этом процесс вставки можно закончить; в противном случае нужно вставить новый указатель на сына в текущий узел сразу после того сына, в поддерево которого осуществлялась вставка. Ключ в этой новой записи о сыне должен браться из первой записи того узла, на который указывает результат insertR (это и будет минимальный ключ соответствующего поддерева). Наконец, должна быть та же проверка, что и при вставке в лист: до конца ли заполнен текущий узел. Действия, связанные с этой проверкой, тоже аналогичны: если нет, вернуть 0, иначе вызвать split и вернуть ее результат. На самом деле, все это может быть записано достаточно компактно; как это делается, см. далее.

Для этого нужно сделать следующее: сначала найти среди ключей текущего узла первый, который больше вставляемого; затем, подготовить переменную типа Item, которую мы будем вставлять (заполнить ее поля значениями вставляемых ключа и данных, указатель на сына установить в 0). Далее, нужно рассмотреть случай, когда текущий узел — не лист (параметр высоты > 1). В этом случае рекурсивно вызывается метод insertR для вставки новой пары в нужное поддерево. Затем, если мы вставляли ключ, который был меньше минимального ключа в текущем узле, нужно подправить значение минимального ключа в первом поддереве, а заодно и позицию, которую мы нашли в самом начале метода. Наконец, если рекурсивный вызов insertR вернул 0, нужно закончить, иначе нужно изменить ту переменную типа Item, которую мы подготовили для вставки, а именно записать в поле ключа первый ключ нового узла (указатель на который вернул рекурсивный вызов insertR), а в поле указателя на сына — сам результат insertR. На этом обработка случая, когда мы вставляем не в лист, закончена. Последнее, что надо сделать — надо вставить подготовленную переменную типа Item в текущий узел на подготовленную позицию (найденную в начале метода и возможно подправленную впоследствии) при помощи insertB, и после этого, если текущий узел не заполнен до конца, вернуть 0; если же он заполнен, разделить его на два при помощи split, и вернуть результат этого вызова split.

7. Написать функцию insert, принимающую в качестве параметров ключ и данные и вставляющую указанную пару (ключ, данные) в дерево при помощи вызова метода insertR. Если он вернул не 0, значит, ему пришлось разделить старый корень на два узла; это означает, что надо надстроить новый корень (с двумя записями — одна про старый корень, другая — про новый узел, т. е. результат insertR). Более подробное описание следует ниже. Нужно завести новый корень с двумя записями (одна соответствует старому корню: ключ берется из первой записи старого корня, данные — 0, указатель на сына — старое значение root; другая — результату вызова insertR: ключ берется из первой записи узла, на который указывает результат insertR, данные — 0, указатель на сына — результат insertR). При этом высота всего дерева (поле height) увеличивается на 1.

8. Написать функцию removeB, принимающую в качестве параметров указатель на узел и номер и удаляющую указанную запись из массива записей узла, на который указывает первый параметр (все записи с большими номерами сдвигаются на одну позицию в сторону уменьшения номеров, общее число сыновей уменьшается на 1).

9. Написать функцию join, принимающую в качестве параметров указатель p на узел и номер n и соединяющую два узла, на которые указывают записи под номерами n и $n+1$ в узле $*p$, в один. При этом все записи из сына узла $*p$ с номером $n+1$ переписываются в конец массива записей у сына узла $*p$ с номером n, у этого сына подправляется число сыновей, тот сын, из которого были взяты записи, удаляется при помощи free, и, наконец, запись с номером $n+1$ удаляется из узла $*p$ с помощью функции removeB из предыдущей задачи. Можно предполагать, что суммарное количество записей в обоих объединяемых узлах меньше N, но не меньше $N/2$.

10. Написать функцию pass_forward, принимающую в качестве параметров указатель p на узел и номер n и переписывающую последнюю запись из сына, на которого указывает n-я запись в $*p$ в сына, на которого указывает $n+1$-я (новая запись должна быть там первой в силу упорядоченности ключей). Вставить запись в новый узел можно при помощи insertB, а удалить из того узла, откуда она берется, можно при помощи removeB. Нужно также не забыть подправить значение ключа в записи того сына, куда попадает новая запись (чтобы в записи этого сына действительно оказался минимальный ключ соответствующего поддерева, который теперь будет равен ключу перемещенной записи).

11. Написать функцию pass_backward, принимающую в качестве параметров указатель p на узел и номер n и переписывающую первую запись из узла, на который указывает $n+1$-я запись в $*p$ в узел, на который указывает n-я.

Это делается аналогично тому, что происходило в предыдущей задаче.

12. Написать функцию correct_filling, принимающую в качестве параметров указатель p на узел и номер n и исправляющую нагрузку (т. е. число сыновей) узла, на которого указывает n-я запись в $*p$, в предположении, что она равна $N/2-1$. Самый простой способ это сделать — позаимствовать эту запись (при помощи одной из функций двух предыдущих задач) у одного из ближайших братьев, если это не приведет к недостаче записей у него (т. е. после этого у него останется не меньше $N/2$ сыновей). Если же этот способ не сработает, то можно объединить (при помощи join) сына с недостатком записей с соответствующим его братом, в результате чего мы получим узел с $N-1=(N/2-1)+(N/2)$ записями — проблема опять будет решена.

13. Написать функцию removeR, принимающую в качестве параметров указатель на узел, ключ и высоту узла, на который указывает первый параметр, и два указателя на логические переменные corr_key и corr_fill и удаляющую пару с указанным ключом из поддерева указанного узла. Механизм удаления следующий: как и в случае вставки, существуют два случая. Если нам надо удалить пару из листа, мы ищем в нем удаляемый ключ; если его нет, делать нечего. Иначе, пользуемся removeB и удаляем интересующую нас пару. Наконец, при помощи логических параметров, переданных через указатели, сигнализируем вызывавшей функции о двух обстоятельствах: corr_key устанавливается в true, если мы удаляли первую запись — тогда в вышестоящем узле надо подправить ключ в записи, описывающей того сына, с которым мы работали; corr_fill устанавливается в true, если в результате удаления в листе осталось меньше $N/2$ записей. Если мы удаляем пару из промежуточного узла, нужно завести две логические переменные, определить сына, из поддерева которого нужно удалить ключ (это будет последний сын, минимальный ключ поддерева которого не больше удаляемого ключа; если же таких сыновей нет вовсе, это означает, что удаляемый ключ меньше минимального ключа в дереве, значит, такого ключа в дереве нет, и делать нечего). Далее, нужно рекурсивно вызвать функцию removeR для этого сына.

Затем, нужно проанализировать те логические значения, которые функция вернула: если в поддереве изменился минимальный ключ, соответствующую запись (того сына, для которого была вызвана функция), нужно подправить, записав туда новый ключ из первой записи этого сына. Если же после удаления у этого сына образовался недостаток записей, это должно быть исправлено при помощи функции correct_filling. Наконец, делается то же, что и в конце первого случая: проверяем, изменился ли первый ключ (при исправлении ключа в записи сына, в поддереве которого было удаление, после removeR, если этот сын был первым) и не осталось ли в текущем узле (после вызова correct_filling) недостатка в сыновьях; в соответствии с результатами этих проверок устанавливаются параметры corr_key и corr_fill.

14. Написать функцию remove, принимающую в качестве параметров ключ и удаляющую пару с указанным ключом. Для этого достаточно вызвать функцию предыдущей задачи с указателем root (и естественно ключом, а также двумя логическими переменными), при этом на значения логических переменных можно не обращать внимания. Но после вызова removeR, нужно проверить следующее: если в корне осталась одна запись, и высота дерева больше 1 (т. е. кроме корня в дереве есть еще узлы), то нужно удалить корневой узел, указателю на корень присвоить указатель на единственного сына старого корня, и уменьшить высоту дерева (поле height) на 1. Разумеется, нужно делать это правильно (потому, что, если удалить корень сначала, то и указатель на его единственного сына будет потерян; здесь не обойтись без дополнительной переменной типа указатель на узел). Это действие похоже на удаление первого элемента из списка.

52 Красно-черные деревья

Красно-черные деревья представляют собой другое, впрочем, не менее популярное, решение задачи о построении балансирующегося двоичного дерева поиска. Они дают несколько более медленный поиск, но более быструю перебалансировку, чем АВЛ-деревья.

На самом деле, красно-черные деревья — это попытка превратить B-дерево с N=4 (см. предыдущий раздел) в двоичное дерево. В таком дереве имеются два типа узлов: узлы с двумя и с тремя сыновьями (в промежуточном варианте, во время работы функций вставки и удаления, возможны также узлы с четырьмя сыновьями).

Для указанного превращения можно воспользоваться представлением узла с тремя сыновьями как фрагмента двоичного дерева, состоящего из двух узлов (один из них является отцом другого), каждый из которых имеет, как положено узлу двоичного дерева, двух сыновей. Узел с четырьмя сыновьями представляется как конструкция из трех двоичных узлов: один из них является отцом оставшихся, которые друг другу являются братьями.

В таком виде функция поиска работает ровно также, как и для обычного двоичного дерева поиска. Однако, для поддержания сбалансированности необходимо знать, какие фрагменты полученного двоичного дерева произошли из каких узлов исходного B-дерева. Для этого у каждого узла вводится дополнительное поле, хранящее его «цвет», который может быть красным или черным. Правила раскраски узлов двоичного дерева следующие:

1) Узел исходного B-дерева с двумя сыновьями при преобразовании сохраняется как есть, и является черным.

2) Узел исходного дерева с тремя сыновьями представляется конструкцией из двух узлов, из которых один является отцом другого. В этом случае отец имеет черный цвет, а сын — красный.

3) Узел исходного дерева с четырьмя сыновьями представляется конструкцией из трех узлов, из которых один является отцом двух других. В этом случае отец имеет черный цвет, а оба сына — красный.

Задачи.

1. Написать структурный тип узла красно-черного дерева.
2. Написать структурный тип красно-черного дерева.
3. Написать функцию поиска узла по ключу для красно-черного дерева.
4. Написать функцию вставки пары (ключ, данные) (с необходимой перебалансировкой; можно переделать соответствующую функцию из B-деревьев) для красно-черного дерева.
5. Написать функцию удаления пары по значению ключа (тоже с необходимой перебалансировкой; можно опять же переделать соответствующую функцию из B-деревьев) для красно-черного дерева.

53 Гибридные деревья

Кроме вышеизложенных основных вариантов использования деревьев для решения задачи об ассоциативном массиве, существуют и гибридные варианты, например, деревья троичного поиска. Если ключи — строки, то такое дерево будет устроено следующим образом: у каждой вершины дерева имеются данные (символьного типа) и не более трех сыновей.

Если нам нужно найти данные по ключу-строке, нужно начать перемещение по дереву с корня, находясь в котором, анализируется первый символ ключа. Если его код меньше, чем код символа в корне, мы переходим в первого (левого) сына корня и продолжаем анализировать первый символ ключа. Если имеет место равенство, переходим во второго (среднего) сына корня, и далее рассматриваем второй символ ключа. Наконец, в оставшемся случае переходим к третьему (правому) сыну корня, продолжая рассматривать первый символ ключа.

Таким образом, мы перемещаемся вниз по дереву, в каждой его вершине рассматривая определенный символ ключа и сравнивая его с тем символом, который хранится в данной вершине. Если имеет место равенство, мы перемещаемся в среднего сына текущей вершины, переходя к рассмотрению следующего символа в ключе. Если же равенства нет, мы выбираем левого или правого сына в зависимости от того, в какую сторону имеем неравенство, и перейдя в него, продолжаем рассматривать тот же символ ключа, что и в текущей вершине.

Так мы движемся вниз по дереву до тех пор, пока не будут рассмотрены до конца все символы ключа. Та вершина, в которую мы при этом попадем, и будет содержать искомую информацию.

54 Декартовы деревья

Имеется и еще один вариант деревьев для решения задачи об ассоциативном массиве, так называемые декартовы деревья. Грубо говоря, это некая комбинация деревьев двоичного поиска с очередями по приоритетам. В узлах таких деревьев кроме ключей и данных имеется еще и приоритет, и если по расположению ключей это дерево двоичного поиска, то по приоритетам — соответствующая очередь. Смысл такого усложнения в том, что приоритет назначается новым узлам случайным образом и служит только для поддержания разветвленности дерева, как замена для (достаточно сложных) алгоритмов перебалансировки.

55 Хэш-таблицы

Последний способ построения ассоциативных массивов, который будет здесь рассмотрен, — хэш-таблицы. Суть этого подхода в том, чтобы завести массив списков пар из ключа и данных. Число элементов массива должно приблизительно равняться ожидаемому числу пар, которые нужно хранить. Для работы такой структуры самой важной деталью является так называемая хэш-функция. Она принимает значение ключа и выдает индекс в массиве того списка, в котором должны храниться пары с таким значением ключа. В идеале, каждая пара из ключа и данных должна храниться в своем элементе массива, но подобрать настолько хорошую хэш-функцию очень непросто. И если случится так, что у двух разных ключей значение хэш-функции одинаково, обе пары с такими ключами попадут в один и тот же элемент массива, и будут храниться в стоящем там списке (это и есть объяснение того, почему в каждом элементе массива хранится не одна пара, а список).

Задачи.

1. Завести хэш-таблицу на 1000 элементов массива (ключи и данные — целые числа).
2. Написать хэш-функцию для таблицы из предыдущей задачи.
3. Написать функцию поиска данных по ключу в хэш-таблице.
4. Написать функцию добавления новой пары в хэш-таблицу.
5. Написать функцию удаления пары из хэш-таблицы.

56 Задача поиска на примере игры «Быки и коровы»

Игра «Быки и коровы» состоит в следующем. Играют двое. Первый игрок загадывает четырехзначное число, все цифры которого различны. Цель второго — угадать это число за как можно меньшее число попыток. Одна попытка состоит в том, что второй игрок называет свою версию загаданного числа и получает в ответ два числа от 0 до 4 — числа быков и коров. Бык — это цифра в загаданном числе, совпадающая с цифрой в попытке, стоящая на правильном месте. Например, если загадано было число 2356, а названо в качестве попытки 4326, то число быков будет 2 (3 и 6). Корова — та цифра в попытке, которая встречается в загаданном числе, но не на правильном месте. В вышеприведенном примере корова будет одна (2). Так что в этом примере ответ на такую попытку будет звучать так: 2 быка и 1 корова. Ответ «4 быка» означает, что число угадано правильно.

Требуется:

1) Написать программу, загадывающую число (случайным образом), вводящую попытки пользователя его угадать, и отвечающую на них правильно (числа быков и коров), до тех пор, пока число не будет угадано.

2) Переделать программу п. 1) в угадывающую, т. е. выдающую попытки угадать число и вводящую ответы пользователя (числа быков и коров; на этот раз, число загадывает пользователь). При правильном алгоритме число должно угадываться не более, чем за 6–7 попыток.

57 Динамическое программирование

Очень часто задача поиска зависит от целого параметра или даже набора параметров. В таких случаях для решения задачи иногда удается свести задачу к аналогичной задаче, но с меньшими значениями параметров; при этом, для минимальных значений параметров решение задачи очевидно. Такой прием называется рекурсией. Однако, рекурсия имеет один недостаток — если применять ее необдуманно, она может приводить к многократному вычислению одних и тех же выражений, за счет чего эффективность решения задачи резко уменьшается (пример — вычисление чисел Фибоначчи). С этой проблемой можно побороться, запоминая промежуточные результаты, и используя запомненные данные там, где нужно повторно вычислять некоторые выражения. Все это вместе и называется динамическим программированием; этот прием часто используется для решения задач поиска и оптимизации.

Задачи.

1) Дан одномерный массив целых чисел. Требуется определить наибольшее значение суммы некоторого отрезка массива из подряд идущих элементов. При правильном алгоритме это делается за один проход по массиву и на каждый элемент тратится постоянное число операций. Далее, нужно переделать программу так, чтобы выдавать еще и индексы первого и последнего элементов, входящих в отрезок, дающий наибольшую сумму.

2) Задана квадратная матрица A из положительных вещественных чисел, содержащая коэффициенты перевода денег из одной валюты в другую (валюты соответствуют столбцам и строкам матрицы, и A_{ij} — сколько валюты номер i можно купить за единицу валюты номер j). Требуется найти для целого числа n максимальную выгоду за не более чем n операций обмена (они должны начинаться и оканчиваться одной и той же валютой).

3) Пусть имеется две матрицы A размера $m \times n$ и B размера $k \times l$. Тогда их можно перемножить, если $n = k$, и в этом случае для умножения таких матриц требуется выполнить mnl операций умножения чисел. Кроме того, произведение матриц ассоциативно, т. е. для любых матриц A, B, C подходящих размеров выполняется равенство $(AB)C = A(BC)$. Требуется ввести два массива из чисел строк и столбцов матриц соответственно, и определить наименьшее количество операций умножения чисел, которое надо вычислить, чтобы найти произведение таких матриц. Т. е. имеется набор матриц $A_1, \ldots, A_n$, для которых известны размеры, и требуется найти наименьшее число операций умножения чисел, которые нужно вычислить, чтобы найти произведение $A_1 A_2 \ldots A_n$ (при разных способах расстановки скобок число операций может быть разным; полезно также учесть, что произведение матриц некоммутативно, т. е. менять можно только расстановки скобок, но никак не порядок матриц в произведении).

4) Дан набор длин слов, которые нужно разместить на нескольких строках и оценочная функция, которая принимает набор длин слов, помещенных на одной строке, и выдает число, оценивающее качество данной строки (качество всего текста оценивается суммой значений такой функции на всех строках). При этом пустые строки не допускаются, и порядок слов менять нельзя. Найти наибольшее качество размещения указанного набора слов на нескольких строках.

58 Решение систем линейных уравнений методом Гаусса

Система линейных уравнений имеет общий вид

$$\begin{cases} a_{11}x_1 + \ldots + a_{1n}x_n = & b_1 \\ \vdots & \vdots \\ a_{m1}x_1 + \ldots + a_{mn}x_n = & b_m \end{cases}$$

Здесь a_{ij} и b_i — известные коэффициенты, а x_j — неизвестные, которые нужно найти.

Доказано, что решение такого вида системы всегда устроено следующим образом: либо решений нет вообще, либо имеется определенный набор свободных переменных из числа x_j, которые принимают произвольные значения, а значения любых других неизвестных (которые называются зависимыми) выражаются как линейные функции от значений свободных переменных.

Одним из самых простых методов решения такого рода систем уравнений является метод Гаусса. Суть его в том, чтобы применять к исходной системе эквивалентные (т. е. не меняющие множества решений) преобразования до тех пор, пока мы не придем к системе, решение которой очевидно.

В методе Гаусса используются преобразования трех типов. Преобразование первого типа состоит в том, чтобы поменять местами два уравнения в системе. Преобразование второго типа состоит в том, чтобы умножить некоторое уравнение на ненулевое число, т. е. все коэффициенты этого уравнения и его свободный член домножаются на одно и то же ненулевое число. Преобразования третьего типа состоят в том, чтобы вычесть одно уравнение из другого, т. е. выбираются два уравнения и все коэффициенты первого из них заменяются на разности этих коэффициентов и соответствующих (при той же неизвестной) коэффициентов второго.

При помощи таких преобразований система линейных уравнений приводится к так называемому ступенчатому виду. Он характеризуется следующими свойствами:

1) В каждом следующем уравнении количество нулевых коэффициентов в начале уравнения строго больше, чем в предыдущем. Исключения допускаются только тогда, когда и в предыдущем, и в следующем уравнении все коэффициенты и свободный член нулевые.

2) Для любого уравнения U, в котором имеется ненулевой коэффициент, во всех остальных уравнениях при переменной, при которой в уравнении U стоит первый ненулевой коэффициент, коэффициент равен нулю. Другими словами, если в каком-то столбце (т. е. среди коэффициентов при одной и той же переменной) встречается первый ненулевой коэффициент какого-либо уравнения, то все остальные коэффициенты в этом столбце должны быть нулевыми.

Очевидно, что если после приведения к ступенчатому виду у нас появилось уравнение вида $0 = b$, где b — ненулевое число, исходная система не имела решений. Если же таких уравнений нет, то система имеет решения, причем переменные, соответствующие столбцам из условия 2), будут зависимыми, а все остальные — свободными. Оставшиеся же ненулевые уравнения будут служить для выражения значений зависимых переменных через свободные.

Теперь возникает следующий вопрос: «Как при помощи описанных выше преобразований действительно привести систему уравнений к ступенчатому виду?»

Мы рассмотрим несколько разновидностей алгоритма Гаусса, последовательно переходя от простых к более сложным.

Самая простая разновидность алгоритма Гаусса называется алгоритмом Гаусса-Жордана. Она состоит в следующем.

Сначала мы смотрим на первый столбец системы. Предположим, что первый его элемент отличен от нуля. Тогда, каков бы ни был первый коэффициент второго уравнения, мы всегда можем при помощи указанных выше преобразований сделать его нулевым, просто домножив первое уравнение на ненулевое число так, чтобы его первый коэффициент совпал с первым коэффициентом второго уравнения, и затем вычтя первое уравнение из второго. Аналогично мы можем обнулить все остальные коэффициенты первого столбца. Таким образом, после этих преобразований единственным ненулевым коэффициентом в первом столбце будет первый.

Теперь обратимся ко второму столбцу, и предположим, что во втором уравнении второй коэффициент отличен от нуля. Тогда мы можем аналогично тем действиям, которые были предприняты для обнуления коэффициентов первого столбца, обнулить все вторые коэффициенты всех уравнений, кроме второго. Важно заметить здесь, что при обнулении вторых коэффициентов первые не испортятся, поскольку к началу обнуления вторых коэффициентов первый коэффициент второго уравнения уже обнулен, и потому вычитание второго уравнения с любым коэффициентом из любого другого уравнения не меняет его первый коэффициент. Продвигаясь теперь таким образом от одного столбца к следующему, мы наконец приведем нашу систему уравнений к ступенчатому виду.

Задача 1. Написать программу, решающую систему линейных уравнений методом Гаусса-Жордана.

Только что изложенный метод, однако, обладает тем существенным недостатком, что нам необходимо предполагать отличие от нуля некоторых коэффициентов. Конечно, поскольку вычисления в вещественных числах выполняются лишь приближенно, точное равенство нулю некоторых коэффициентов в реальных задачах не встретится никогда; однако, нужные нам коэффициенты могут быть очень маленькими числами, и необходимое нам деление на них может приводить к резкому падению точности вычислений.

Поэтому часто используются некоторые приемы повышения надежности вычислений. Одним из простейших является так называемый выбор главного элемента по столбцу. В этом случае, когда нам нужно с помощью очередного уравнения обнулять коэффициенты из некоторого столбца, мы сначала ищем среди всех коэффициентов в данном столбце у еще не использовавшихся уравнений наибольший по модулю, и его уравнение меняем местами с уравнением, номер которого совпадает с номером рассматриваемого столбца.

Задача 2. Доделать решение предыдущей задачи так, чтобы оно использовало выбор главного элемента по столбцу.

Вышеуказанный прием, однако, можно еще усилить, если разрешить обрабатывать столбцы не в том порядке, в котором они стоят в системе, т. е. в порядке индексов неизвестных, а в произвольном порядке. Такой подход называется выбор главного элемента по всей подматрице и состоит в следующем. Когда нам надо найти очередной коэффициент, с помощью которого мы будем обнулять все остальные коэффициенты его столбца, мы ищем этот коэффициент теперь не только среди всех коэффициентов очередного столбца, но среди всех столбцов, которые еще не участвовали в этом процессе, кроме столбца свободных членов.

Такой подход решает также и проблему свободных переменных. Если бы мы производили вычисления точно, мы могли бы встретится с ситуацией, когда в очередном столбце все коэффициенты уравнений, еще не задействованных в качестве «исключателей», т. е. тех уравнений, при помощи которых мы обнуляли коэффициенты, равны нулю. В этом случае нужно было бы пометить неизвестную, соответствующую этому столбцу, как свободную, и перейти к следующему столбцу. Но на практике мы вычисляем лишь приближенно, и если все интересующие нас элементы очередного столбца малы по модулю, мы не можем сказать, почему — то ли они должны быть равны нулю, и отличаются от нуля только по причине приближенности вычислений, то ли нет.

Задача 3. Доделать решение предыдущей задачи так, чтобы оно использовало выбор главного элемента по всей подматрице.

Задача 4. Если A — квадратная матрица, под обратной матрицей A^{-1} понимается такая матрица того же размера, что $AA^{-1} = E$, где E — так называемая единичная матрица, элементы главной диагонали которой равны 1, а все остальные — 0. Разработать алгоритм вычисления обратной матрицы и написать соответствующую программу, реализующую идеи: а) простейшего метода Гаусса-Жордана, б) с выбором главного элемента по столбцу, в) по всей подматрице.

59 Транспортная задача

Рассмотрим следующую задачу. Пусть имеется n складов, на каждом из которых имеется запас продукции одного и того же вида в объеме a_j единиц, где j — номер склада, $1 \le j \le n$. Мы будем предполагать, что продукция — весовая или настолько мелкоштучная, что ее количество с удовлетворительной точностью вполне может быть выражено вещественным числом. Пусть также имеется m потребителей этой продукции, потребность каждого из которых выражается числом b_i единиц, где i — номер потребителя, $1 \le i \le m$. Мы также предполагаем, что продукция может быть доставлена с любого склада любому потребителю, и стоимость доставки x единиц продукции со склада номер j потребителю номер i определяется по формуле $c_{ij}x$, где c_{ij} — некоторое неотрицательное число.

Под планом понимается указание доставлять x_{ij} единиц продукции со склада номер j потребителю номер i, для всех имеющихся значений i и j. Таким образом, план полностью описывает стратегию доставки продукции со складов потребителям и состоит из mn вещественных чисел. Под допустимым планом понимается план, соответствующий задаче, т. е. такой набор неотрицательных чисел x_{ij}, что $\sum_{i=1}^{m} x_{ij} = a_j$ для всех j и $\sum_{j=1}^{n} x_{ij} = b_i$ для всех i.

Общая стоимость плана доставки всей продукции всем потребителям определяется как $\sum_{i,j} c_{ij}x_{ij}$. Задача теперь состоит в том, чтобы по заданным объемам складов, потребностям и ценам доставки (т. е. числам a, b и c) найти наиболее дешевый план, или один из таких планов, если их несколько.

Алгоритм решения этой задачи состоит из двух частей. В первой части ищется какой-нибудь допустимый план, не обязательно оптимальный. Самый простой метод нахождения такого плана называется методом северо-западного угла и состоит в следующем. Сначала мы рассматриваем первый склад и первого потребителя, и предписываем доставить из первого склада первому потребителю максимально допустимое количество продукции. Очевидно, это количество будет определяться как минимум из запасов первого склада и потребностей первого потребителя. После такого назначения будет выполняться одно из двух утверждений: либо запас первого склада будет полностью исчерпан, тогда его можно временно исключить из рассмотрения, подкорректировав потребность первого потребителя в соответствии с тем фактом, что часть ее мы уже удовлетворили, либо будет полностью удовлетворена потребность первого потребителя (тогда временно исключаем из рассмотрения его, аналогично подправив запас первого склада). Продолжая назначать перевозки таким образом, мы обязательно получим некоторый допустимый, хотя и скорее всего неоптимальный, план.

Вышеописанный метод называется методом северо-западного угла потому, что обычно план изображают в виде таблицы чисел x_{ij}, в которой индекс i служит номером строки, а индекс j — номером столбца. Так вот, данный метод состоит в том, чтобы на каждом шаге рассматривать самый верхний левый (северо-западный) элемент такой таблицы среди тех, куда по условиям задачи еще можно назначить какие-либо перевозки, и в этот элемент назначать перевозку максимально возможного количества продукции.

Вторая часть алгоритма состоит в последовательном изменении имеющегося плана до тех пор, пока он не станет оптимальным. Для описания этого процесса нам понадобится ряд терминов.

Под ситуацией мы будем понимать две пары индексов $1 \le i_1 < i_2 \le m$ и $1 \le j_1 < j_2 \le n$. Под стоимостью ситуации мы понимаем число $x_{i_1j_1}c_{i_1j_1} + x_{i_2j_2}c_{i_2j_2} + x_{i_1j_2}c_{i_1j_2} + x_{i_2j_1}c_{i_2j_1}$. Под изменением стоимости ситуации мы понимаем одну из двух операций над планом: либо мы вычисляем $w = \min(x_{i_1j_1}, x_{i_2j_2})$ и затем w добавляется к $x_{i_1j_2}$ и $x_{i_2j_1}$, и вычитается из $x_{i_1j_1}$ и $x_{i_2j_2}$, либо, симметрично, $w = \min(x_{i_2j_1}, x_{i_1j_2})$ и затем w добавляется к $x_{i_1j_1}$ и $x_{i_2j_2}$, и

вычитается из $x_{i_2j_1}$ и $x_{i_1j_2}$. Очевидно, изменение стоимости ситуации сохраняет допустимость плана, однако стоимость нового плана может отличаться от стоимости исходного. Если стоимость плана может быть строго уменьшена путем изменения стоимости данной ситуации, то такая ситуация называется улучшаемой, а такое преобразование плана, приводящее к его удешевлению — улучшением плана.

Имеется теорема, утверждающая следующее: если улучшаемых ситуаций нет, план является оптимальным, т. е. наиболее дешевым из всех допустимых планов. Если же мы найдем улучшаемую ситуацию, ее необходимо улучшить.

Итак, получив первый план по методу северо-западного угла, мы вычисляем стоимости всех ситуаций и перспективы их улучшения. Если все они неулучшаемы, план оптимальный. Если это не так, находим любую улучшаемую ситуацию и улучшаем ее. Далее процесс повторяется до тех пор, пока мы не получим оптимальный план, т. е. план без улучшаемых ситуаций. Имеется еще одна теорема, утверждающая, что каков бы ни был план, полученный по методу северо-западного угла, и как бы мы ни выбирали на очередном шаге, какую ситуацию улучшать (разумеется, из тех, для которых это возможно), мы всегда придем к оптимальному плану за конечное число шагов. Правда, количество этих шагов может зависеть от того, какие ситуации мы улучшаем в первую очередь, и от того, как изначально были расположены числа a, b и c (от чего может сильно зависеть степень неоптимальности плана, полученного по методу северо-западного угла).

Задача. Запрограммировать вышеописанный алгоритм решения транспортной задачи.

60 Численное решение уравнений

Далеко не все уравнения могут быть решены аналитически. Например, из графика видно, что уравнение $e^x = x + 3$ имеет два решения, однако они не могут быть найдены аналитически, т. е. точно. Поэтому часто возникает необходимость решать уравнения приближенно, т. е. находить некоторые числа, расположенные достаточно близко от точных решений.

Существует несколько методов отыскания приближенных решений уравнений. Все они предполагают выполнение ряда условий, при которых данный метод можно применять, и при выполнении этих условий предлагают ряд действий, причем чем больше действий мы выполняем, тем точнее мы вычисляем приближенное решение нашего уравнения.

Различные методы отличаются друг от друга «придирчивостью», т. е. ограничительностью условий применимости данного метода, и скоростью сходимости, т. е. тем, насколько быстро растет точность с ростом числа выполненных действий.

Самым простым методом решения уравнений является метод половинного деления. Исходными данными для этого метода являются функция f и концы отрезка $[a, b]$ (мы решаем уравнение $f(x) = 0$ на указанном отрезке). Условием применимости этого метода будет следующее: на концах нашего отрезка функция f принимает значения разных знаков.

Сам метод состоит в следующем: мы берем середину исходного отрезка и рассматриваем случаи:

а) значение в середине равно 0 (с требуемой точностью) — тогда корень найден, на чем процесс и заканчивается;

б) иначе на концах одной из половинок нашего отрезка функция опять будет принимать значения разных знаков, и мы переходим к рассмотрению этой половинки, применяя теперь уже к ней наш метод; мы делаем это до тех пор, пока рассматриваемый отрезок не станет короче требуемой точности, когда в качестве ответа будет годится середина нашего отрезка.

Задача 1. Написать программу, находящую приближенное значение положительного корня уравнения $e^x = x + 3$, причем в программе должна быть функция вычисления f и функция решения уравнения с параметрами f, a и b.

Задача 2. Усовершенствовать предыдущую программу, чтобы она искала все корни уравнения $f(x) = 0$ следующим образом: сначала перебираются все отрезки вида $[n, n + 1]$ для целых значений n от -10 до 10, и на тех отрезках, на которых функция меняет знак, ищутся корни, что дает некоторый список корней; затем рассматриваются отрезки вдвое короче на в два раза большем интервале (от -20 до 20), и мы получаем новый список корней, куда входят и корни, найденные на предыдущем шаге, и так далее до тех пор, пока на очередном шаге в список не добавится ни одного нового корня. При помощи указанного усовершенствования решить уравнение $\sin x = x/20$.

Следующий по увеличению скорости сходимости метод называется методом хорд. Он похож на метод половинного деления, и условия применимости у него такие же. Отличие состоит в том, что мы рассматриваем на очередном шаге не середину имеющегося отрезка, а точку, полученную как пересечение исходного отрезка от $(a, 0)$ до $(b, 0)$ и отрезка между концами графика f на этом отрезке от $(a, f(a))$ и $(b, f(b))$ (эти отрезки лежат на плоскости, и потому у каждого конца этих отрезков две координаты).

Задача 3. Решить задачи 1 и 2, но в качестве метода в них использовать метод хорд вместо половинного деления. Сравнить скорость сходимости двух этих методов.

Следующие два метода построены на несколько другом принципе. Первый из них называется методом итераций. Для того, чтобы им воспользоваться, исходное уравнение $f(x) = 0$ заменяется на $g(x) = x$, где $g(x) = cf(x) + x$ (c — постоянная, значение которой подбирается, исходя из требований применимости данного метода). Требования применимости для метода итераций следующие: $|g'(x)| < 1$ на $[a, b]$. Сам метод итераций состоит в выборе начального

приближения x_0 и затем построении последовательности $x_{n+1} = g(x_n)$. Останавливается этот процесс тогда, когда разность между соседними приближениями x_i станет меньше требуемой точности. Последнее посчитанное приближение и считается за ответ.

Задача 4. Подобрать константу c для уравнения $e^x = x + 3$ на отрезке $[0, 2]$ и решить его методом итераций. Сравнить скорость сходимости для начального приближения 0 со скоростью сходимости предыдущих методов.

Последний метод, который мы здесь рассмотрим, называется методом касательных или методом Ньютона. Он похож на метод итераций, но формула для g будет несколько другая: $g(x) = x - f(x)/f'(x)$, где $f'(x)$ — производная функции f. Этот метод сходится гораздо быстрее метода итераций, но он капризнее: он требует, чтобы было $f''(x)f'(x) > 0$ от корня x^* до начального приближения x_0 (в случае, когда $x_0 > x^*$).

Задача 5. Написать программу, находящую приближенное значение положительного корня уравнения $e^x = x+3$ на отрезке $[0, 2]$ по методу Ньютона (начальное приближение $x_0 = 2$) и сравнить скорость его сходимости с предыдущими методами.

Метод Ньютона может применяться и для вычисления хорошо известных математических функций, например, корней. Так, один из быстрейших методов вычисления $\sqrt[3]{a}$ основан на решении уравнения $x^3 = a$ методом Ньютона.

Оказывается, два последних обсуждаемых метода применимы не только к одиночным уравнениям, но и к системам уравнений. Пусть дана система уравнений

$$\begin{cases} f_1(x, y) = 0, \\ f_2(x, y) = 0. \end{cases}$$

Тогда для решения системы методом итераций нам нужно ввести две дополнительные функции $g_1(x, y) = x + c_{11}f_1(x, y) + c_{12}f_2(x, y)$, $g_2(x, y) = y + c_{21}f_1(x, y) + c_{22}f_2(x, y)$, где c_{ij}, как и в методе итераций для уравнений, — константы, подбираемые из соображений применимости метода. Условие же применимости метода будет такое:

$$\sqrt{\left(\frac{\partial g_1}{\partial x}\right)^2 + \left(\frac{\partial g_1}{\partial y}\right)^2 + \left(\frac{\partial g_2}{\partial x}\right)^2 + \left(\frac{\partial g_2}{\partial y}\right)^2} < 1$$

в области, где мы ищем корень; здесь $\frac{\partial g_1}{\partial x}$ обозначает частную производную функции g_1 по переменной x (нужно вычислить производную функции g_1, считая x переменной, а y — постоянным параметром).

В качестве начального приближения здесь уже должны выступать два числа x_0, y_0. Формулы для вычисления очередного приближения будут такими: $x_{n+1} = g_1(x_n, y_n)$, $y_{n+1} = g_2(x_n, y_n)$.

Задача 6. Подобрать константы c_{ij} и решить систему

$$\begin{cases} \ln(x + y^2) - \sin(xy) = 1, \\ x + 2e^{x+y} - 3xy = 1. \end{cases}$$

методом итераций ($x \in [-1, 1]$, $y \in [-2, 0]$).

Метод Ньютона в данной ситуации получается подстановкой

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = -\frac{1}{\frac{\partial g_1}{\partial x}\frac{\partial g_2}{\partial y} - \frac{\partial g_1}{\partial y}\frac{\partial g_2}{\partial x}} \begin{pmatrix} \frac{\partial g_2}{\partial y} & -\frac{\partial g_1}{\partial y} \\ -\frac{\partial g_2}{\partial x} & \frac{\partial g_1}{\partial x} \end{pmatrix}.$$

Здесь опущены аргументы функций двух переменных; у всех таких функций в этих выражениях должны быть аргументы x_n, y_n. В методе Ньютона c_{ij} уже не будут константами, но будут зависеть от текущего приближения. Те значения для c_{ij}, которые здесь использованы, являются аналогами $1/f'$ в методе Ньютона для уравнений (здесь используется матрица, обратная матрице Якоби, для тех, кто знает, что это такое).

Задача 7. Подобрать начальное приближение и решить систему задачи 6 методом Ньютона, сравнив скорость сходимости с методом итераций.

61 Длинная арифметика

Для нужд обычных программ 32-х разрядных целых чисел, как правило, вполне достаточно. Если их разрядности не хватает, можно использовать 64-х разрядные целые числа, которых хватит наверняка. Однако, для нужд вычислительных экспериментов в некоторых областях математики, прежде всего, естественно, теории чисел, могут потребоваться гораздо более длинные числа. Вообще, для такого применения очень удобно иметь такой тип целых чисел, функции для обработки значений которого сами заботятся о том, сколько памяти выделяется под целое число, и изменяют это количество в процессе выполнения программы автоматически в соответствии с потребностями. Размер таких целых чисел ограничивается исключительно количеством имеющейся в компьютере оперативной памяти, что позволяет прозрачно оперировать очень большими числами.

К реализации такого типа есть два подхода. Такие числа можно хранить как в массиве обычных целых чисел, так и в связном списке. Т. е. знак такого числа хранится отдельно, а абсолютная величина (в двоичной системе

счисления) разрезается на порции по 31 биту (если целое число содержит 16 битов, то длина порции будет 15) и последовательность этих порций хранится, например, в (динамическом) массиве.

Задачи.

1. Написать структурный тип данных для хранения таких чисел.

2. Написать функцию для копирования таких чисел из одной переменной такого типа в другую.

3. Написать функцию для сложения таких чисел (предполагая, что оба они неотрицательны).

4. Написать функцию для сравнения таких чисел (предполагая, что оба они неотрицательны). Она должна иметь два параметра такого типа и возвращать 1, если первое число больше, 0, если они равны, и -1, если второе число больше.

5. Написать функцию для вычитания таких чисел (предполагая, что оба они неотрицательны; результат может иметь любой знак).

6. Написать функцию для сложения таких чисел, каждое из которых может иметь любой знак.

7. Написать функции для битовых операций (и, или, не, сдвиг) для таких чисел; все эти операции игнорируют знак числа.

Что касается умножения, то его проще всего реализовать в столбик в двоичной системе счисления.

8. Написать функцию, выдающую степень двойки при старшем разряде такого числа.

9. Написать функцию, выдающую значение двоичной цифры числа по той степени двойки, которая стоит при данной цифре; например, 0 соответствует младшему биту, 1 — следующему (стоящему при 2^1), и т. д.

10. Написать функцию для умножения таких чисел.

11. Написать функцию для превращения строки в такое число.

Деление реализовать несколько труднее, но самый простой способ — углом опять же в двоичной системе.

12. Написать две функции для деления таких чисел с остатком (первая функция выдает неполное частное, вторая — остаток).

13. Написать функцию для превращения такого числа в строку.

14. С помощью написанных ранее функций посчитать 1000!.

62 Модульная арифметика

В некоторых разделах математики широко используются вычисления в остатках по фиксированному модулю. Следующий ряд задач посвящен этой теме.

Задачи.

1. Написать структурный тип данных, хранящий величину модуля и остаток по этому модулю (модуль всегда должен быть положительным, и остаток должен лежать в пределах от 0 до модуля-1). Написать функцию residue(число, модуль), возвращающую объект данного структурного типа, представляющий остаток от указанного числа по указанному модулю (внимание! если число отрицательно, стандартная операция mod возвращает отрицательный результат).

2. Написать функции add, sub и mul, возвращающие, соответственно, сумму, разность и произведение объектов указанного в предыдущей задаче структурного типа. Эти функции должны проверять, что мы вычисляем соответствующее выражение с остатками по одному и тому же модулю; в противном случае, нужно выводить на экран сообщение об ошибке и возвращать «недопустимый объект» (объект, у которого в поле модуля записан 0).

3. Написать функцию, по двум целым параметрам выдающую НОД и коэффициенты его линейного разложения (через параметры, передаваемые через указатели). Т. е. если этой функции переданы числа a и b, она должна вернуть $c =$НОД(a, b) и такие числа k и m, что $c = ka + mb$ (для вычисления таких k и m нужно слегка модифицировать алгоритм Евклида). Пользуясь этой функцией, реализовать операцию деления для остатков dvd (если эта операция для конкретных переданных ей значений операндов не определена, нужно выводить на экран сообщение об ошибке и возвращать «недопустимый объект»; если результатом этой операции может быть несколько разных остатков, возвращать любой из них).

4. Написать функцию pow(остаток, степень) для возведения остатков в целую степень такую, которая использует не более $2\log_2 n$ умножений при возведении в степень $n > 0$ (для отрицательных n вычисляем (1/остаток)$^{-n}$, см. задачу 3). Используя эту функцию, определить период последовательности $7^n \pmod{100}$, $n = 0, 1, 2, \ldots$.

5. Написать функцию, принимающую остаток (объект указанного в предыдущих задачах типа) и динамический массив остатков-коэффициентов многочлена, и возвращающую значение данного многочлена на указанном остатке. Используя эту функцию, написать другую, принимающую динамический массив коэффициентов уравнения и возвращающую динамический массив решений этого уравнения, определяемых перебором (модуль определяется из объектов-коэффициентов уравнения, у них у всех он должен быть один и тот же). Используя вторую функцию, определить все решения сравнений [Фаддеев, Соминский 41 с,е] а) $x^3 - 3x + 1 \equiv 0 \pmod{19}$; б) $x^3 \equiv 10 \pmod{37}$.

6. Используя первую функцию из задачи 3, написать функцию, реализующую утверждение «Китайской теоремы об остатках»: если $m_1, \ldots, m_n$ попарно взаимно просты, то для любых остатков a_i по модулю m_i найдется единственный

остаток x по модулю $m_1 m_2 \dots m_n$ такой, что $x \equiv a_i \pmod{m_i}$. Функция должна принимать динамический массив остатков $a_i \pmod{m_i}$ и возвращать указанный остаток $x \pmod{m_1 m_2 \dots m_n}$.

7*. Написать функцию, возвращающую первообразный корень по простому модулю (параметр функции).

8*. Написать функцию, возвращающую индекс остатка по простому модулю по первообразному корню (параметры функции).

63 Вычисления в комплексных числах

Широко используемое в анализе поле вещественных чисел обладает одним существенным недостатком: не каждый многочлен с вещественными коэффициентами имеет вещественные корни. Для исправления этого недостатка было изобретено поле комплексных чисел, состоящее из выражений вида $a + bi$, где a и b — произвольные вещественные числа, а i — специальный символ (коэффициент при i называется мнимой частью комплексного числа, а свободный член — его вещественной частью). Для сложения и вычитания комплексных чисел используются традиционные правила (коммутативность сложения, вынесение i за скобки). Для умножения также используется дистрибутивность сложения относительно умножения ($x(y + z) = xy + xz$) и правило $i^2 = -1$. Если $z = x + yi$, то под сопряженным к z числом $\bar{z}$ понимается $x - yi$. Чтобы определить деление комплексных чисел, нужно домножить делимое и делитель на число, сопряженное к делителю, после чего нужно будет делить на вещественное число, для чего можно просто воспользоваться дистрибутивностью.

Задача 1. Написать структурный тип данных, представляющий комплексное число. Написать также функции add, sub, mul, dvd, осуществляющие арифметические действия с комплексными числами, а также функцию conj, вычисляющую для своего комплексного аргумента сопряженное число.

Задача 2. Написать функцию sqrt_c, вычисляющую для комплексного числа квадратный корень (формулы для вычисления квадратного корня можно найти, просто выписав систему уравнений на вещественную и мнимую часть результата, и решив ее). Написать еще одну функцию, решающую квадратное уравнение в комплексных числах (формула будет такая же, как и в вещественных числах, но нет ограничений на дискриминант и все действия выполняются в комплексных числах). Пользуясь этой функцией, решить квадратное уравнение $(1+2i)x^2 - (4+13i)x - 47 + 21i = 0$.

Задача 3. Написать функцию cbrt_c, вычисляющую кубический корень из комплексного числа a — параметра данной функции, при помощи решения уравнения $x^3 = a$ методом Ньютона (см. параграф по ссылке «Приближенное решение уравнений и систем»). Написать еще одну функцию, решающую кубическое уравнение с комплексными коэффициентами по формуле Кардано (см. Википедия, запрос в Google «формулы Кардано википедия»). При этом нужно только заметить, что если $x = \sqrt[3]{a}$ — одно из значений кубического корня из комплексного числа a, найденное функцией cbrt_c, то два других получаются по формуле $(-\frac{1}{2} \pm \frac{\sqrt{3}}{2}i)x$. Пользуясь этой функцией, решить кубическое уравнение $(3 + 5i)x^3 - (12 + 20i)x^2 - (65 - 39i)x + 74 - 228i = 0$.

Задача 4. Написать функции: exp_c, вычисляющую e^x для комплексного x по формуле $e^{a+bi} = e^a(\cos b + i \sin b)$; ln_c, вычисляющую натуральный логарифм комплексного числа (обратная функция к e^x, определяется с точностью до добавки в $2\pi in$ при целом n); sin_c, вычисляющую синус комплексного числа x по формуле $\sin x = \frac{e^{ix} - e^{-ix}}{2i}$; cos_c — косинус по формуле $\cos x = \frac{e^{ix} + e^{-ix}}{2}$; tg_c — тангенс; arcsin_c, arccos_c, arctg_c — обратные тригонометрические функции (находятся из решения уравнений на e^{ix} и затем нужно использовать логарифм). Пользуясь этими функциями, найти arcsin(5) (оказывается, это просто комплексное число).

Поскольку комплексное число определяется парой вещественных, мы можем геометрически представлять себе множество комплексных чисел как плоскость (так же, как множество вещественных чисел представляется как прямая). На этой плоскости можно ввести не только традиционные декартовы координаты (вещественная и мнимая часть), но и полярные (для данного комплексного числа, рассматриваемого как точка на плоскости, полярные координаты называются модулем (расстояние) и аргументом (угол)). При этом модуль однозначно определяется для любого комплексного числа, а аргумент определяется только для ненулевых чисел и с точностью до добавки, кратной 2π.

Если теперь у нас имеется некоторая элементарная функция, а также кривая на плоскости комплексных чисел, причем функция не обращается в ноль на указанной кривой, мы можем рассмотреть приращение аргумента значения функции вдоль этой кривой. Т. е. мы начинаем движение по кривой и следим за тем, как меняется аргумент значения нашей функции (значение функции, рассматриваемое как вектор, поворачивается в ту или иную сторону). Если кривая замкнута, то при перемещении вдоль нее и возврате в ту же точку, из которой мы начинали, значение функции делает несколько полных оборотов вокруг точки ноль. Приращение аргумента в этом случае равно числу этих оборотов, умноженному на 2π. Причем для замкнутых кривых не имеет никакого значения, с какой точки начинать обход, лишь бы мы прошли по всей кривой и вернулись обратно в точку, где начали обход.

Оказывается, для элементарных функций в комплексных числах имеется аналог метода половинного деления для приближенного решения уравнений (на самом деле, этот метод работает не только для элементарных функций, но и для функций из несколько более широкого класса, а именно аналитических функций). Этот метод основан на возможности вычислить число корней заданной функции в заданной области, обойдя эту область по ее границе. Если на этой плоскости у нас имеется некоторая ограниченная замкнутой кривой область, то число корней любой

элементарной функции, определенной во всей этой области вместе с границей, равно приращению аргумента значения этой функции вдоль границы области, деленному на 2π, т. е. числу оборотов значения функции вокруг точки ноль при обходе по границе области.

Теперь предположим, что нам нужно найти корень некоторой элементарной функции в заданном прямоугольнике на комплексной плоскости. Обойдя этот прямоугольник и вычислив приращение аргумента значений функции, мы можем сказать, есть ли в этом прямоугольнике корни нашей функции и сколько их. Далее мы можем поделить этот прямоугольник пополам (горизонтально или вертикально, разумно делить пополам ту сторону, которая длиннее) и аналогичным методом выяснить, имеются ли корни в каждой из половинок. Деля таким образом прямоугольник далее и запоминая только те его части, в которых есть корни, мы в конце концов приходим к набору достаточно маленьких прямоугольничков, в каждом из которых лежит по одному корню. Если размер каждого прямоугольника меньше требуемой точности, на этом процесс и заканчивается.

Можно сочетать этот метод с методом Ньютона: как только размер прямоугольника становится достаточно малым и в нем содержится один корень, этот корень определяется при помощи метода Ньютона с начальным значением — любой точкой данного прямоугольника.

Осталось обсудить только, как найти приращение аргумента для значения конкретной функции на некоторой кривой. Поскольку граница прямоугольника состоит из отрезков, мы обсудим этот вопрос, ограничиваясь отрезками в качестве кривых (общее приращение аргумента на ломаной линии, очевидно, складывается из приращений аргумента на каждом ее звене).

Итак, у нас есть функция и отрезок, и нам нужно найти приращение аргумента функции на этом отрезке. Для этого мы ставим на отрезок несколько точек, делящих его на равные части небольшой длины (разумно взять их не длиннее 0,01) и далее вычисляем приращение аргумента на каждом маленьком отрезке, предполагая, что это изменение невелико, просто вычисляя разность между аргументом значения функции в конце маленького отрезочка и в его начале. После этого, мы складываем все так вычисленные разности. На первый взгляд, это то же самое, что просто взять и вычесть из аргумента значения функции в конце всего отрезка значение аргумента в его начале, но на самом деле это не так (результаты таких двух методов вычисления приращения аргумента могут отличаться на величину, кратную 2π).

Теперь мы удваиваем число отрезков разбиения, и производим вычисления снова. Если результат совпадает с полученным ранее, он считается верным, иначе мы опять удваиваем число отрезков разбиения, и пересчитываем приращение аргумента снова, делая так до тех пор, пока не получим на некотором разбиении и на вдвое более частом один и тот же результат.

Задача 5. Написать программу, ищущую все решения уравнения $x^7+(5-18i)*x^6-(142+82i)x^5-(602-656i)x^4+(1375+2472i)x^3+(6347-558i)x^2-(8434+17110i)x-33110+42320i=0$ в прямоугольнике $-10.375 \le x \le 10.582$, $-10.273 \le y \le 10.964$ методом, комбинирующим метод половинного деления и метод Ньютона.

64 Арифметика многочленов

1. Написать структурный тип данных «одночлен», представляющий одночлены из не более чем 26 переменных (переменные — маленькие латинские буквы). Он должен хранить степени каждой переменной, с которой она входит в одночлен (массив из 26 целых чисел), а также еще одно целое число — коэффициент при одночлене. Также нужно реализовать функции, вычисляющие произведение и частное одночленов (поскольку деление возможно не всегда, нужна еще логическая функция, проверяющая, что деление возможно), а также процедуры чтения одночлена с клавиатуры и печати его на экран.

2. Написать структурный тип «многочлен», хранящий список одночленов (см. предыдущую задачу). Также нужно реализовать функции, вычисляющие сумму, разность и произведение многочленов (последняя функция должна раскрывать скобки, чтобы привести произведение многочленов к виду суммы одночленов), а также процедуры чтения многочлена с клавиатуры и печати его на экран.

Для дальнейшего нам понадобится ввести отношение порядка на множестве одночленов, или, правильнее сказать, на наборах степеней переменных, поскольку коэффициенты при одночленах не участвуют в этом отношении порядка никак. Набор степеней переменных можно рассматривать как вектор из 26 целых чисел. На таких наборах вводится лексикографический порядок: мы ищем первую координату, которая у этих векторов различается, и порядок между значениями этой координаты определяет порядок между векторами. Например, в смысле только что определенного отношения порядка, $x > y$, поскольку у всех переменных до x степени в обоих одночленах совпадают (и равны нулю), а степень x в первом одночлене (1) больше, чем во втором (0).

3. Написать функцию, сравнивающую одночлены в соответствии с только что определенным отношением порядка. Она должна иметь два параметра (сравниваемые одночлены), и целый результат (-1, если первый одночлен меньше второго, 0 — равен, т. е. одночлены могут отличаться только коэффициентом, и 1 — больше).

4. Написать процедуру, приводящую в многочлене подобные члены. Она должна сначала упорядочивать одночлены, входящие в состав многочлена, используя функцию предыдущей задачи, по убыванию, а затем, когда одночлены

с одинаковыми степенями переменных окажутся рядом в списке, заменять их на их сумму, складывая коэффициенты. Помимо приведения подобных членов, эта процедура будет упорядочивать одночлены по убыванию, что очень пригодится нам в последующих задачах.

5. Написать процедуру деления одного многочлена на другой. Она должна делать следующее: берем наибольший одночлен первого многочлена и делим его на наибольший одночлен второго. Если такое деление невозможно, результат отрицательный — первый многочлен не делится на второй. Если же оно возможно, то добавляем P — результат деления наибольших одночленов к тому многочлену, в котором мы накапливаем результат деления, и вычитаем из делимого произведение делителя и P, после чего повторяем указанные действия до тех пор, пока не будет выполнено одно из условий: либо на очередном шаге деление наибольших одночленов невозможно, и тогда делимое не делится на делитель, либо делимое обнулится, и тогда у нас будет сформировано частное.

Многочлен называется симметрическим, если он не меняется от любой перестановки его переменных. Например, для двух переменных многочлены $a + b$ и ab являются симметрическими, а $a + 3b$ и a^2b — нет. Для любого числа переменных n можно определить стандартный набор из n симметрических многочленов, которые называются элементарными, следующим образом: для $i = 1, \ldots, n$ i-й элементарный симметрический многочлен e_i представляет собой сумму всех произведений переменных в наборах по i штук. Например, для трех переменных элементарными симметрическими многочленами будут $e_1 = a + b + c$, $e_2 = ab + ac + bc$ и $e_3 = abc$ (все эти многочлены выписаны в порядке убывания их одночленов).

Имеется следующая замечательная теорема: всякий симметрический многочлен может быть представлен как многочлен от элементарных симметрических многочленов. Например, $a^2 + b^2 + c^2 = (a + b + c)^2 - 2(ab + ac + bc)$. Одно из самых простых доказательств этой теоремы несколько напоминает алгоритм деления многочленов из последней задачи и состоит в следующем: берем самый большой одночлен в нашем многочлене, пусть это будет $Cx_1^{n_1} \ldots x_k^{n_k}$. Поскольку у нас многочлен симметрический, а одночлен наибольший, мы обязательно будем иметь $n_1 \geq n_2 \geq \ldots \geq n_k$. Пусть теперь $n_1 = \ldots = n_{i_1} > n_{i_1+1} = \ldots = n_{i_2} > \ldots > n_{i_{s-1}+1} = \ldots = n_{i_s}$, где $i_s = k$. Тогда если вычесть из исходного многочлена произведение $Ce_{i_1}^{n_{i_1}-n_{i_2}} e_{i_2}^{n_{i_2}-n_{i_3}} \ldots e_{i_s}^{n_{i_s}}$, то в результате мы получим также симметрический многочлен, но в нем уже самый большой одночлен будет меньше, чем был в исходном. Таким образом, рано или поздно мы придем к тому, что исходный многочлен обнулится.

6. Написать функцию, принимающую в качестве параметров симметрический многочлен и число n переменных в нем, и возвращающий другой многочлен такой, что если в него подставить e_1 вместо первой переменной, e_2 вместо второй, …, e_n вместо n-й, то получится тот многочлен, который был передан в качестве параметра. Например, если были переданы многочлен $a^2 + b^2 + c^2$ и число 3, то вернуть надо $a^2 - 2b$ (см. начало предыдущего абзаца).

65 Численное решение дифференциальных уравнений

В задачах механики часто возникают дифференциальные уравнения, которые нельзя решить аналитически, т. е. получить явную формулу для решения, пусть даже с использованием символа интеграла. Такие дифференциальные уравнения можно решить только численно, т. е. приближенно при помощи определенных вычислений.

Здесь будет описан один из самых популярных методов для численного решения дифференциальных уравнений и систем первого порядка. Вообще говоря, в механике обычно встречаются уравнения и системы уравнений второго порядка (ускорение выражается через положение и скорость), поэтому перед применением этого метода их нужно преобразовать к системам уравнений первого порядка, введя дополнительные переменные, обозначающие первые производные координат.

Данный метод (метод Рунге-Кутты 4 порядка), как и большое количество других численных методов для решений задачи Коши для уравнений и систем первого порядка, имеет дело с задачами Коши (задано начальное значение искомой функции) вида:

$$\frac{dy}{dx} = F(x, y), y(x_0) = y_0.$$

Здесь мы не будем делать никакого различия между уравнениями и системами уравнений, просто в случае уравнения $y(x)$ будет обозначать скалярную функцию, а в случае систем — вектор-функцию из соответствующего числа компонент.

Метод состоит в последовательном вычислении приближенных значений y_i для искомых величин $y(x_i)$, где x_i — последовательность точек, в которых нам нужно получить приближенные значения искомой функции. В простейшем случае (более сложные варианты мы рассматривать не будем) x_i — арифметическая прогрессия с разностью h (это число называется шагом сетки, а сами x_i — ее узлами).

Расчетные формулы для данного метода выглядят следующим образом:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

где

$$k_1 = hF(x_n, y_n), \tag{1}$$

$$k_2 = hF(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1), \tag{2}$$

$$k_3 = hF(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_2), \tag{3}$$

$$k_4 = hF(x_n + h, y_n + k_3). \tag{4}$$

Оценка погрешности данного метода — довольно трудное дело, однако имеется следующая формула для главного члена асимптотики погрешности на очередном шаге, в предположении, что предыдущие шаги вычислены точно:

$$y(x_{n+1}) - y_{n+1} \approx \frac{1}{31}(y_{n+1} - y^*_{n+1}),$$

где y^*_{n+1} — приближенное значение искомой функции, полученное по y_{n-1} и x_{n-1} с шагом $2h$.

66 Коды Хаффмана

Предположим, нам нужно передать по сети определенное сообщение, в котором встречаются символы из определенного конечного набора $c_1, \ldots, c_n$. При этом физически линия связи позволяет передавать только отдельные биты (0 или 1). Встает вопрос о том, как закодировать (т. е. представить в виде последовательностей из 0 и 1) символы из нашего набора так, чтобы общая длина сообщения в битах была минимальной. Результат кодирования всего сообщения представляется как результат приписывания друг к другу кодов отдельных символов, в порядке следования их в исходном сообщении. Чтобы впоследствии при чтении кода сообщения слева направо его можно было раскодировать, коды символов должны удовлетворять такому условию: никакой код одного символа не может совпадать с начальным отрезком кода другого символа.

Эту задачу как раз и решает код Хаффмана. Чтобы построить такой код, мы будем рассматривать следующую структуру данных. Нам понадобится набор пар, первым элементом которых является положительное целое число, а вторым — двоичное дерево, ребра которого помечены символами 0 или 1, а каждый лист — одним символом из нашего набора.

Изначально набор пар содержит столько пар, сколько у нас символов. Число в такой паре — количество вхождений данного символа в сообщение, а дерево состоит из единственного узла (корня, он же лист), помеченного соответствующим символом.

Теперь мы начинаем объединять пары, заменяя две из них на одну последовательно по следующим правилам. Берем две пары с наименьшими числами, и заменяем их на одну пару так: у результирующей пары число будет равно сумме чисел у исходных пар, а дерево будет состоять из корня, к которому прикрепляются в качестве сыновей корни двух деревьев из исходных пар (в какое из деревьев будет вести ребро из корня, помеченное 0, а в какое — 1, не важно).

В конце концов у нас останется одна пара, число у которой будет равно длине сообщения, а в дереве в качестве листьев будут встречаться по одному разу все символы из нашего набора. Чтобы теперь получить код некоторого символа, нужно рассмотреть путь в этом дереве от корня до соответствующего листа, и последовательно считать метки ребер на этом пути.

Задачи.

1. Написать функцию, строящую дерево кода Хаффмана по набору чисел вхождений отдельных символов в сообщение.

2. Написать функцию, кодирующую сообщение (параметры — сообщение, т. е. массив номеров символов, и дерево кодов, результат — массив логических значений — код сообщения).

3. Написать функцию, декодирующую сообщение (параметры — сообщение, т. е. массив логических значений, и дерево кодов, результат — массив номеров символов).