

Московский государственный университет имени М.В.Ломоносова
Казахстанский филиал

А. Ж. Баев

Введение в программирование на языке C и знакомство со средой UNIX

Учебное пособие

Нур-Султан
2019

УДК 004.4(075)
ББК 32.973.202я73
Б 15

*Рекомендовано к печати Ученым советом
Казахстанского филиала МГУ имени М.В.Ломоносова*

Автор:

Баев Ален Жуматаевич, преподаватель кафедры математики и информатики Казахстанского филиала МГУ имени М.В.Ломоносова.

Рецензенты:

Сальников А.Н., к.ф.-м.н., доцент кафедры Автоматизации систем вычислительных комплексов факультета ВМК МГУ имени М.В. Ломоносова, ведущий научный сотрудник лаборатории Вычислительной электродинамики.

Абдикалыков А.К., к.ф.-м.н., старший преподаватель кафедры математики и информатики Казахстанского филиала МГУ имени М.В. Ломоносова.

Редактор:

Воронова Е.С., преподаватель кафедры филологии Казахстанского филиала МГУ имени М.В. Ломоносова.

Баев А.Ж.

Б15 Введение в программирование на языке C и знакомство со средой UNIX / А. Ж. Баев. — Нур-Султан, 2019. — 404 с.

ISBN 978-601-332-357-2

Данное пособие предназначено для студентов первого курса направления «Математика» Казахстанского филиала МГУ имени М.В. Ломоносова. Основная цель — формирование первичных навыков программирования на языке C. Материал представлен в форме 14 тем, каждая из которых содержит теоретический материал, подробный разбор ключевых задач и задания для самостоятельной работы. Также в пособии описаны основные аспекты поведения в командной строке UNIX-систем на примере bash (Born Again Shell) и работы с компилятором C на примере gcc в среде UNIX.

УДК 004.4(075)

ББК 32.973.202я73

©Баев А.Ж., 2019

©Казахстанский филиал МГУ, 2019

ДОРОГИЕ СТУДЕНТЫ!

В Казахстанском филиале МГУ создана уникальная система образования, которая гарантирует высокое качество обучения студентов. Это достигается полным соблюдением стандартов учебных программ Московского университета в Филиале. Занятия в Филиале проводят ведущие профессора и преподаватели из Москвы и из Филиала. Ежегодно более 130 профессоров и преподавателей МГУ командированы Московским университетом в Филиал для чтения лекций и проведения семинаров.

Студенты направления «Математика» получают широкий спектр знаний. Одну из лучших школ математической подготовки в мире дополняют курсы по другим направлениям, в том числе и по программированию. Выпускники находят применение своим навыкам алгоритмического мышления не только в сфере информационных технологий, но и других сферах деятельности. Часть выпускников работает в ведущих компаниях Казахстана и России, специализирующихся на информационных технологиях, где активно применяют полученные знания. Курс по программированию согласуется с классическим подходом к обучению — закладывается фундамент, после изучения которого студенты успешно осваивают стремительно обновляющиеся современные информационные технологии и языки программирования.

Автор пособия — выпускник и преподаватель Казахстанского филиала МГУ — проводит большую работу по подготовке студентов и организации студенческих олимпиад по математике и программированию.

Желаю успешного освоения курса и профессионального роста!

**Директор
Казахстанского филиала МГУ
профессор**

А. В. СИДОРОВИЧ

Содержание

1	Техническое введение. Терминал и консольные редакторы	13
2	Числовые типы. Арифметические операторы	28
3	Переменные. Приведение типов. Первая программа	38
4	Условный оператор. Математические функции	67
5	Цикл while	97
6	Символьный тип	122
7	Цикл for	141
8	Массивы	166
9	Указатели и строки	196
10	Функции	229
11	Функция в качестве аргумента. Рекурсия	261
12	Структуры	283
13	Указатели. Динамические массивы.	309
14	Динамические матрицы. Аргументы командной строки	334
15	Файлы	356
16	Ответы к проверочным работам	375
17	Задания для подготовки к промежуточному контролю	389
18	Основная литература	402
19	Дополнительная литература	402

Содержание (подробное)

1	Техническое введение. Терминал и консольные редакторы	13
1.1	Операционная система	13
1.2	Терминал	13
1.3	Консольные текстовые редакторы	22
1.4	Напо: работа с файлами	22
1.5	Напо: редактирование файлов	23
1.6	Напо: навигация по тексту	24
1.7	Напо: файл настройки	25
1.8	Задания для самостоятельной работы	27
2	Числовые типы. Арифметические операторы	28
2.1	Целочисленные типы	28
2.2	Целочисленные константы	29
2.3	Целочисленные арифметические операции	30
2.4	Переполнение	32
2.5	Вещественные типы	33
2.6	Вещественные константы	33
2.7	Вещественные арифметические операции	34
2.8	Приоритет арифметических операций	35
2.9	Задания для самостоятельной работы	37
3	Переменные. Приведение типов. Первая программа	38
3.1	Переменные	38
3.2	Оператор присваивания	39
3.3	Инициализация при объявлении и мусорные значения	39
3.4	Неизменяемые переменные	40
3.5	Арифметические действия	41
3.6	Укороченные арифметические операторы	42
3.7	Неявное преобразование типов	44
3.8	Явное преобразование типов	46
3.9	Приоритет выполнения операторов	49
3.10	Комментарии	50
3.11	Минимальная программа	50
3.12	Вывод текста	51
3.13	GCC: компиляция	51
3.14	Форматный вывод	52
3.15	Форматный ввод	55
3.16	GCC: работа с ошибками	57
3.17	Отступы	58
3.18	Примеры	59

3.19	Задания для самостоятельной работы	63
3.20	Практикум на ЭВМ	65
4	Условный оператор. Математические функции	67
4.1	Операторы сравнения	67
4.2	Логические операторы	69
4.3	Приоритеты операторов	72
4.4	Тождества де Моргана	74
4.5	Условный оператор	74
4.6	Составной оператор	77
4.7	Пустой оператор	78
4.8	Вложенный условный оператор	80
4.9	Функция модуля с целочисленным аргументом	80
4.10	Функции с вещественным аргументом	82
4.11	Неэффективное использование «math.h»	84
4.12	Линковка библиотеки «math.h»	85
4.13	Примеры	85
4.14	Задания для самостоятельной работы	93
4.15	Практикум на ЭВМ	95
5	Цикл while	97
5.1	Цикл с предусловием while	97
5.2	Цикл с постусловием do while	99
5.3	Переменная-счётчик	100
5.4	Зацикливание	101
5.5	Отладочная печать	102
5.6	Отладка в gdb	103
5.7	Примеры	105
5.8	Задания для самостоятельной работы	118
5.9	Практикум на ЭВМ	120
6	Символьный тип	122
6.1	Таблица кодов ASCII	122
6.2	Арифметика и сравнение	123
6.3	Смена регистра	125
6.4	Битовые операции	126
6.5	Ввод-вывод символьного типа	126
6.6	Кириллица	128
6.7	Bash: перенаправление ввода-вывода	130
6.8	Примеры	131
6.9	Задания для самостоятельной работы	137
6.10	Практикум на ЭВМ	139

7	Цикл for	141
7.1	Цикл for	141
7.2	Условие продолжения цикла	143
7.3	Переменные счётчики	144
7.4	Вложенные циклы	145
7.5	Перебор всех пар	149
7.6	Проблема общего счётчика во вложенных циклах	150
7.7	Оператор «запятая»	151
7.8	Условный оператор и условие продолжение цикла	152
7.9	Бесконечный цикл, break, continue	153
7.10	Примеры	155
7.11	Задания для самостоятельной работы	162
7.12	Практикум на ЭВМ	164
8	Массивы	166
8.1	Одномерный массив	166
8.2	Применение массивов	167
8.3	Размер массива в задачах	168
8.4	Размеры переменных	170
8.5	Адреса	170
8.6	Копирование и сравнение	172
8.7	Ваш первый segmentation fault	173
8.8	Санитайзеры	175
8.9	Многомерные массивы	175
8.10	Матрицы	176
8.11	Адреса в двумерном массиве	177
8.12	Первая размерность многомерных массивов	178
8.13	Примеры	178
8.14	Задания для самостоятельной работы	191
8.15	Практикум на ЭВМ	193
9	Указатели и строки	196
9.1	Взятие адреса	196
9.2	Разыменование указателя	197
9.3	Ваш второй segmentation fault	199
9.4	Арифметика указателей	200
9.5	Приоритет	200
9.6	Работа с массивами через указатели	201
9.7	Полезные соотношения	202
9.8	Приведение указателей	203
9.9	Вывод адресов	204

9.10	Строка	204
9.11	Ввод–вывод строк с использованием <code>stdio.h</code>	205
9.12	Особенности ввода–вывода	207
9.13	Библиотека « <code>ctype.h</code> »	208
9.14	Библиотека « <code>string.h</code> »	209
9.15	Длина строки	210
9.16	Сравнение строк	211
9.17	Поиск подстроки в строке	214
9.18	Копирование строк	217
9.19	Заполнение массива	218
9.20	Справочная система <code>man</code>	218
9.21	Примеры	219
9.22	Задания для самостоятельной работы	225
9.23	Практикум на ЭВМ	227
10	Функции	229
10.1	Собственная функция	229
10.2	Прототип функции	229
10.3	Тело функции и возвращаемое значение	232
10.4	Функции без аргументов	234
10.5	Функции без возвращаемого значения	235
10.6	Несколько операторов <code>return</code>	236
10.7	Использование результатов вычисления функции	237
10.8	Приведение типа аргумента	237
10.9	Локальные переменные	238
10.10	Глобальные переменные	240
10.11	Передача аргументов по значению и по указателю	241
10.12	Массив как аргумент функции	243
10.13	Матрица как аргумент функции	245
10.14	Макроопределение типа	246
10.15	Примеры	247
10.16	Задания для самостоятельной работы	255
10.17	Практикум на ЭВМ	258
11	Функция в качестве аргумента. Рекурсия	261
11.1	Функция как аргумент	261
11.2	Системный стек	262
11.3	Рекурсия	264
11.4	Хвостовая рекурсия	266
11.5	Системный стек как хранилище данных	267
11.6	Избыточные вычисления при рекурсии	269

11.7	Примеры	270
11.8	Задания для самостоятельной работы	277
11.9	Практикум на ЭВМ	279
12	Структуры	283
12.1	Описание структуры	283
12.2	Объявление переменных	284
12.3	Инициализация структур	285
12.4	Расположение в памяти	288
12.5	Оператор копирования	289
12.6	Передача по значению	289
12.7	Передача по указателю	291
12.8	Оператор стрелка	291
12.9	Макроопределение типа	293
12.10	Примеры	294
12.11	Задания для самостоятельной работы	304
12.12	Практикум на ЭВМ	305
13	Указатели. Динамические массивы.	309
13.1	Одна переменная	309
13.2	Массив	310
13.3	Функции и динамическая память	311
13.4	Альтернативная функция для массивов	312
13.5	Неопределенное поведение программы	313
13.6	Расширение или уменьшение массива	313
13.7	Массив для буферизации	315
13.8	Динамические структуры	317
13.9	Сравнение статической и динамической памяти	317
13.10	Передача аргументов функции по указателю	318
13.11	Примеры	321
13.12	Задания для самостоятельной работы	328
13.13	Практикум на ЭВМ	330
14	Динамические матрицы. Аргументы командной строки	334
14.1	Статические матрицы	334
14.2	Статические матрицы как аргументы функции	334
14.3	Обнуление строки матрицы	336
14.4	Два способа представления динамических матриц	336
14.5	Быстрая перестановка строк	338
14.6	Динамические матрицы как аргументы функции	339
14.7	Отличия динамической и статической матрицы	342
14.8	Массив строк	342

14.9	Аргументы командной строки	347
14.10	Преобразование чисел и строк	348
14.11	Задания для самостоятельной работы	352
14.12	Практикум на ЭВМ	353
15	Файлы	356
15.1	Открытие файла	356
15.2	Чтение и запись	357
15.3	Конец файла	358
15.4	Перенаправление потоков ввода–вывода	361
15.5	Пример работы с графическим файлом	363
15.6	Задания для самостоятельной работы	371
15.7	Практикум на ЭВМ	372
16	Ответы к проверочным работам	375
17	Задания для подготовки к промежуточному контролю	389
17.1	Теоретическая часть	389
17.2	Практическая часть	397
18	Основная литература	402
19	Дополнительная литература	402

Предисловие

Дисциплина «Технология программирования и работа на ЭВМ» является важной составной частью в подготовке бакалавров по направлению «Математика» в Казахстанском филиале МГУ имени М.В.Ломоносова. Основная цель данной дисциплины — научить студентов реализовывать математические методы и алгоритмы с использованием языков С и С++. «Технология программирования и работа на ЭВМ» преподается в течение первых четырех семестров:

- в первом семестре рассматриваются базовые конструкции языка С и работа в терминале среды UNIX;
- во втором семестре изучаются некоторые классические задачи, алгоритмы их решения и реализации на языке С с подготовкой отчетов в системе вёрстки «LaTeX»;
- третий семестр посвящен объектно-ориентированной парадигме с реализацией алгоритмов и структур данных на языке С++;
- в четвертом семестре студенты изучают особенности параллельного программирования на примере технологии pthread для решения математических задач.

Данное пособие предназначено для работы в первом семестре. Материал рассчитан на студентов, не имеющих опыта программирования на языке С. Техническое введение даёт возможность познакомиться с работой в терминале на уровне достаточном для выполнения практических заданий. Пособие включает в себя 14 тематических блоков. Каждый блок содержит теоретический материал, задачи с подробными решениями и задания для самостоятельной работы.

Итогом освоения данной дисциплины является приобретение следующих общепрофессиональных и профессиональных компетенций:

- способности находить, анализировать, реализовывать программно и использовать на практике математические алгоритмы, в том числе с применением современных вычислительных систем;
- способности использовать методы математического и алгоритмического моделирования при решении теоретических и прикладных задач.

В результате освоения дисциплины в первом семестре студент должен знать:

-
1. синтаксис языка C;
 2. технологию разработки программ на языке C под UNIX;
 3. методы отладки программ на C под UNIX.

В результате освоения дисциплины студент должен уметь:

1. работать в терминале под UNIX без графического интерфейса;
2. составлять и реализовывать алгоритм программы на основе математической постановки задачи;
3. отлаживать код на C в терминале.

Пособие имеет свою специфику: автор акцентирует внимание на разбор часто встречающихся ошибок в коде у студентов при выполнении практических заданий. Также в примерах даются методические рекомендации для написания эффективного кода и разбираются базовые алгоритмы. Кроме этого пособие содержит технические рекомендации по оформлению кода в хорошем стиле.

По материалам, представленным в пособии, обучались студенты первого курса Казахстанского филиала МГУ с 2015 года. На итоговую редакцию пособия в значительной мере повлияли отзывы студентов направления «Математика»: Болотникова Дмитрия, Газизова Куата, Киселёва Ильи, Коробова Павла, Михно Ксении, Пикулиной Алисы, Оспанова Тимура, Савицкой Нины и Тасжанова Тимура. Отдельная благодарность студенту направления «Прикладная математика и информатика» Кабдуали Беку за ценные замечания.

Выражаю глубочайшую благодарность любимой жене, Баевой Юлие, за постоянную поддержку во время создания пособия и активную помощь в подготовке пособия к публикации.

Если Вы обнаружите ошибки и опечатки или захотите высказать мнение о пособии, пишите на почту: bayev.alen@yandex.kz.

1 Техническое введение.

Терминал и консольные редакторы

Для выполнения практических заданий в течение всего цикла обучения настоятельно рекомендуется использовать консольный терминал типа `bash`. Использование терминала упрощает написание и отладку консольных программ.

1.1 Операционная система

В качестве операционной системы рекомендуется использовать систему семейства UNIX, где работа в терминале является более естественной в отличие от систем семейства Windows. Начинающим студентам, которые ещё не работали с UNIX, предлагается использовать Ubuntu или macOS.

Процесс установки системы обычно занимает не более одного вечера. Наиболее правильный вариант, по личному мнению автора, — установить ОС Ubuntu в качестве второй системы на компьютер (ноутбук). Установку можно провести по инструкциям из статьи «Домашнее задание №1 для 1 курса: установить Ubuntu, не сломав Windows», которую легко найти в интернете. В качестве альтернативы есть возможность установить вторую операционную систему на виртуальную машину или непосредственно в самой ОС Windows как приложение.

Для успешного освоения необходимо стараться выполнять все привычные действия с файлами и директориями через терминал, о котором и пойдёт речь ниже.

1.2 Терминал

Работа в терминале позволяет ускорить и автоматизировать большинство действий, которые пользователи привыкли делать мышкой. После интенсивной работы в терминале в течение нескольких недель работа по набору и отладке кода ускорится в разы.

Перед началом работы запустим эмулятор терминала комбинацией клавиш `ctrl + alt + t`. Если комбинация не сработала, то можно запустить терминал через меню программ (называется `terminal`, `xterm`, `konsole` или похожим образом). Откроется терминал с приветствием:

```
user@user-notebook:~$
```

где

user	имя пользователя,
@	разделитель,
user-notebook	название компьютера,
:	разделитель,
~	текущее положение,
\$	разделитель.

Символом тильда (`~`) обозначается домашний каталог (вероятнее всего эту будет `/home/user/`). Здесь располагаются личные документы и файлы настроек пользователя с именем **user**.

После данного приветствия будет расположен мигающий курсор, который означает готовность терминала к вводу команд. Далее везде в пособии символ **\$** будет означать, что текст вводится в терминале (сам символ вводить не нужно). По окончании ввода команды её необходимо подтвердить нажатием клавиши **enter**. Рассмотрим некоторые команды на примерах.

Узнать текущее местоположение (англ. `print working directory`):

```
$ pwd
```

Вывод будет соответствовать полному адресу текущего каталога (он же директория), в котором Вы сейчас находитесь:

```
/home/user
```

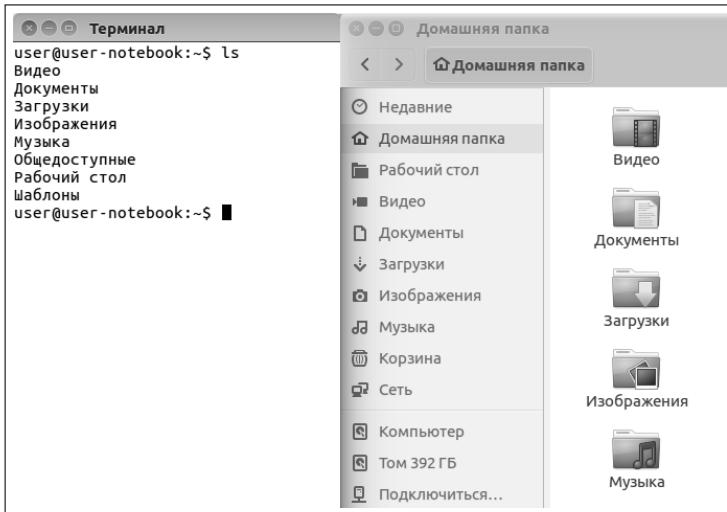
Обратите внимание, что вложение директории отмечается прямым слешем (символ `/`), в отличие от системы семейства windows, где разделителя является обратный слеш.

Посмотреть содержимое каталога (англ. `list`):

```
$ ls
```

При работе с русскоязычной версией системы на выводе будет список директорий, которые есть по умолчанию в домашнем каталоге: Видео, Документы, Загрузки, Изображения, Музыка, Общедоступные, Рабочий стол и Шаблоны.

Для лучшего восприятия следующих команд рекомендуется открыть файловый менеджер (аналог программы «проводник») и разместить рядом с терминалом так, чтобы одновременно было видно происходящее и в терминале, и в графическом файловом менеджере:



Помимо простых вызовов команд, некоторые команды обрабатывают дополнительные аргументы. Например, чтобы создать директорию, необходимо указать её название после соответствующей команды **mkdir** (англ. make directory):

```
$ mkdir mecmath
```

В итоге в домашней директории появится каталог **mecmath**. В этом можно убедиться, используя команду

```
$ ls
```

Результат выполнения команды можно сохранить и в текстовый файл. Например, узнать текущую дату и время (англ. date):

```
$ date
```

Результат будет выглядеть примерно так:

```
Sun September 8 16:16:16 +06 2019
```

Чтобы результат был записан в файл **result.txt** (имя файла можно выбрать любое), следует сделать перенаправление вывода с помощью символа больше:

```
$ date > result.txt
```

Если файла **result.txt** нет, то он будет создан и в него будет записан результат команды **date**. В противном случае его содержимое будет удалено и также будет записан результат **date**.

Чтобы посмотреть содержимое текстового файла, не обязательно его открывать в каком-либо редакторе. Достаточно воспользоваться командой для вывода одного или нескольких файлов (англ. concatenate):

```
$ cat result.txt
```

на экране появится содержимое файла

```
Sun September 8 16:16:16 +06 2019
```

Стоит обратить внимание, что файл должен быть именно текстовый. То есть просмотреть таким образом содержимое из файлов, созданных в офисных редакторах (**.odt**, **.docx**) не получится, так как последние являются **zip**-архивами, а не текстовыми файлами.

Если команда ожидает на вход некоторый текст, то его можно подать из файла. Например, данная команда (англ. stream editor) заменяет в тексте каждую цифру 6 на цифру 7:

```
$ sed 's/6/7/g'
```

То есть на вход

```
2016-2026
```

команда вернет

```
2017-2027
```

Команда **sed** не завершается после обработки одной строки. Чтобы её завершить и вернуться к приветствию, нужно нажать комбинацию **ctrl + d**.

А теперь давайте заменим все цифры 6 в дате, которую мы сохранили в файле **result.txt**.

```
$ sed 's/6/7/g' < result.txt
```

на экране появится содержимое файла

```
Sun September 8 17:17:17 +07 2019
```

Скопировать и переместить файл можно командами **cp** (от англ. copy) и **mv** (от англ. move) соответственно. Для этого нужно указать файл-источник и файл-назначение:


```
$ cp result.txt copy.txt
```

И копировать, и перемещать можно задавая путь до файла с использованием / (прямой слеш) в качестве разделителя директорий:

```
$ mv copy.txt mechmath/copy.txt
```

Кстати, команда `mv` часто используется и для переименования файла (перемещение файла в эту же директорию под другим именем):

```
$ mv result.txt date.txt
```

Если предстоит работать с файлами некоторой директории, то удобнее сменить текущую директорию командой `cd` (англ. **change directory**):

```
$ cd mechmath
```

Обратите внимание, что после смены текущей директории у Вас поменяется и приветствие:

```
user@user-notebook:~/mechmath$
```

Убедимся, что файл `copy.txt`, который был отправлен в директорию `mechmath` находится на месте:

```
$ ls
```

Для адресации родительской директории используется две точки (`..`). То есть, чтобы подняться на каталог выше, следует набрать:

```
$ cd ..
```

Чтобы не допускать опечаток в длинных командах, следует активно использовать две вспомогательные опции терминала:

- история команд (клавиши «вверх» и «вниз»);
- автодополнение (клавиша «табуляция»).

Если в терминале нажать клавишу «вверх», то появится последняя введенная команда (то есть `cd ..`). Если нажать клавишу «вверх» повторно — то предпоследняя команда (`ls`) и так далее (на третье нажатие появится команда `cd mechmath`). Клавиша «вниз» перебирает историю команд в обратном направлении.

Если набрать команду (например, `cd`) и несколько первых букв директории (например, `m`), то нажатие клавиши табуляции приведет к поиску

файлов и директорий, начинающих с данной буквы. Если такой файл или директория будут определяться однозначно, то терминал допишет его имя автоматически. В противном случае можно нажать табуляцию ещё раз, после чего терминал выдаст все возможные варианты дополнения.

Напоследок удалим все файлы и директорию, которые были созданы в процессе наших действий. Для этого вернёмся в домашнюю директорию:

```
$ cd
```

Удаляем файл:

```
$ rm mechmath/copy.txt
```

При удалении нужно быть крайне внимательным. Удаление в таком виде — процесс безвозвратный (то есть файлы не попадают в корзину, а удаляются безвозвратно).

Удаляем директорию:

```
$ rmdir mechmath
```

Таким способом можно удалять только пустую директорию. Если же в директории есть файлы, их можно удалить так (символом «звездочка» обозначается любое имя):

```
$ rm mechmath/*
```

В терминале можно комфортно работать и с внешними устройствами типа USB флеш-накопителей. Когда флешка вставляется в порт, операционная система, как правило, самостоятельно монтирует её. Это означает, что появляется каталог, который повторяет всю внутреннюю структуру флешки. В системах семейства Ubuntu она появляется в каталоге `/media/user/xxxx-xxxx/`, где вместо `user` будет имя текущего пользователя, а вместо `x` — некоторые достаточно случайные цифры или имя флешки (если оно было задано). Чтобы выбрать нужную флешку, достаточно просмотреть содержимое командой `ls` (для перебора доступных названий используйте автодополнение):

```
$ ls /media/user/7454-7202/
```

Скопировать документ на флешку можно так:

```
$ cp result.txt /media/user/7454-7202/
```

Скопировать каталог с документами на флешку можно с помощью флага рекурсивного копирования (`-r`):

```
$ cp -r mecmath /media/user/7454-7202/
```

После любых действий по изменению содержимого флешки (копирование или удаление файлов), её следует отключить программно, прежде чем отключать физически. Аналогом безопасного извлечения является размонтирование:

```
$ umount /media/user/7454-7202/
```

Для любой команды имеется справка, которую можно прочесть в самом терминале. Например, если Вы хотите узнать, как получить дополнительную информацию при командой `ls`, обратитесь к «манам» (англ. manual — руководство):

```
$ man ls
```

Например, в справке к команде `ls` написано:

```
-l use a long listing format
```

Это означает, что запуск с флагом `-l`

```
$ ls -l
```

выдаст список файлов в более подробном формате. Выйти из справки можно с помощью нажатия клавиши `q`.

Стоит отметить, что выход из активных программ (справка, монитор ресурсов, текстовый редактор), может быть не всегда очевиден. В простых приложениях часто эту функцию выполняет клавиша `q`. Но бывают и сложные варианты. Пожалуй, самая известная шутка про работу в терминале связана с выходом из редактора `vim`, где требуется фантастическая для новичков эрудиция, чтобы найти последовательность: `esc`, `:q`, `enter`.

Очень мощным средством являются конвейеры (обозначены символом `|`, которые позволяют вызывать несколько команд обработки текста друг за другом. В таблице 3 приведены самые базовые примеры таких команд. Также стоит отметить, что `bash` является скриптовым языком программирования. Данные аспекты предлагается изучить интересующимся самостоятельно: https://www.opennet.ru/docs/RUS/bash_scripting_guide/

В качестве необычных для новичков программ можно посмотреть программы из Таблицы 5 (необходимо установить дополнительные пакеты).

действие	команда
путь до текущей директории	<code>pwd</code>
содержимое текущей директории	<code>ls</code>
содержимое директории <code>dir</code>	<code>ls dir</code>
перейти в директорию <code>dir</code>	<code>cd dir</code>
перейти на уровень вверх	<code>cd ..</code>
перейти в предыдущую директорию	<code>cd -</code>
перейти в домашнюю директорию	<code>cd</code>

Таблица 1: bash: навигация

действие	команда
создать файл <code>file.txt</code>	<code>touch file.txt</code>
удалить файл <code>file.txt</code>	<code>rm file.txt</code>
создать директорию <code>dir</code>	<code>mkdir dir</code>
удалить пустую директорию <code>dir</code>	<code>rmdir dir</code>
удалить директорию <code>dir</code>	<code>rm dir -r</code>
скопировать файл <code>from.txt</code> в <code>to.txt</code>	<code>cp from.txt to.txt</code>
скопировать директорию <code>from</code> в <code>to</code>	<code>cp from to -r</code>
переместить (переименовать) <code>from.txt</code> в <code>to.txt</code>	<code>mv from.txt to.txt</code>
получить статистику файла <code>file.txt</code> (строки, слова, символы)	<code>wc file.txt</code>
сравнить два файла <code>left.txt</code> и <code>right.txt</code>	<code>diff left.txt right.txt</code>
найти файл <code>file.txt</code> в директории <code>dir</code>	<code>find dir -name file.txt</code>

Таблица 2: bash: работа с файлами

действие	команда
записать вывод команды <code>cmd</code> в файл <code>file.txt</code>	<code>cmd >file.txt</code>
направить на ввод команды <code>cmd</code> данные из файла <code>file.txt</code>	<code>cmd <file.txt</code>
вывести содержимое файла <code>file.txt</code>	<code>cat file.txt</code>
вывести начало файла <code>file.txt</code>	<code>head file.txt</code>
вывести конец файла <code>file.txt</code>	<code>tail file.txt</code>
отсортировать вывод команды <code>cmd</code>	<code>cmd sort</code>
выбрать строки вывода команды <code>cmd</code> с текстом <code>text</code>	<code>cmd grep text</code>
найти текст <code>foo</code> и заменить на <code>bar</code> в выводе команды <code>cmd</code>	<code>cmd sed 's/foo/bar/g'</code>

Таблица 3: bash: работа с текстовым потоком

действие	команда
свободное место в каталоге <code>dir</code> (англ. disk free)	<code>df dir</code>
занятое место в каталоге <code>dir</code> (англ. disk usage)	<code>du dir</code>
все процессы (англ. processes)	<code>ps -aux</code>
наиболее активные процессы	<code>top</code>
завершить процесс под номером <code>id</code>	<code>kill id</code>
измерить время работы программы <code>cmd</code>	<code>time cmd</code>
установить пакет <code>pkg</code>	<code>sudo apt install pkg</code>
удалить пакет <code>pkg</code>	<code>sudo apt remove pkg</code>
скачать файл <code>url</code>	<code>wget url</code>

Таблица 4: bash: разное

действие	пакет	команда
проиграть аудио файл	sox	play audio.mp3
открыть web страницу	lynx	lynx http://msu.kz
распечатать файл на принтере		lpr file.pdf
посмотреть на паровоз	sl	sl

Таблица 5: bash: разное

1.3 Консольные текстовые редакторы

В качестве среды разработки рекомендуется использовать консольные текстовые редакторы: **nano** (более простой и имеющий минимальную функциональность) или **vi** (более сложный и имеющий очень продвинутую функциональность). Данные редакторы есть в большинстве дистрибутивов Linux по умолчанию. Использование графических IDE (даже таких простых как **gedit**) отвлекает посторонними действиями, которые новички постоянно норовят выполнять мышкой. Поэтому весь первый семестр рекомендуется использовать именно редактор консольного типа. Знакомство с **vi** лучше начать со специального тренажера **vimtutor**, который можно установить из репозитория:

```
$ sudo apt install vimtutor
```

Ниже приведено небольшое знакомство с редактором **nano**.

1.4 Nano: работа с файлами

Чтобы создать текстовый файл **hello.txt** в редакторе **nano** необходимо выполнить:

```
$ nano hello.txt
```

Если такого файла нет, то он будет создан (правда не сразу, а в момент сохранения).

Далее терминал преобразуется: наверху появится номер версии **nano** и имя открытого файла **hello.txt**, а внизу — описание горячих клавиш и соответствующие действия. Наберём текст: «Hello, great math student!» Можно заметить, что в правом верхнем углу появилась надпись «Изменён».

```

GNU nano 2.5.3      Файл: hello.txt      Изменён
Hello, great math student!

^G Помощь      ^O Записать      ^W Поиск      ^K Вырезать
^X Выход      ^R ЧитФайл      ^\ Замена      ^U Отмен. вырезку

```

Чтобы сохранить файл на диск, используем комбинацию клавиш `ctrl + o` (в подсказках это описано как `^O Записать`). В этот момент надпись «Изменён» исчезнет. Закрывает редактор комбинация `ctrl + x` (в подсказках это описано как `^X Выход`). Теперь мы снова оказались в терминале. Выведем содержимое текущего каталога командой

```
$ ls
```

В списке файлов появится файл `hello.txt`. Посмотрим его содержимое с помощью команды

```
$ cat hello.txt
```

В терминале увидим текст, который был набран в редакторе.

```
Hello, great math student!
```

Вот мы и познакомились с простейшим способом редактирования текста в текстовом файле.

1.5 Nano: редактирование файлов

Чтобы вырезать и вставить одну или несколько подряд идущих строк следует использовать комбинации `ctrl + k` и `ctrl + u` (в подсказках описаны как `^K Вырезать` и `^U Отмен.вырезку`). Давайте вырежем первую строку и вставим три таких же строки: нажимаем один раз `ctrl + k` (содержимое строки попадает в буфер редактора) и три раза нажимаем `ctrl + u`. Все эти действия можно сделать если зажать клавишу `ctrl` и последовательно нажать `k`, `u`, `u`, `u`). Должны получиться 3 одинаковые строки.

```

Hello, great math student!
Hello, great math student!
Hello, great math student!

```

Чтобы продолжить дальнейшие эксперименты, изменим строки:

```
Hello, student!  
Hello, math student!  
Hello, great math student!
```

Рассмотрим, как поменять порядок строк. Например, перенесем последнюю строку в самое начало. С помощью стрелок вверх и вниз наводим курсор на последнюю строку и вырезаем её (**ctrl** + **k**). Далее с помощью стрелок вверх и вниз переносим курсор за первую строку и вставляем (**ctrl** + **u**).

```
Hello, great math student!  
Hello, student!  
Hello, math student!
```

Вырезать можно не только одну строку, но и целый блок строк, идущих подряд. Давайте вернём исходный порядок данных строк, вырезав последние две и вставив их в начале. Сначала перейдем курсором на вторую строку и вырежем сразу две строки (удерживая клавишу **ctrl** дважды нажимаем клавишу **k**). Последние две строки попадут в буфер, на экране останется только первая строка:

```
Hello, great math student!
```

Далее переносим курсор на первую строку и производим вставку (**ctrl** + **u**). Получаем исходный порядок:

```
Hello, student!  
Hello, math student!  
Hello, great math student!
```

Также стоит отметить команды для выделения, копирования и вставки произвольного блока текста. Включить режим выделения произвольного блока можно командой **ctrl** + **6**. Далее стрелками выбираем нужный блок. Комбинацией **alt** + **6** копируем текст. А вставка выполняем уже привычной командой **ctrl** + **u**.

1.6 Nano: навигация по тексту

Отличительной особенностью любого редактора кода является то, что в нём используется моноширинный шрифт (то есть все буквы имеют одинаковый размер по ширине). То есть у каждой буквы есть своя позиция в строке и в столбце. Перемещаясь только по строкам, номер столбца не меняется. Это позволяет очень быстро ориентироваться в тексте, когда

Вам нужно найти соответствующую строку и столбец. Чтобы видеть, на какой по счету строке и в каком столбце находится курсор, необходимо запустить редактор с дополнительным флагом `-с`:

```
$ nano -с hello.txt
```

```
GNU nano 2.5.3      Файл: hello.txt      Изменён

Hello, student!
Hello, math student!
Hello, great math student!

[ строка 2/4(50%), ряд 8/21 (38%), символ 23/64 (35%) ]
^G Помощь      ^O Записать    ^W Поиск      ^K Вырезать
^X Выход       ^R ЧитФайл    ^_ Замена     ^U Отмен. вырезку
```

После такого запуска информация о положении курсора будет отображаться над справкой (буква `m` под курсором расположена на 2 строке из 4 в 8 столбце из 21).

Для быстрого перемещения курсора по строкам лучше пользоваться комбинацией `ctrl + shift + minus` (минус в верхнем ряду основной клавиатуры, справа от нуля). После чего редактор предложит указать номер строки, на которую необходимо перейти.

Для быстрого перемещения курсора по столбцам можно использовать комбинации `,ctrl + left` (стрелка влево) и `ctrl + right` (стрелка вправо), которые позволяют смещаться сразу на целое слово.

1.7 Nano: файл настройки

У многих программ, которые запускается в системе, есть файлы настройки. Эти файлы содержат команды, которые выполняют настройку программы перед её запуском. Называются такие файлы `.PROGRAMrc`, где вместо `PROGRAM` записано имя программы. Точка в начале файла обозначает, что файл скрытый и его не видно по умолчанию (увидеть скрытые файлы можно командой `ls -a`). А последние буквы `rc` означает `run commands` (выполни команды при запуске приложения). Чтобы настроить `nano`, откроем его файл настроек:

```
$ nano .nanorc
```

По умолчанию он пустой. Чтобы `nano` показывал номера строк, делал умные переносы, заменял табуляцию на 4 пробелы, опишем настройки:

действие	комбинация клавиш
сохранить	<code>ctrl + o</code>
выйти	<code>ctrl + x</code>
начать выделение произвольного блока	<code>ctrl + 6</code>
вырезать строку или выделение	<code>ctrl + k</code>
вырезать блок строк	<code>ctrl + k + k + ...</code>
скопировать строку или выделение	<code>alt + 6</code>
скопировать блок строк	<code>alt + 6 + 6 + ...</code>
скопировать другой файл (табуляция)	<code>ctrl + r</code>
вставить строку	<code>ctrl + u</code>
отменить последнее действие	<code>alt + u</code>
уменьшить отступ строки или выделения	<code>alt + shift + [</code>
увеличить отступ строки или выделения	<code>alt + shift +]</code>
перейти к строке	<code>ctrl + shift + -</code>
перейти к следующему слову	<code>ctrl + →</code>
перейти к предыдущему слову	<code>ctrl + ←</code>
найти текст	<code>ctrl + w</code>
найти и заменить текст	<code>ctrl + \</code>
подробная справка	<code>ctrl + g</code>

Таблица 6: nano: базовые команды, редактирование и навигация

```
set linenumbers
set autoindent
set tabstospaces
set tabsize 4
```

Теперь достаточно перезапустить **nano** и увидеть разницу.

Здесь приведён список наиболее полезных команд по личному опыту автора. Напоследок хочется напомнить одну простую мысль: не нужно учить все комбинации, нужно их использовать.

Для более подробной информации полезно почитать справку, которая открывается и закрывается комбинацией `ctrl + g`.

1.8 Задания для самостоятельной работы

Выполните последовательно действия в терминале. Запишите все команды, набранные в терминале, и комбинации клавиш, использованные в текстовом редакторе во время выполнения задания.

1. В текущей директории создать директорию `Lion` и перейти в директорию `Lion`.
2. Открыть в редакторе `nano` файл `Timon.txt`. Записать текст

```
<<Awimbawe>>
```

Выйти из редактора.

3. Скопировать содержимое файла `Timon.txt` в файл `Pumba.txt`.
4. Открыть в редакторе `nano` файл `Pumba.txt` и дописать текст

```
<<Hakuna Matata>>}
```

три раза в конце файла. Сохранить файл и выйти.

5. Вывести текущую директорию и список файлов в ней.
6. Распечатать содержимое файлов `Pumba.txt` и `Timon.txt`.
7. Распечатать содержимое флешки, если считать, что подключено только одно внешнее устройство с файлами.
8. Скопировать на флешку файл `Pumba.txt`. Безопасно извлечь флешку.
9. Распечатать содержимое файла `Pumba.txt` в терминале с заменой всех слов `tata` на `th`.
10. Удалить все созданные директории и файлы.

2 Числовые типы.

Арифметические операторы

Сердце центрального процессора называется арифметико-логическим устройством, потому что по сути, процессор — это очень умный калькулятор. А программы, которые мы будем писать — это инструкции для этого калькулятора. Прежде чем начать писать инструкции, познакомимся с главными действующими лицами: с числами и арифметическими операциями над ними. Для лучшего понимания, стоит вспомнить позиционные системы счисления (особенно двоичную и шестнадцатеричную).

2.1 Целочисленные типы

Целые числа, представимые в компьютере, бывают разных типов. Во-первых, они бывают беззнаковые (принимают только неотрицательные значения) и знаковые (принимают и положительные, и отрицательные значения). Во-вторых, они различаются по размеру занимаемой памяти (1, 2, 4 или 8 байт).

Начнем с беззнаковых типов. Числа такого типа в компьютере хранятся в двоичной системе счисления, где под каждую двоичную цифру выделяется один бит. С учётом того, что в 1 байте содержится 8 бит получаем, что для чисел размера 1, 2, 4 и 8 байта максимальное количество двоичных цифр равно 8, 16, 32 и 64 соответственно. Например, число 13 в двоичной системе счисления равно 1101_2 . Если для хранения данного числа выбрать `unsigned char` (беззнаковый целочисленный тип в 1 байт, то есть 8 бит), то храниться оно будет в виде:

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

Если для хранения данного числа выбрать `unsigned int` (беззнаковый целочисленный тип в 4 байта, то есть 32 бита), то храниться оно будет в виде:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Для m -битного беззнакового числа максимальное значение равно $2^m - 1$. Например, числа типа `unsigned int` принимают значения от 0 до $2^{32} - 1$.

тип	байт	минимум	максимум
unsigned char	1	0	$2^8 - 1$ 255
unsigned short	2	0	$2^{16} - 1$ 65 535
unsigned int	4	0	$2^{32} - 1$ 4 294 967 295
unsigned long long	8	0	$2^{64} - 1$ 18 446 744 073 709 551 615

Беззнаковые типы, как правило, используются для битовых операций.

Перейдем к знаковым типам. У таких чисел половина всех значений является отрицательными числами, а половина — неотрицательными.

тип	байт	минимум	максимум
char	1	-2^7 -128	$2^7 - 1$ 127
short	2	-2^{15} -32 768	$2^{15} - 1$ 32 767
int	4	-2^{31} -2 147 483 648	$2^{31} - 1$ 2 147 483 647
long long	8	-2^{63} -9 223 372 036 854 775 808	$2^{63} - 1$ 9 223 372 036 854 775 807

Разумеется, учить приведённые диапазоны не нужно. На практике достаточно помнить, как они вычисляются и их приближённые оценки: `int` от -10^9 до 10^9 и `long long` от -10^{18} до 10^{18} .

2.2 Целочисленные константы

Числа в программе, которые задаются явно, называются константами. Обычные целые числа относятся к типу `int` (то есть знаковые 4-байтовые числа). Например, `42` — константа типа `int`. Суффикс `U` обозначает беззнаковый тип, то есть `42U` — константа типа `unsigned int`. Также есть возможность задать константы типа `long long`, для этого необходимо добавить суффикс `LL`. А префикс `0x` задаёт константу в 16-ричной системе счисления. Примеры одной и той же константы со значением 42:

- `42` — тип `int`;
- `42U` — тип `unsigned int`;

- 42LL — тип `long long`;
- 42ULL — тип `unsigned long long`;
- 0x2A — тип `int` в шестнадцатиричной системе счисления.

Для многих начинающих программистов не всегда понятно такое пристальное внимание к шестнадцатиричной системе счисления. Основная цель — компактно записывать большие значения. Например, чтобы не запоминать большую константу 4 294 967 295, ограничивающую тип `unsigned int` сверху, достаточно записать это число в шестнадцатиричной системе счисления:

$$4\,294\,967\,295 = 2^{32} - 1 = 16^8 - 1 = 1\,0000\,0000_{16} - 1 = FFFF\,FFFF_{16}.$$

Значит, константу можно объявить так:

- 0xFFFFFFFFU — максимальное число типа `unsigned int`;
- 0x7FFFFFFF — максимальное число типа `int`;
- -0x80000000 — минимальное число типа `int`.

Стоит помнить, что константа обязательно должна помещаться в соответствующий диапазон. Например, такие константы как 1000000000000 (десять нулей) или 100000000000000000000LL (двадцать нулей) будут ошибочными. При попытке использовать данные константы будет выдаваться предупреждение:

`overflow in implicit constant conversion`

2.3 Целочисленные арифметические операции

Для выполнения арифметических действий к числам применяют так называемые операторы. Например, чтобы вычислить сумму 5 и 3 нужно, написать выражение

$$5 + 3$$

Говоря языком программирования, здесь применяется оператор сложения к двум операндам: 5 и 3. Результат вычисления данного оператора равен 8. Для целых чисел определены 5 арифметических операторов:

- `a + b` сложение чисел *a* и *b*;
- `a - b` вычитание чисел *a* и *b*;

- `a * b` умножение чисел *a* и *b*;
- `a / b` целочисленное деление *a* на *b* (целая часть);
- `a % b` целочисленное деление *a* на *b* (остаток).

Если мы применяем оператора к числам одного типа, то и результат будет того же типа. Например:

- `1+2` даст результат 3 типа `int`,
- `2ULL * 3ULL` даст результат `6ULL` типа `unsigned long long`.

А какого типа будет результат, если мы сложим числа разного типа? Для этого все целочисленные типы упорядочены по «рангу», и применяется следующее правило: «меньший по рангу» операнд преобразуется к «большему по рангу» операнду. Ранг зависит, в первую очередь, от размера (чем больше размер, тем выше ранг), во вторую очередь — от наличия знака (знаковые имеют приоритет выше беззнаковых):

`unsigned char` → `char` → `unsigned short` → `short` →
→ `unsigned int` → `int` → `unsigned long long` → `long long`.

Приведём пример операторов сложения, вычитания и умножения для разных типов:

- `5 + 3LL` преобразуется к операции `5LL + 3LL` и даст результат `8LL`;
- `5LL - 3` преобразуется к операции `5LL - 3LL` и даст результат `2LL`;
- `5U * 3ULL` преобразуется к операции `5ULL * 3ULL` и даст результат `15ULL`;
- `5 + 3U` преобразуется к операции `5U + 3U` и даст результат `8U`;
- `5ULL - 3` преобразуется к операции `5ULL - 3ULL` и даст результат `2ULL`.

Операторы деления представлены сразу в двух вариантах: для вычисления частного и для вычисления остатка при делении целых чисел. В случае с положительными операндами результаты вполне предсказуемы:

- `17 / 5` даст результат 3;
- `17 % 5` даст результат 2.

При делении отрицательных чисел на положительные результат равен противоположному числу от деления этих же чисел по модулю:

- $-17 / 5$ даст результат -3 ;
- $-17 \% 5$ даст результат -2 .

При целочисленном делении на отрицательные числа округление производится не самым очевидным образом, поэтому рекомендуется использовать только деление на положительные числа. А в случае целочисленного деления на нуль во время выполнения программы произойдет её остановка и будет выдано не совсем ожидаемое сообщение:

`Floating Point Exception (core dumped)`

«Исключение в операции с плавающей точкой (сделан дамп памяти)». Объясняется такое сообщение тем, что при переполнении операций над вещественными числами также происходит деление на нуль. Поэтому, хоть это и не связано с числами с плавающей запятой, при таком сообщении стоит проверить деление целых чисел на корректность.

2.4 Переполнение

Все арифметические действия производятся по модулю 2^m , где m — число бит для данного типа. Причем в качестве результата выбирается такое число по модулю, которое лежит в допустимом диапазоне значений данного типа. То есть беззнаковые числа зациклены так:

$$\begin{aligned} \dots &\rightarrow 4294967294 \rightarrow 4294967295 \rightarrow 0 \rightarrow 1 \rightarrow \\ &\rightarrow 2 \rightarrow 3 \rightarrow \dots \rightarrow 4294967294 \rightarrow 4294967295 \rightarrow 0 \rightarrow 1 \rightarrow \dots \end{aligned}$$

При арифметических операциях с беззнаковыми числами возможны переполнения следующего вида:

- $0U - 1U$ даст результат $4294967295U$.

Это не является ошибкой с точки зрения языка и может быть использовано в корректных целях, но об этом всегда стоит помнить.

Знаковые числа зациклены так:

$$\begin{aligned} \dots &\rightarrow +2147483646 \rightarrow +2147483647 \rightarrow -2147483648 \rightarrow -2147483647 \rightarrow \dots \\ &\dots \rightarrow -2 \rightarrow -1 \rightarrow 0 \rightarrow +1 \rightarrow +2 \rightarrow \dots \\ \dots &\rightarrow +2147483646 \rightarrow +2147483647 \rightarrow -2147483648 \rightarrow -2147483647 \rightarrow \dots \end{aligned}$$

Например, при сложении двух положительных знаковых чисел, может получиться отрицательное число (это наиболее распространённый тип переполнения у новичков):

- $2000000000 + 1000000000$ даст результат -1294967296

(результат даёт такой же остаток, что и 3000000000 при делении на 2^{32}).

2.5 Вещественные типы

Так как вещественных чисел на любом конечном отрезке несчётно, то хранить все числа из диапазона в конечных ресурсах компьютера невозможно. В компьютере точно представляются только рациональные числа с конечным числом знаков после запятой в двоичной системе счисления, причем количество знаков тоже ограничено. Каждое число представляется в нормализованном виде. Например, число $12,5 = 1100,1_2 = 1,1001_2 \cdot 2^3$. Число $m = 1001$ называется мантиссой (дробная часть нормализованного числа), а число $e = 3$ — экспонентой (количество позиций, на которое «уплывает» запятая). То есть вещественное число x представляется в виде:

$$x = (1 + m) \cdot 2^e.$$

Числа при таком подходе расположены неравномерно. Например, числа около единицы идут с минимальным шагом, который значительно меньше 1 («машинный эпсилон»). Числа, близкие к максимальным значениям, идут шагом больше 1.

В языке C есть несколько вещественных типов, которые различаются по объёму занимаемой памяти и, соответственно, точности представления чисел. В рамках курса будет рассмотрены вещественные числа одинарной и двойной точности.

тип	размер	мантисса	экспонента	диапазон целых	максимум
float	4 байта	23 бита	8 бит	2^{24} 16 777 216	$\approx 10^{38}$
double	8 байт	52 бита	11 бит	2^{53} 9 007 199 254 740 992	$\approx 10^{308}$

Для вещественных типов интересно не только максимальное и минимальное вещественное значение, но и непрерывный диапазон целых значений. Например, все целые числа до числа от $-16\,777\,216$ до $16\,777\,216$ можно представить в типе `float`, а число $16\,777\,217$ — нельзя.

2.6 Вещественные константы

Обычные вещественные константы имеют тип `double`. Разделителем целой и дробной части выступает точка, а не запятая. Для задания константы типа `float` необходимо добавить суффикс `f`. Также возможно задание чисел в «научном стиле» с помощью десятичной экспоненты. Для этого необходимо использовать суффикс `e`, после которого указать степень десятки. Примеры:

- 12.3 — тип `double` со значением 12,3;
- 12.3f — тип `float` со значением 12,3;
- 12. — тип `double` со значением 12,0;
- 12.f — тип `float` со значением 12,0;
- 123e-1 — тип `double` со значением 12,3;
- 12e3 — тип `double` со значением 12 000;
- -12e3 — тип `double` со значением -12 000;
- 12e-3 — тип `double` со значением 0,012;
- -12e-3 — тип `double` со значением -0,012.

2.7 Вещественные арифметические операции

Для вещественных чисел определены 4 арифметических оператора:

- `a + b` сложение чисел *a* и *b*;
- `a - b` вычитание чисел *a* и *b*;
- `a * b` умножение чисел *a* и *b*;
- `a / b` вещественное деление *a* на *b*.

Результат применения оператора к числам одинакового вещественного типа будет того же типа. Применение оператора к числам разного типа аналогично целочисленным операторам: «меньший по рангу» операнд преобразуется к «большему по рангу» операнду. Вещественные типы упорядочены по размеру. А любой вещественный тип имеет ранг выше целочисленного.

`... → unsigned long long → long long → float → double.`

Ниже приведены примеры операторов сложения, вычитания, умножения и деления для разных типов:

- `5.0 + 3LL` преобразуется к операции `5.0 + 3.0` и даст результат `8.0`;
- `5.f - 3` преобразуется к операции `5.f - 3.f` и даст результат `2.f`;

- $5e-1 * 3ULL$ преобразуется к операции $0.5 * 3.0$ даст результат 1.5;
- $5 / 3.0$ преобразуется к операции $5.0 / 3.0$ даст результат около 1.666667.

Деление с остатком для вещественных чисел не определено. Попытка вычисления значения $5 \% 3.0$ приведет к ошибке

`error: invalid operands to binary %`

При арифметических операциях с вещественными числами не стоит забывать о том, что числа хранятся приближенно. Например:

- $16777216.f + 1.f$ даст $16777216.f$

Так как вещественного числа со значением $16777217.f (= 2^{24} + 1)$ не бывает из-за ограничений мантииссы в типе `float`. В задачах, где вычисления можно выполнить в целых числах, лучше не прибегать к вещественным типам.

К особенностям арифметических операций над вещественными числами относится то, что деление на ноль не вызывает остановки программы. Результатом такого деления будет `inf` (бесконечность). Работать с такими значениями не имеет смысла, но они позволяют локализовать проблемные вычисления.

2.8 Приоритет арифметических операций

Приоритет операторов умножения и деления ($*$ / $\%$) выше, чем приоритет сложения и вычитания $+$ $-$. Здесь стоит заострить внимание на том, что оператор деления с остатком $\%$ имеет абсолютно такой же приоритет, что и вычисление целой части $/$ и умножение $*$. Операторы одного приоритета выполняются слева направо. Приоритет любого оператора можно повысить с помощью круглых скобок. Примеры:

- $1 / 3 + 2 / 3 + 3 / 3$

преобразуется к

$$((1 / 3) + (2 / 3)) + (3 / 3) = (0 + 0) + 1 = 1$$

- $(1 + 2 + 3) / 3$

преобразуется к

$$((1 + 2) + 3) / 3 = (3 + 3) / 3 = 6 / 3 = 2$$

- $1.0 / 3 + 2 / 3.0 + 3.0 / 3.0$

преобразуется к

$$((1.0 / 3.0) + (2.0 / 3.0)) + (3.0 / 3.0) = 2.00$$

- $12345 \% 100 / 10$

преобразуется к

$$(12345 \% 100) / 10 = 45 / 10 = 4$$

- $12345 / 10 \% 10$

преобразуется к

$$(12345 / 10) \% 10 = 1234 \% 10 = 4$$

- $16 / 5 - 16 \% 5 + 16 * 5$

преобразуется к

$$((16 / 5) - (16 \% 5)) + (16 * 5) = (3 - 1) + 80 = 82 .$$

Хороший стиль оформления кода. Арифметические операторы лучше отделять от операндов пробелами.

Многим начинающим программистам кажется, что разницы между таким кодом:

```
1+(2-3)*4
```

и таким кодом

```
1 + (2 - 3) * 4
```

почти нет. С точки зрения выдачи ответа, конечно — нет. А вот с точки зрения чтения кода другим человеком — есть. Например, сравните такой код:

```
1e-1-2e-3
```

и такой код:

```
1e-1 - 2e-3
```

Последний вариант значительно улучшает понимание записи (вычесть два вещественных числа).

2.9 Задания для самостоятельной работы

1. Определить, какие из данных констант больше 30000 (ответ обосновать):
 - а) 0xB0BA;
 - б) 0xFULL;
 - в) 0xCAFE?
2. Константы какого типа минимального размера следует использовать, если требуется:
 - а) сложить два целых числа 1 000 000 и 2 000 000;;
 - б) умножить два целых числа 1 000 000 и 2 000 000;
 - в) поделить одно вещественно число 1,0 пополам (точно);
 - г) сложить два вещественных числа $1\,000\,000\,000,0 + 2\,000\,000\,000,0$ точно?
3. Определить корректные варианты вычисления среднего арифметического чисел 3 и 6 (ответ обосновать):
 - а) $(3 + 6) / 2$;
 - б) $3U / 2U + 6 / 2$;
 - в) $3.0 / 2.0 + 6.0 / 2.0$;
 - г) $3.f / 2.f + 6 / 2$;
 - д) $3 / 2 + 6.f / 2.f$;
 - е) $(3.f + 6.f) / 2$;
 - ж) $(3 + 6) / 2LL$.
4. Определить **тип** результата и **объём памяти** в байтах для этого результата.
 - а) $1e2 * 2$;
 - б) $1 / 3$;
 - в) $1LL / 3f$.
5. Подписать **порядок** выполнения арифметических действий и **вычислить** значения:
 - а) $(1 + 3 + 4) / 3 * 3$;
 - б) $2 + 100 / 3 \% 25 * 4 - 1$;
 - в) $1e1+2e2-1e-1$;

3 Переменные. Приведение типов.

Первая программа

В рамках данной темы будет подробно разобран вопрос о переменных, операторе присваивания и приведении типов. Также будет написана первая программа, представлены средства стандартного ввода-вывода и описан основной цикл отладки программ.

3.1 Переменные

Программирования не было бы как такого, если бы программы представляли собой только операции над константами. Для минимальной полезной программы необходимо как минимум где-то хранить эти константы. У нас будет возможность хранить их в оперативной памяти компьютера (RAM). Чтобы **объявить** переменную в памяти, необходимо указать её тип и имя. В качестве типа можно использовать любой из уже рассмотренных типов целых чисел. Имя переменной — это любой набор букв, цифр и символа подчеркивания `_`. При этом имя не должно начинаться с цифры (чтобы не перепутать с константами, такими как `0xCAFE`) и не должно совпадать с уже существующими именами в программе. Не следует забывать, что строчные и заглавные буквы отличаются (то есть могут быть 2 различные переменные `x` и `X`, хоть это и крайне не рекомендуется делать). Например:

```
int x;
```

— выделить в памяти переменную размером 4 байта, к которой можно обращаться по имени `x`;

```
short tmp;
```

— выделить в памяти переменную размером 2 байта, к которой можно обращаться по имени `tmp`.

Обратите внимание, что объявление переменной — это инструкция языка C. Каждую такую инструкцию необходимо завершать символом `;` (точка с запятой). Разумеется, объявлять переменные необходимо до начала их использования.

Можно объявить сразу несколько переменных, если они одного типа:

```
long long a1, a2, a3;
```

— выделить 3 переменных по 8 байт каждая, к которым можно обращаться по именам `a1`, `a2` и `a3` соответственно;

```
double first_position, second_position;
```

— выделить 2 переменные по 8 байт, к которым можно обращаться по именам `first_position` и `second_position` соответственно.

Имена переменных должны нести в себе смысл, например, совпадать с обозначениями в математических формулах.

Если переменная состоит из нескольких слов, то слова либо разделяются подчеркиванием (так называемый «snake case»: `first_position`), либо каждое слово, начиная со второго, пишется с заглавной буквы (так называемый «camel case»: `firstPosition`).

Хороший стиль оформления кода. Все переменные в пределах одной программы должны быть названы в едином стиле («snake case» или «camel case»).

3.2 Оператор присваивания

После объявления переменной в соответствующую этой переменной память можно записать некоторое значение. Для этого в языке C есть оператор присваивания (англ. assign), который обозначается символом `=` (равно).

Чтобы **присвоить** переменной значение, достаточно написать её имя слева от оператора, а присваиваемое значение справа:

```
int x;  
x = 7;
```

В переменную `x` попадёт значение 7.

Так можно записать в переменную `first_position` значение 12.3 и скопировать это значение в переменную `second_position`.

```
double first_position, second_position;  
first_position = 12.3;  
second_position = first_position;
```

То есть в переменной `first_position` и `second_position` будут значения 12.3.

3.3 Инициализация при объявлении и мусорные значения

Оператор присваивания можно использовать непосредственно при объявлении переменной:

```
int x = 7;
double first_position = 12.3, second_position = 12.3;
```

Присвоить переменным **a** и **b** значения 5 и 12 соответственно, а затем поменять их значения местами.

```
int a = 5, b = 12, c;
c = a;
a = b;
b = c;
```

Неинициализированная переменная (в примере выше переменная **c**) изначально может иметь любое значение. Назовем такие значения «мусорными» и далее будем обозначать их вопросительным знаком.

Чтобы понимать, как работает данный код, следует выполнить его по шагам. Для этого запишем значения всех переменных после каждого действия:

строка	код	<i>a</i>	<i>b</i>	<i>c</i>
1	<code>int a = 5, b = 12, c;</code>	5	12	?
2	<code>c = a;</code>	5	12	5
3	<code>a = b;</code>	12	12	5
4	<code>b = c;</code>	12	5	5

Таким образом, в переменной **a** окажется число 12, а в переменной **b** будет число 5.

К теме «мусорных» значений можно вспомнить анекдот: Васе дали 2 яблока, сколько у него яблок?

```
int apples;
apples = apples + 2;
```

Ответ: сколько угодно, так как неизвестно изначальное количество яблок. Поэтому, переменную необходимо сначала инициализировать и только потом использовать значение.

3.4 Неизменяемые переменные

Если необходимо использовать одну и ту же константу, то хорошим тоном является хранить её неизменяемой переменной, дописав специальное слово **const**. Например:

```
const int month_per_year = 12;
```


Их значение можно инициализировать только при объявлении, в дальнейшем их изменять запрещено. Это используется для интервалов времени: число минут в часе, часов в сутках, месяцев в году. Таким же образом полезно обозначать большие константы, чтобы не ошибаться при каждом использовании. Например, если число миллион используется в программе более одного раза, то лучше объявить его как константу:

```
const int million = 1000000;
```

При попытке изменения таких переменных

```
const int million = 1000000;
million = 2000000;
```

будет выдана следующая ошибка:

```
$ error: assignment of read-only variable
```

3.5 Арифметические действия

Справа от оператора присваивания могут быть арифметические выражения. В таком случае сначала будет вычислено значение арифметического выражения, а потом это значение присвоено. В арифметических выражениях могут присутствовать как числа, так и переменные.

Пример 1. Найти периметр треугольника со сторонами 3, 4 и 5.

```
int a = 3, b = 4, c = 5, P;
P = a + b + c;
```

Сначала вычисляется $((3 + 4) + 5)$, результат присваивается переменной $P = 12$.

Пример 2. Присвоить переменным a и b значения 5 и 12 соответственно, а затем поменять их значения местами, не используя дополнительную переменную.

```
1 int a = 5, b = 12;
2 a = a + b;
3 b = a - b;
4 a = a - b;
```

Для лучшего понимания запишем значение всех переменных после каждого действия:

строка	код	<i>a</i>	<i>b</i>
1	<code>int a = 5, b = 12;</code>	5	12
2	<code>a = a + b;</code>	17	12
3	<code>b = a - b;</code>	17	5
4	<code>a = a - b;</code>	12	5

3.6 Укороченные арифметические операторы

Отдавая дань традициям низкоуровневого программирования, в языке C сохранились укороченные арифметические операторы. Например, чтобы увеличить значение переменной *x* на 2, можно использовать как полную инструкцию `x = x + 2;`, так и сокращенную `x += 2;`. В общем виде вместо операции:

```
a = a op (b);
```

можно использовать

```
a op= b;
```

причём в качестве аргумента *b* могут выступать константы, переменные и выражения. Например, вычислим значения переменных после выполнения операций:

```
int x = 5, y = 4;
y += 2;
y *= x;
x -= 3;
```

Для этого выпишем подробнее:

строка	код	<i>x</i>	<i>y</i>
1	<code>int x = 5, y = 3;</code>	5	4
2	<code>y = y + 2;</code>	5	6
3	<code>y = y * x;</code>	5	30
4	<code>x = x - 10;</code>	2	30

Ещё один пример с укороченными операторами:

```
int x = 2, y = 5;
x *= x + y;
y -= x / y;
```

Приведем укороченные операторы к обычным и выпишем последовательно все изменения переменных:

строка	код	x	y
1	<code>int x = 2, y = 5;</code>	2	5
2	<code>x = x * (x + y);</code>	14	3
3	<code>y = y - (x / y);</code>	14	-1

В отличие от математики в языке C есть унарные операторы: унарный минус и унарный плюс.

```
int x = 10, y;
y = -x;
y = +x;
```

Стоит отличать унарный минус перед константой (например, -5) и унарный минус перед переменной (например, $y = -x$). В первом случае он является частью константы, а во втором случае это оператор, который производит вычисления. Например, посчитаем количество арифметических операторов в следующих примерах.

Один оператор вычитания:

```
y = x - 3;
```

Один оператор сложения:

```
y = -3 + x;
```

Два оператора (унарный минус и сложение):

```
y = -x + 3;
```

Язык C позволяет компактно записать увеличение и уменьшение целочисленной переменной на единицу: операторы инкремента (`a++` и `++a`) и декремента (`a--` и `--a`). Разумеется, возникает вопрос, почему их по два вида. Оказывается, данные операторы можно использовать сразу для двух целей: не только изменить переменную, но и получить её значение.

Первый вид `a++` и `a--` (суффиксный) сначала возвращает старое значение переменной, а затем её изменяет. Второй вид `++a` и `--a` (префиксный) сначала изменяет значение переменной, а затем возвращает новое значение переменной. Пример суффиксного инкремента:

```
int x = 2, y;
y = x++;
```

После выполнения данного кода в переменной y будет 2, а в переменной x будет 3. Пример префиксного инкремента:

```
int x = 2, y;  
y = ++x;
```

После выполнения данного кода в обеих переменных будет 3.

Хороший стиль оформления кода. Каждый оператор присваивания и арифметического действия рекомендуется отделять от операндов пробелами слева и справа. Но это не относится к унарным операторам и операторам инкремента и декремента.

Хороший стиль оформления кода. На каждой строке должно быть не более одной завершённой инструкции.

Например:

```
a = b + 2;  
a++;
```

— хорошо;

```
a = b + 2; a ++;
```

— плохо.

3.7 Неявное преобразование типов

Один из видов неявного преобразования типов уже был рассмотрен на предыдущем занятии (тема «Числовые типы. Арифметические действия»), когда у арифметических операторов были аргументы разных типов. Сейчас же рассмотрим ещё один вариант неявного преобразования, происходящий во время выполнения оператора присваивания.

Например:

```
int a = 2.3;
```

Произойдёт преобразование типов: константа 2.3 типа `double` преобразуется к константе 2 типа `int`. Это даст такой же эффект, что и

```
int a = 2;
```

Чтобы присвоить целочисленной переменной значение 1 000 000 000, можно поступить следующим образом: задать константу `1e9` типа `double` и присвоить её в переменную типа `int`:

```
int a = 1e9;
```

Произойдёт преобразование типов `double` \rightarrow `int`. Это даст такой же эффект, что и

```
int a = 1000000000;
```

Рассмотрим преобразование знакового типа к беззнаковому. Если знаковое число x типа `int` неотрицательно, то оно кодируется в двоичной системе счисления обычным образом. А если оно отрицательное, то ему соответствует двоичная запись числа $2^{32} + x$ (параграф «Переполнение» предыдущей темы). Таким образом -1 кодируется как $2^{32} - 1$ (максимальное значение типа `unsigned int`). При следующем присваивании

```
unsigned int a = -1;
```

произойдет преобразование типов `int` \rightarrow `unsigned int` с потерей значения. Это приведет к такой операции:

```
unsigned int a = 4294967295;
```

Крайне осторожно нужно производить приведение типа при уменьшении ранга (например, `double` \rightarrow `int`), так как может происходить потеря точности. Например:

```
int a = 1234.5678;
```

приведет к

```
int a = 1234;
```

Таким образом можно инициализировать переменные маленького размера через константы типа `int`:

```
short a = 30000;
unsigned short a = 60000;
char ch = 100;
unsigned char = 200;
```

Пример 3. Присвоить переменным `a` и `b` значения 5 и 6 соответственно, а затем найти их среднее арифметическое.

Прежде чем решать задачи, всегда стоит очень внимательно подходить к выбору типа переменных. Например, что будет, если использовать все переменные типа `int`:

```
int a = 5, b = 6, m;  
m = (a + b) / 2;
```

Мы не получим искомого значения, так как $(5 + 6) / 2$ будет вычислено как 5. Ясно, что результат должен быть нецелым. Пусть теперь переменная `m` будет типа `double`:

```
int a = 5, b = 6;  
double m;  
m = (a + b) / 2;
```

Но и здесь результат будет неверным. Сначала производятся вычисления правой части и только потом присваивание. То есть результат справа будет вычислен как $(5 + 6) / 2 = 5$, а присваивание произойдет уже неправильного значения `m = 5`. Для решения проблемы достаточно заметить целочисленное деление на вещественное, то есть сделать так, чтобы один из аргументов был вещественный.

```
int a = 5, b = 6;  
double m;  
m = (a + b) / 2.0;
```

3.8 Явное преобразование типов

Если некоторую константу, переменную или выражение необходимо привести к данному типу, то достаточно написать имя типа в скобках перед константой, переменной или выражением.

- `(int) 5.3` равно 5;
- `(int) -5.3` равно -5;
- `(double) 5` равно 5.0;
- `(long long) 1e6` равно 1 000 000LL.

Например, чтобы разделить целочисленные константы 7 на 2 как вещественные, можно привести одну из них к вещественному типу:

- `(double)7 / 2` равно 3.5.
- `7 / (double)2` равно 3.5.

Если константы можно явно задать как вещественные, то с переменными так уже не получится. Поэтому данный приём актуален при работе с переменными.

Во время приведения вещественных чисел к целым, важно помнить несколько особенностей:

1. округление вещественных чисел производится в сторону нуля. Данная инициализация

```
int x = -5.9;
```

соответствует такой инициализации

```
int x = -5;
```

2. вещественные числа типа `float` не покрывают полностью диапазон типа `int`. Данная инициализация

```
float x = 16777217;
```

соответствует такой инициализации

```
float x = 16777216.0f;
```

так как вещественного значения $2^{24} + 1$ типа `float` не бывает.

3. вещественные числа типа `double` не покрывают полностью диапазон типа `long long`. Данная инициализация

```
double z = 9007199254740993;
```

соответствует такой инициализации

```
double z = 9007199254740992.0;
```

так как вещественного значения $2^{53} + 1$ типа `double` не бывает.

Пример 4. Найти сумму и произведение чисел 123456789 и 987654321.

Конечно, можно написать сразу оператор присваивания для соответствующих выражений. Но есть как минимум 2 причины, по которым эти числа лучше положить в переменные: не придется набирать одни и те же константы дважды и легко изменить код при изменении самих чисел.

Какой тип для переменных выбрать? Сами числа могут быть типа `int`. Сумма тоже умещается в тип `int`. А вот произведение не вмещается, поэтом для произведения необходимо использовать тип подходящего размера, например `long long`.

```
int a = 123456789, b = 987654321, sum;
long long mult;
sum = a + b;
mult = a * b;
```

Но данный код неверно вычислит произведение, так как сначала производятся вычисления правой части и только потом присваивание. Уже на этапе умножения происходит переполнение. Решением может быть один из 3 вариантов: объявить хотя бы одну из переменных `a` типа `long long`, сделать неявное преобразование типа или сделать явное преобразование типа. Первый вариант не требует пояснений:

```
long long a = 123456789, b = 987654321, sum, mult;
sum = a + b;
mult = a * b;
```

Второй вариант заключается в умножении на фиктивную единицу типа `long long`:

```
int a = 123456789, b = 987654321, sum;
long long mult;
sum = a + b;
mult = a * 1LL * b;
```

Результат вычислений преобразуется так:

$$(123456789 * 1LL) * 987654321 \rightarrow 123456789LL * 987654321$$

что даст правильный ответ. Но наиболее правильным с точки зрения читаемости кода будет третий вариант:

```
int a = 123456789, b = 987654321, sum;
long long mult;
sum = a + b;
mult = (long long)a * b;
```

Стоит отметить, что вариант с таким приведением `mult = (long long)(a * b);` не является правильным, так как здесь сначала происходит переполнение при умножении двух чисел типа `int` и уже полученный неправильный результат приводится к типу `long long` (фактически код ничем не отличается от варианта с неявным приведением `mult = a * b;`).

3.9 Приоритет выполнения операторов

Если с порядком выполнения бинарных арифметических операторов всё достаточно очевидно, то для остальных операторов стоит сделать уточнение. Таблица приоритетов всех операторов, о которых была речь выше, выглядит следующим образом:

Приоритет	Оператор	Примеры
1	<code>++ --</code>	<code>a++</code> или <code>a--</code>
2	<code>++ -- + -</code>	<code>++a</code> или <code>-a</code>
2	<code>(type)</code>	<code>(int)a</code>
3	<code>* / %</code>	<code>a * b</code> или <code>a % b</code>
4	<code>= += -= *= /= %=</code>	<code>a = b</code> или <code>a /= b</code>

У бинарных операторов порядок вычисления операндов производится слева направо. То есть для выражения $(a + b) * (c + d)$ сначала вычисляется $a + b$, потом вычисляется $c + d$, и в конце перемножаются результаты в скобках.

Разберем следующий пример:

```
x *= -a + b % 10 * (-3) * (-a);
```

Выпишем порядок выполнения операторов:

1. унарный минус `-a`;
2. остаток от деления `b % 10`;
3. умножение `(b % 10) * (-3)`;
4. унарный минус `-a`;
5. умножение `(b % 10) * (-3)) * (-a)`;
6. сложение `(-a) + ((b % 10) * (-3)) * (-a)`;
7. умножение с присваиванием
 $x = x * ((-a) + ((b \% 10) * (-3)) * (-a)).$

3.10 Комментарии

В тексте программы есть возможность оформить комментарий, то есть выделить некоторый участок, который будет игнорироваться компилятором. Зачем он нужен? Как правило для двух целей: либо сделать некоторые пометки для пояснений читающему код, либо временно «выключить» из работы некоторый код. Кстати, считается очень хорошим тоном комментировать исходный код программы, если он производит не очень очевидные действия. Эти пометки оформляются как комментарии. Существует два вида комментариев: однострочные в стиле C++ (начинаются с `//`, заканчиваются в конце строки) и многострочные в стиле C (начинаются с `/*`, заканчиваются `*/`). К слову, последние удобно набирать на цифровой клавиатуре, где символы умножения и деления расположены рядом.

```
int x; //this text is simple-line comment
x = 5;
/* this text is
multiline comment */
```

3.11 Минимальная программа

Минимальная программа на языке C, которая ничего не делает, выглядит следующим образом:

```
1 int main()
2 {
3
4     return 0;
5 }
```

Данный код будет присутствовать во всех программах. Здесь:

- `int main()` — основная функция, которая обозначает начало программы;
- код между фигурными скобками будет выполняться сверху вниз;
- `return 0;` — команда, завершающая выполнение нашей программы.

3.12 Вывод текста

Простейшая программа — программа «Hello world». Она должна вывести текст «Hello world» в терминал. Для вывода текста в терминал проще всего использовать функцию `puts()`. Для этого необходимо подключить специальный заголовочный файл `stdio.h` (англ. standard input output, header) до начала программы:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      puts("Hello world");
6      return 0;
7  }
```

Обратите внимание на двойные кавычки, в которые заключается текст, на точку с запятой, на фигурные и круглые скобки — здесь чаще всего можно ошибиться. И не путайте ноль 0 в команде `return 0`; и букву o в слове `stdio`.

3.13 GCC: компиляция

Во-первых, создадим в редакторе `nano` файл:

```
$ nano prog.c
```

Наберём исходный код программ, сохраним его (`ctrl + o`, `enter`) и выйдем обратно в терминал (`ctrl + x`).

Во-вторых, скомпилируем данную программу с помощью компилятора `gcc` (GNU Compiler Collection — набор компиляторов для различных языков программирования):

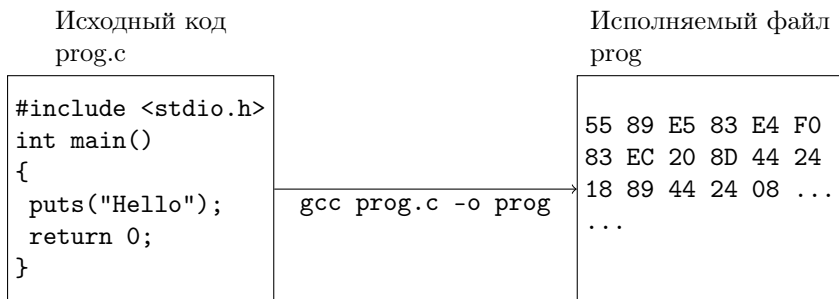
```
$ gcc prog.c -o prog -Wall
```

Мы указали параметры:

-o `prog` — создать после компиляции исполняемый файл `prog`;

-Wall — показывать все предупреждения.

Если ничего не произошло, то программа успешно скомпилировалась. Теперь у Вас есть два файла: исходный код `prog.c` и исполняемый код `prog`. Именно второй файл и есть программа, которую можно запустить на компьютере.



Если появилось сообщение

Command 'gcc' not found

то достаточно установить компилятор командой

`sudo apt install build-essential`

В случае других ошибок, нужно их исправить. Как правило, это опечатки: проверьте код «буква в букву» (точки с запятой и прочие «мелкие» детали вовсе не мелкие). А чтобы понять, какие именно были допущены ошибки, прочтите параграф «GCC: работа над ошибками» в данной теме.

В-третьих, запустим программу:

`$./prog`

Здесь `./` означает, что программа находится в текущей директории. После компиляции и запуска этой программы на экране появится приветственное слово:

Hello world
\$

Если бы программа находилась, например, во вложенной директории `home01`, то компиляцию и запуск можно было бы осуществить так

`$ gcc home01/prog.c -o home01/prog -Wall`
`$ home01/prog`

3.14 Форматный вывод

Как печатать значения переменных? Делегируем это функции форматного вывода `printf`. Укажем имя переменной и соответствующий ей формат:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n = 2018;
6     printf("%d", n);
7     return 0;
8 }
```

Смотрим на вывод:

```
$ nano prog.c
$ gcc prog.c -o prog -Wall
$ ./prog
2018$
```

Добавим перенос строки, чтобы вывод программы не «приклеивался» к приветствию командной строки. Перенос строки — обратный слеш и символ `n` (от англ. *newline*) — дань традиций печатной машинке.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n = 2018;
6     printf("%d\n", n);
7     return 0;
8 }
```

С помощью такого символа появляется переход на новую строку (как будто программа нажимает `enter`). Это необходимо для двух целей: во-первых, для того чтобы вывод нашей программы не «приклеивался» к приветствию командной строки; во-вторых, чтобы гарантировать вывод на экран.

Напишем программу для вычисления суммы двух целых чисел 3 и 4, которые изначально находятся в переменных `a` и `b`, и выведем результат вычислений на экран.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 3, b = 4, sum;
```

```
6      sum = a + b;
7      printf("%d\n", sum);
8      return 0;
9  }
```

Скомпилируем и запустим программу:

```
$ gcc prog.c -o prog -Wall
$ ./prog
7
$
```

Для вывода различных типов используется следующий интерфейс:

```
printf("%format", variable).
```

Часть текста, которая заключена в кавычки, называется форматной строкой. Эта строка определяет, в каком виде вводятся или выводятся данные. Например, форматная подстановка `%d` означает, что вводимое число трактуется как число типа `int`. Подстановка `%f` означает, что вводимое число трактуется как вещественное с десятичной запятой.

формат	размер	тип	пример
<code>%hhu</code>	1	<code>unsigned char</code>	17
<code>%hh</code>	1	<code>char</code>	17
<code>%hu</code>	2	<code>unsigned short</code>	2017
<code>%h</code>	2	<code>short</code>	-2017
<code>%u</code>	4	<code>insigned int</code>	20172018
<code>%d</code>	4	<code>int</code>	-20172018
<code>%x</code>	4	<code>insigned int</code>	CAFE2018 (16-ричная cc)
<code>%llu</code>	8	<code>unsigned long long</code>	12345678987654321
<code>%ll</code>	8	<code>long long</code>	-12345678987654321
<code>%f</code>	4	<code>float</code>	1.23456f
<code>%e</code>	4	<code>float</code>	123456e+05f
<code>%g</code>	4	<code>float</code>	123456e+05f
<code>%lf</code>	8	<code>double</code>	1.23456
<code>%e</code>	8	<code>double</code>	123456e+05
<code>%g</code>	8	<code>double</code>	123456e+05

В простых программах для вывода фиксированного текста лучше использовать `puts`, а для вывода чисел `printf`.

3.15 Форматный ввод

Раз мы пишем простейший калькулятор, то наверняка хотим дать пользователю возможность вводить эти числа и не компилировать программу для каждой новой пары чисел. Напишем программу, которая при запуске будет ожидать ввода двух целых чисел, а после ввода складывать их и выводить результат на экран. Для ввода воспользуемся функцией:

```
scanf("%format", &variable).
```

Пример 5. Даны 2 целых числа от -10^9 до 10^9 . Найти их сумму.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b, sum;
6      scanf("%d", &a);
7      scanf("%d", &b);
8      sum = a + b;
9      printf("%d\n", sum);
10     return 0;
11 }
```

Можно сделать ещё лучше — объединить две операции ввода в одну.

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

Обратите внимание, что перед именем переменной в функции `scanf` необходимо добавлять символ `&` (амперсанд). Он служит указанием на то, что переменная будет изменена (в отличие от вывода, где переменная не изменяется). Если его не написать:

```
int a;
scanf("%d", a);
```

то получите предупреждение:

```
$ warning: format '%d' expects argument of  
type 'int *', but argument 2 has type 'int'
```

Обсудим более тонкие настройки ввода-вывода. Во-первых, по умолчанию спецификатор `%lf` позволяет выводить вещественные числа с точностью 6 знаков после запятой. Изменить точность вывода можно спецификатором точкой:

```
double x = 0.12345678;  
printf("%lf\n", x);  
printf("%.9lf\n", x);
```

Вывод (число выводится с 9 знаками после запятой):

```
0.123457  
0.123456780
```

Во-вторых, ширина вывода. Её можно изменять так:

```
printf("%d%d%d\n", 1, 10, 100);  
printf("%4d%4d%4d\n", 1, 10, 100);
```

Вывод (на каждое число по 4 символа):

```
1101000  
1 10 100
```

В-третьих, при настройке ширины можно указывать символ заполнения (символ по умолчанию — пробел). Например, можно сделать форматный ввод-вывод для момента времени:

```
int h, m, s;  
scanf("%d:%d:%d", &h, &m, &s);  
printf("%02d:%02d:%02d", h, m, s);
```

Вывод (дополнение нулями при необходимости):

```
12:4:6  
12:04:06
```

В-четвертых, пробел в форматном вводе позволяет вводить числа через любой пробельный символ, включая пробел, табуляцию и даже перенос строки.

3.16 GCC: работа с ошибками

В качестве примера с ошибкой выступит код в файле `prog.c`:

```
1 int main()
2 {
3     x = 5;
4     printf("%d", x);
5     return 0;
6 }
```

Компиляция данной программы:

```
$ gcc prog.c -o prog -Wall
```

Вместо правильного «ничего» на экране появилось сообщение с ошибкой:

```
prog.c: In function 'main':
prog.c:3:2: error: 'x' undeclared
(first use in this function)
x = 5;
^
```

Тут проявляется очень важный навык программиста — умение находить и исправлять ошибки. Давайте разберёмся, как искать ошибки. Ищем первую запись, в которой имеется слово **error**:

```
prog.c:3:2: error
```

Эта запись означает, что в файле `prog.c` на третьей строке рядом со вторым символом что-то не так. Далее идет пояснение: переменная `'x'` не объявлена (`'x' undeclared`). Объявим её! Для этого снова откроем редактор и допишем тип для корректного объявления:

```
1 int main()
2 {
3     int x = 5;
4     printf("%d", x);
5     return 0;
6 }
```

Сохраним и снова компилируем.

```
prog.c: In function 'main':
prog.c:4:5: warning: implicit declaration of
function 'printf' [-Wimplicit-function-declaration]
```

```
printf("%d\n", x);  
~
```

Теперь ошибок нет, но есть предупреждение (англ. **warning**). В принципе, программа скомпилировалась, и её можно даже запустить. Но компилятор рекомендует исправить код. Данным предупреждением просит уточнить, где именно находится функция **printf** (неявное объявление **printf**). Допишем включение заголовочного файла с описанием.

```
1  #include <stdio.h>  
2  
3  int main()  
4  {  
5      int x = 5;  
6      printf("%d", x);  
7      return 0;  
8  }
```

Сохраним и снова скомпилируем. Теперь нет ни ошибок, ни предупреждений. Поэтому можем запускать программу:

```
$ ./prog
```

Стоит отметить, что ошибка не обязательно будет находиться в указанной строке, она может быть строкой выше или ниже. Кстати, большинство редакторов поддерживает горячие комбинации для перехода на определенную строку. В редакторе **nano** для этого необходимо нажать **ctrl + shift + -** и далее набрать номер нужной строки.

3.17 Отступы

В коде настоятельно рекомендуется придерживаться определенных правил оформления отступов, так как в программах с большим кодом это значительно улучшает читаемость кода. Отступы следует вводить нажатием клавиши табуляция, а не многократным нажатием пробела.

Пустые строки игнорируются, их тоже можно использовать для улучшения читаемости, разделяя ими логические блоки.

Хороший стиль оформления кода. Внутренняя часть **main** должна быть выровнена по одинаковому отступу. В качестве отступа рекомендуется использовать один из 3 вариантов: табуляция, 4 пробела (клавишей табуляция) или 8 пробелов (клавишей табуляция).

Большинство редакторов поддерживает умные переносы. То есть после нажатия клавиши **enter** курсор переходит на следующую строку, выравнивая курсор на уровень отступа предыдущей строки. Например, для включения такого режима в **nano** его достаточно запустить с флагом **-i** (англ. indent). А чтобы сделать автозамену табуляции на 4 пробела — с флагом **-T4**.

```
$ nano -iT4 prog.c
```

Либо можно произвести настройку файла **.nanorc**, которая была описана в теме «Техническое введение. Терминал и консольные редакторы».

3.18 Примеры

Пример 6. Даны 3 целых числа от -10^6 до 10^6 . Вывести эти числа, сдвинув их циклически влево.

Ввод	10 11 12
Вывод	11 12 10

Поменяем порядок вывода чисел.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a, b, c;
6     scanf("%d %d %d", &a, &b, &c);
7     printf("%d %d %d", b, c, a);
8     return 0;
9 }
```

Пример 7. Даны 3 вещественных положительных числа a, b, c от 0 до 100 таких, что существует треугольник со сторонами a, b, c . Найти квадрат площади данного треугольника и вывести ответ с точностью в два знака после запятой.

Ввод	1.0 1.1 1.2	3.0 4.0 5.0
Вывод	0.27	36.00

Вычислим квадрат площади через формулу Герона.

```

1  #include <stdio.h>
2  int main()
3  {
4      double a, b, c, p, S;
5      scanf("%lf %lf %lf", &a, &b, &c);
6      p = (a + b + c) / 2.0;
7      S = p * (p - a) * (p - b) * (p - c);
8      printf("%.2lf\n", S);
9      return 0;
10 }
```

Пример 8. Дано положительное вещественное число от 0 до 10^9 . Найти дробную часть данного числа с точностью в три знака после запятой.

Ввод	12.34	0.34567	3.0
Вывод	0.340	0.346	3.000

Вычислим дробную часть как разницу между числом и целой частью числа. А целую часть числа вычислим приведением вещественного типа к целому.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      double x, fraction;
6      scanf("%lf", &x);
7      fraction = x - (int)x;
8      printf("%.3lf\n", fraction);
9      return 0;
10 }
```

Пример 9. Дано целое число n от 1 до 10^{18} . Необходимо вывести разряд сотен числа n .

Ввод	1234567	2017	1234567898765
Вывод	5	0	7

Поделим число целочисленно на 100 — это операция отбросит две последних цифры числа (например, из числа 1234567 получится 12345). У результата вычислим остаток при делении на 10 — это операция оставит последнюю цифру числа (например, из числа 12345 останется 5).

```

1  #include <stdio.h>
2  int main()
3  {
4      long long n;
5      int ans;
6      scanf("%lld", &n);
7      ans = n / 100 % 10;
8      printf("%d\n", ans);
9      return 0;
10 }
```

Пример 10. Дано положительное вещественное число. Найти последнюю цифру целой части числа.

Ввод	123.4567
Вывод	3

Отбросим дробную часть приведением к целому числу (например, от числа 123.4567 останется 123). У результата вычислим остаток при делении на 10 — это операция оставит последнюю цифру числа (от числа 123 останется 3).

```

1  #include <stdio.h>
2
3  int main()
4  {
5      double x;
6      int ans;
7      scanf("%lf", &x);
8      ans = (int)x % 10;
9      printf("%d\n", ans);
10     return 0;
11 }
```

Пример 11. Дано положительное вещественное число. Найти первую цифру дробной части числа.

Ввод	123.4567
Вывод	4

Умножим число на 10 и выполним действия из предыдущей задачи.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      double x;
6      int ans;
7      scanf("%lf", &x);
8      ans = (int)(x * 10) % 10;
9      printf("%d\n", ans);
10     return 0;
11 }
```

3.19 Задания для самостоятельной работы

1. Определить какие объявления переменной корректны (ответ обосновать). Если объявление корректно, то указать тип приведения (из какого типа в какой):
 - а) `int a = 2.0;`;
 - б) `float 1x = 2;`;
 - в) `unsigned long long m = 1e18;`
 - г) `long long msu.kz = 1;`;
 - д) `short z = 0xFULL;`;
 - е) `double radius_1 = 1;`;
2. В каком случае произведение чисел `int a = 1000000;` и `int b = 2000000;` будет вычислено верно:
 - а) `1LL * a * b;`;
 - б) `a * 1LL * b;`;
 - в) `a * b * 1LL;`;
 - г) `(int)(a * 1LL) * b;`;
 - д) `(long long)(a * b) * 1;`;
 - е) `(long long)a * b;`;

```
int x = 2, y = 3;
x = y - 2;
y = 10 + 5 * x;
x++;
y -= 2;
x *= x;
x -= -x;
```

4. Дано двузначное целое число. Найти сумму цифр.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n, ans;
6     scanf("%d", &n);
7     ans =
8     printf("%d\n", ans);
9     return 0;
10 }
```

5. Дано два целых положительных числа a и b . Вычислить

$$\frac{a+b}{2} + \frac{2}{1/a + 1/b}$$

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b;
6      double ans;
7      scanf("%d %d", &a, &b);
8      ans =
9      printf("%lf\n", ans);
10     return 0;
11 }
```


3.20 Практикум на ЭВМ

1. Даны 3 целых числа от 1 до 10^9 . Вывести их в обратном порядке.

Ввод	1 3 5	1 1 2
Вывод	5 3 1	2 1 1

2. Даны 2 целых числа от 1 до 10^9 . Вывести их сумму.

Ввод	10 2	123456789 1
Вывод	12	123456790

3. Даны 2 целых числа от 1 до 10^9 . Вывести их произведение.

Ввод	10 2	300000 500000
Вывод	20	150000000000

4. Даны 2 целых числа от 1 до 1000 — числитель и знаменатель дроби. Вычислить соответствующее значение десятичной дроби и вывести с точностью 2 знака после запятой.

Ввод	2 3	3 2
Вывод	0.67	1.50

5. Дано целое число от 1 до 10^9 . Вывести в две строки: разряд единиц и разряд десятков.

Ввод	2019	7
Вывод	9 1	7 0

6. Дано целое число от 100 до 999. Найти сумму цифр числа.

Ввод	495	101
Вывод	18	2

7. Даны 3 целых числа a, b, c от 1 до 1000. Найти их среднее арифметическое $\frac{a+b+c}{3}$ и среднее гармоническое $\frac{3}{\frac{1}{a}+\frac{1}{b}+\frac{1}{c}}$ и вывести с точностью 2 знака после запятой.

Ввод	2 3 6	1 2 2
Вывод	3.67 3.00	1.67 1.50

8. Дано вещественное положительное число от 0 до 1000 с 3 знаками после запятой. Найти вторую цифру после запятой.

Ввод	123.456	0.987
Вывод	5	8

9. Даны 3 целых числа a , b , c от 1 до 1000. Вычислить координаты вершины параболы $f(x) = ax^2 + bx + c$.

Ввод	3 2 1	4 8 3
Вывод	-0.33 0.67	-1.00 -1.00

10. Даны 3 вещественных положительных числа от 0 до 1000 с 1 знаком после запятой — стороны треугольника. Гарантируется, что такой треугольник существует. Найти отношение радиусов описанной и вписанной окружностей. Ответ вывести с точностью 2 знака после запятой.

Ввод	3.0 4.0 5.0	2.0 3.0 2.5
Вывод	2.50	2.29

4 Условный оператор.

Математические функции

В рамках данной темы будут рассмотрены операторы сравнения и логические операторы, которые являются основой для условного оператора. Разобраны примеры работы с функциями из заголовочного файла «math.h».

4.1 Операторы сравнения

Оператор сравнения является бинарным оператором с булевым значением. Это означает, что его можно применять к двум операндам A и B , а результатом применения оператора является «истина» или «ложь». К бинарным операторам относятся:

1. $A > B$ — A больше B ;
2. $A < B$ — A меньше B ;
3. $A >= B$ — A не меньше B ;
4. $A <= B$ — A не больше B ;
5. $A != B$ — A не равно B ;
6. $A == B$ — A равно B .

Примеры с константами:

1. $2 > 1$ — истина;
2. $2.0 < 1.0$ — ложь;
3. $2LL >= 2LL$ — истина;
4. $2.f <= 2.f$ — истина;
5. $2U != 3U$ — истина;
6. $2.0 == 3.0$ — ложь.

Если операнды имеют разные типы, то производится неявное приведение к общему типу более высокого ранга, как и в арифметических операторах. Примеры с константами:

1. $1 == 1.0$ — истина (операнд слева приводится от типа `int 1` → `double 1.0`);

2. `1LL > 2` — ложь (операнд справа приводится от типа `int 2` → `long long 2LL`).

Оператор «равно» имеет звание наиболее популярной ошибки из-за схожести с оператором присваивания. Нужно помнить, что записывается он именно двумя символами `==` (сравниваем равноправные аргументы слева и справа), в отличие от одинарного равенства в операторе присваивания `=` (пересылаем значение справа в переменную слева).

Примеры с переменными

```
double x = 2.5, y = 2.0;
```

Сравнение переменных

1. `x > y` — истина;
2. `x < y` — ложь;
3. `x == y` — ложь;
4. `x != y` — истина;
5. `(int)x == y` — истина (обратите внимание, что происходит двойное преобразование типа `y` левого операнда: явное `double 2.5` → `int 2` и неявное `int 2` → `double 2.0`);
6. `x == (int)y` — ложь.

Важным отличием языка `C` является то что обычные числа тоже могут трактоваться как логические значения: нулевые значения — ложь, любые ненулевые — истина. Например:

1. `2` — истина;
2. `0` — ложь.

Пользоваться последним свойством нужно крайне аккуратно.

В задачах часто возникает необходимость выписать условие, которое верно для имеющегося числа или чисел. Рассмотрим некоторые выражения на примере переменных:

```
int n;  
double a, b, c;
```

Далее эти переменные заполняются некоторыми значениями. Рассмотрим примеры операторов сравнения, которые дают значение истина при соответствующих условиях:

1. $n > 0$ — число n положительно;
2. $n \leq 0$ — число n не положительно;
3. $n \% 100 == 0$ — число n делится на 100 (остаток при делении на 100 равен нулю);
4. $n \% 2 == 0$ — число n является четным;
5. $n \% 2 == 1$ — число n является положительным нечетным числом (обратите внимание, что для отрицательных нечетных чисел $n \% 2$ даст результат -1);
6. $n \% 2 != 0$ — число n является нечетным;
7. $n \% 10 == 5$ — число заканчивается на 5;
8. $n / 10 \% 10 == 5$ — разряд десятков равен 5;
9. $a == (\text{int})a$ — вещественное число от -10^9 до 10^9 является целым.

Хороший стиль оформления кода. Каждый оператор сравнения рекомендуется отделять от операндов пробелами слева и справа.

Например:

```
a > b + 3
a * a + b * b <= c * c
```

— хорошо;

```
a>b+3
a*a + b*b<=c*c
```

— плохо.

4.2 Логические операторы

Почти в любом современном языке программирования имеются базовые операторы математической логики:

1. $A \ \&\& \ B$ — оператор AND (логическое И, логическое умножение, конъюнкция, в теории множеств — пересечение) принимает следующие значения в зависимости от операндов:

$\&\&$	A — ложь	A — истина
B — ложь	ложь	ложь
B — истина	ложь	истина

2. $A \parallel B$ — оператор OR (логическое ИЛИ, логическое сложение, дизъюнкция, в теории множеств — объединение) принимает следующие значения в зависимости от операндов:

\parallel	A — ложь	A — истина
B — ложь	ложь	истина
B — истина	истина	истина

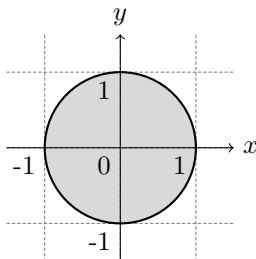
3. $!A$ — оператор NOT (логическое НЕ, логическое отрицания, в теории множеств — дополнение).

Рассмотрим примеры комбинаций логических операторов и операторов сравнения. Даны переменные:

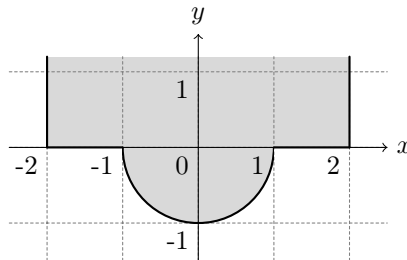
```
int n, m, n2, m2;
double x, y, z;
```

Далее эти переменные заполняются некоторыми значениями. Данные выражения дают значение «истина» при условиях:

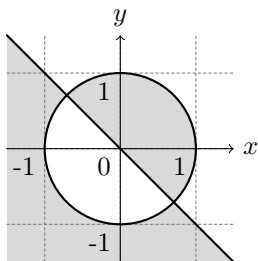
1. $x \geq 0 \ \&\& \ x < 1$ — вещественное число x принадлежит $[0; 1)$;
2. $!(x \geq 0 \ \&\& \ x < 1)$ — вещественное число x не принадлежит $[0; 1)$;
3. $x > 0 \ \&\& \ y > 0$ — точка с координатами $(x; y)$ лежит в первой четверти;
4. $!(x > 0 \ \&\& \ y > 0)$ — точка с координатами $(x; y)$ не лежит в первой четверти;
5. $x * x + y * y \leq 1$ — точка с координатами $(x; y)$ лежит в круге с центром в точке $(0; 0)$ и радиусом 1;



6. $n \% 2 == m \% 2$ — клетка шахматной доски на позиции $(n; m)$ окрашена в черный цвет (считаем, что $(1; 1)$ — черная);
7. $(n + m) \% 2 == 0$ — клетка шахматной доски на позиции $(n; m)$ окрашена в черный цвет (считаем, что $(1; 1)$ — черная);
8. $(n + m + n_1 + m_1) \% 2 == 0$ — две клетки шахматной доски $(n; m)$ и $(n_1; m_1)$ покрашены в один цвет;
9. $x > 0 \ \&\& \ y > 0 \ \&\& \ z > 0$ — все три числа x, y, z положительны;
10. $x > 0 \ || \ y > 0 \ || \ z > 0$ — хотя бы одно из трех чисел x, y, z положительно;
11. $n \% 2 == 0 \ \&\& \ n / 10 \% 2 == 0$ — обе цифры двузначного числа n четные;
12. $n \% 2 == 0 \ || \ n / 10 \% 2 == 0$ — хотя бы одна цифра двузначного числа n четная.
13. $n \% 400 == 0 \ || \ n \% 100 != 0 \ \&\& \ n \% 4 == 0$ — год n является високосным;
14. $x * x + y * y <= 1 \ \&\& \ y >= 0$ — точка с координатами $(x; y)$ лежит в полукруге с центром в точке $(0; 0)$ и радиусом 1;
15. $x + y == 1 \ \&\& \ x >= 0 \ \&\& \ y >= 0$ — точка с координатами $(x; y)$ принадлежит отрезку, соединяющему точки $(0; 1)$ и $(1; 0)$;
16. $x * x + y * y <= 1 \ || \ y > 0 \ \&\& \ x >= -2 \ \&\& \ x <= 2$ — точка с координатами $(x; y)$ принадлежит изображенной ниже области;



17. $(x * x + y * y <= 1) == (x + y >= 0)$ — точка с координатами $(x; y)$ принадлежит изображенной ниже области (либо оба неравенства верны, либо оба неравенства неверны).



Стоит обратить внимание на рекомендацию ставить скобки явно при комбинации операторов, хотя приоритет операторов `||` и `&&` определён. Например, код:

```
a < 0 || a > 5 && a < 10
```

вызывает следующее предупреждение:

```
$ warning: suggest parentheses
      around '&&' within '||'
      [-Wparentheses]
```

В этом случае следует расставить скобки операторов явно:

```
a < 0 || (a > 5 && a < 10)
```

4.3 Приоритеты операторов

Логические операторы `||` и `&&` имеют более низкий приоритет, чем все операторы сравнения. Это означает, что можно писать следующее выражение без скобок:

```
x > 0 && x < 5
```

Однако оператор отрицания `!` имеет более высокий, то есть необходимо использовать скобки:

```
!(x < 4)
```

Таблица приоритетов операторов:

Приоритет	Оператор	Примеры
1	<code>++ --</code>	<code>b < a++</code> <code>b + a--</code>
2	<code>++ -- + -</code>	<code>b == ++a</code> <code>-a != b</code>
2	<code>(type)</code>	<code>(int)a + b</code> <code>(int)(a + b)</code>
3	<code>* / %</code>	<code>a * b > 0</code> <code>a % b == 0</code>
4	<code>< > <= >=</code>	<code>a > b</code> <code>a <= b</code>
5	<code>== !=</code>	<code>a == b</code> <code>a != b</code>
6	<code>&&</code>	<code>a > 0 && b < 0</code> <code>a % 2 == 0 && b % 2 == 0</code>
7	<code> </code>	<code>a > 0 b < 0</code> <code>a % 2 == 0 b % 2 == 0</code>
8	<code>= += -= *= /= %=</code>	<code>a = b > 0</code> <code>a += b == 0</code>

Интересный факт: для логических операторов применяется оптимизированное вычисление слева направо. То есть

- если логическое выражение `a` ложно, то `a && b` принимает значение «ложь», без вычислений значения `b`;
- если логическое выражение `a` истинно, то `a || b` принимает значение «истина», без вычислений значения `b`.

Например, такой код не приведёт к делению на ноль:

```
b > 0 && a / b == 2
```

Если `b == 0`, то первый операнд оператора `И` имеет значение «ложь». То есть второй операнд не влияет на результат и вычисления `a / b == 2` не выполняются.

Разберем следующий пример:

```
x % y > 5 || !(x == y) && x != 11
```

Выпишем порядок выполнения операторов:

1. деление: $x \% y$;
2. сравнение больше: $x \% y > 5$;
3. сравнение равно: $x == y$;
4. логическое отрицания: $!(x == y)$;
5. сравнение не равно: $x != 11$;
6. логическое И: $!(x == y) \ \&\& \ x != 11$;
7. логическое ИЛИ: $x \% y > 5 \ || \ !(x == y) \ \&\& \ x != 11$;

4.4 Тождества де Моргана

Отрицание некоторых условий удобно упрощать, если пользоваться правилом де Моргана:

$$\begin{aligned}!(A \ \&\& \ B) &\text{ эквивалентно } !A \ || \ !B \\!(A \ || \ B) &\text{ эквивалентно } !A \ \&\& \ !B\end{aligned}$$

Например, условие, что x лежит во множестве $(0; 1]$ можно записать как

$$0 < x \ \&\& \ x \leq 1$$

Отрицание к нему — x не лежит во множестве $(0; 1]$ — можно выразить тремя эквивалентными способами:

$$\begin{aligned}!(0 < x \ \&\& \ x \leq 1) \\!(x > 0) \ || \ !(x \leq 1) \\x \leq 0 \ || \ x > 1\end{aligned}$$

4.5 Условный оператор

Условный оператор, который часто упоминается как оператор ветвления, позволяет выделить часть кода (action), которая будет выполняться только в случае, когда условие (condition) принимает значение «истина».

```
if (condition)
    action;
```

Оператор выполняется только в случае, если логическое выражение принимает истинное значение. Следующий код позволяет изменить число на его модуль:

```
double x;  
scanf("%d", &x);  
if (x < 0)  
    x = -x;  
printf("%d", x);
```

Если $x < 0$, то условие в условном операторе «истинно» и выполняется код $x = -x$. В противном случае, действие в условном операторе не выполняется и значение переменной не меняется.

Другой вид условного оператора дополнительно предписывает выполнение альтернативных действий при невыполнении условий.

```
if (condition)  
    action1;  
else  
    action2;
```

Следующий код выводит слово «exist», если треугольник со сторонами a , b , c существует и «not exist» в противном случае:

```
int a, b, c;  
a = ...;  
b = ...;  
c = ...;  
if (a + b > c && b + c > a && c + a > b)  
    puts("exist");  
else  
    puts("not exist");
```

Если $a = 5$, $b = 6$ и $c = 7$, то выполнится код: `puts("exist")`. А если $a = 5$, $b = 6$ и $c = 12$, то выполнится альтернатива: `puts("not exist")`;

В случае необходимости разбора нескольких альтернатив можно использовать вложенные условные оператор (так называемый вариант `if .. else if ... else ...`):

```
if (condition1)  
    action1;  
else if (condition2)  
    action2;  
else
```

```
action3;
```

Например, следующий код позволяет определить количество корней y квадратного уравнения $ax^2 + bx + c = 0$:

```
double a, b, c, D, roots;
a = ...;
b = ...;
c = ...;
D = b * b - 4 * a * c;
if (D < 0)
    roots = 0;
else if (D == 0)
    roots = 1;
else
    roots = 2;
```

Если $a = 1$, $b = 2$, $c = 3$, то мы заходим в «ветку» `roots = 0`. Если $a = 1$, $b = 2$, $c = 1$, то заходим в «ветку» `roots = 1`. Если $a = 2$, $b = 3$, $c = 1$ — то в «ветку» `roots = 2`.

Рассмотрим код полного решения одной из задач.

Пример 1. Дано целое число от 1 до 9. Проверить, является ли оно простым или составным (число 1 не является ни тем, ни другим).

Ввод	1	2	4
Вывод	nothing	prime	composite

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n;
6     scanf("%d", &n)
7     if (n == 2 || n == 3 || n == 5 || n == 7)
8         puts("prime");
9     else if (n == 4 || n == 6 || n == 8 || n == 9)
10        puts("composite");
11    else
12        puts("nothing");
13    return 0;
14 }
```

4.6 Составной оператор

Для выполнения нескольких действий в одной ветке условного оператора их следует заключить в фигурные скобки `{ ... }`. Такие скобки образуют составной оператор.

Пример 2. Даны 3 вещественных числа. Проверить, существует ли треугольник с данными сторонами. Если такой треугольник существует, то вывести его полупериметр.

Ввод	1 2 4	4 5 6
Вывод	not exist	7.50

```

1  #include <stdio.h>
2
3  int main()
4  {
5      double a, b, c, p;
6      scanf("%lf %lf %ld", &a, &b, &c);
7      if (a + b > c && b + c > a && c + a > b)
8      {
9          p = (a + b + c) / 2.0;
10         printf("%.2lf", p);
11     }
12     else
13         puts("not exist");
14     return 0;
15 }
```

Хороший стиль оформления кода. Обычно используется один из двух вариантов оформления открывающейся скобки составного оператора: на строке с условным оператором или на новой строке. Следует придерживаться единого стиля на протяжении всей программы.

Например, второй вариант оформления кода программы:

```

1  #include <stdio.h>
2
3  int main() {
4      double a, b, c, p;
```

```
5      scanf("%lf %lf %ld", &a, &b, &c);
6      if (a + b > c && b + c > a && c + a > b) {
7          p = (a + b + c) / 2.0;
8          printf("%.2lf", p);
9      } else {
10         puts("not exist");
11     }
12     return 0;
13 }
```

4.7 Пустой оператор

Данным оператором новички могут воспользоваться непреднамеренно. Например так:

```
int x = 5;
if (x < 0);
    x = 0;
```

После выполнения данного кода, в переменной x будет значение 0, а не 5.

Обратите внимание на точку с запятой на второй строке. Она завершает выполнение действия условного оператора (это и есть пустой оператор), а присваивание $x = 5$; выполняется вне контекста условного оператора:

```
int x = 5;
if (x < 0) {
}
x = 0;
```

Данной небрежности можно избежать, если компилировать приведенный код (именно с такими отступами) с флагом `-Wall`:

```
$ gcc prog.c -o prog -Wall
```

В ответ получим предупреждение:

```
prog.c:5:4: warning: this 'if' clause
           does not guard...
           [-Wmisleading-indentation]
if (x < 0);
~
prog.c:6:8: note: ...this statement,
           but the latter is
```

```

misleadingly indented as
    if it were guarded by the 'if'
x = 0;
^

```

Хороший стиль оформления кода. После условия условного оператора точка с запятой не ставится, а действие пишется на новой строке с дополнительным отступом в виде табуляции.

Например:

```

if (x < 0)
    x = -1;

```

— хорошо;

```

if (x < 0) x = -1;

```

— плохо;

```

if (x < 0)
x = -1;

```

— плохо.

Хороший стиль оформления кода. Альтернатива условного оператора выравнивается по уровню соответствующего условного оператора.

Например:

```

if (x < 0)
    x = -1;
else
    x = 1;

```

— хорошо;

```

if (x < 0) x = -1; else x = 1;

```

— плохо;

```

if (x < 0) x = -1;
else x = 1;

```

— плохо.

4.8 Вложенный условный оператор

Если внутри одного условного оператора имеется другой условный оператор, то необходимо корректно расставить не только отступы, но и фигурные скобки. Например, такой код не позволяет определить вложенность условных операторов визуально:

```
int a = 5;
if (a > 0)
if (a < 4)
    puts("OK");
else
    puts("NO");
```

К какому условному оператору относится `else`? Если к первому, то ничего не будет выведено на экран; если ко второму, то будет выведено слово «NO». В языке C ветка `else` всегда относится к ближайшему оператору `if`. То есть правильный вариант оформления:

```
int a = 5;
if (a > 0)
{
    if (a < 4)
        puts("OK");
    else
        puts("NO");
}
```

О такой неоднозначности Вы точно не забудете, если будете компилировать код с флагом `-Wall`.

4.9 Функция модуля с целочисленным аргументом

Рассмотрим некоторые математические функции, представленные в языке C. Функция — это некоторый чёрный ящик, который на вход принимает аргументы и на выход выдает результат. Например, для вычисления модуля целого числа в языке C имеется функция:

```
int abs(int);
```

В нашем случае `int` перед именем представляет собой тип результата функции, `abs` — имя функции, а `int` в круглых скобках — тип аргумента. Данная функция хранится в стандартной библиотеке `stdlib.h`. Применим её в задаче про шахматные фигуры.

Пример 3. Даны 4 целых числа x_1, y_1, x_2, y_2 от 1 до 8. Проверить, можно ли из позиции $(x_1; y_1)$ перейти в позицию $(x_2; y_2)$ ходом коня.

Ввод	1 2 2 4	1 2 4 3
Вывод	Yes	No

Для этого вычислим модули смещения по каждой координате

$$dx = |x_1 - x_2|$$

$$dy = |y_1 - y_2|$$

и проверим условия перехода. Ходом коня можно сместиться на одну клетку по горизонтали $dx = 1$ и на две клетки по вертикали $dy = 2$ или наоборот.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int x1, y1, x2, y2, dx, dy;
7      scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
8      dx = abs(x1 - x2);
9      dy = abs(y1 - y2);
10     if ((dx == 1 && dy == 2) ||
11         (dx == 2 && dy == 1))
12         puts("Yes");
13     else
14         puts("No");
15     return 0;
16 }
```

Переход других шахматных фигур:

1. $dx == 0 \text{ || } dy == 0$ — можно перейти ходом ладьи;
2. $dx == dy$ — можно перейти ходом слона;
3. $dx * dx + dy * dy != 5$ — можно перейти ходом коня;
4. $dx == dy \text{ || } dx == 0 \text{ || } dy == 0$ — можно перейти ходом ферзя.

4.10 Функции с вещественным аргументом

Важное замечание: арифметические действия, которые были рассмотрены в предыдущих темах, выполняются значительно быстрее, чем математические функции, приведенные ниже. Например, операцию возведения в квадрат следует реализовывать как простое умножение `x * x`, а не средствами данной библиотеки `pow(x, 2)`.

Функция вычисления квадратного корня из вещественного числа имеет следующий прототип:

```
double sqrt(double x);
```

Если необходимо вычислить корень из 25.0, то функцию следует использовать следующим образом:

```
double r;  
r = sqrt(25.0);
```

Для вычисления модуля вещественного числа используется функция с прототипом:

```
double fabs(double x);
```

Для вычисления модуля числа -2.5 вызываем:

```
double r;  
r = fabs(-2.5);
```

Стоит отдельно подчеркнуть, что функция `double fabs(double)` из библиотеки `math.h` используется для нахождения модуля вещественных чисел, для вычисления модуля целых чисел есть функция `int abs(int)` из библиотеки `stdlib.h`.

В качестве аргумента также может выступать переменная, выражение или результат вычисления другой функции. Например, вычислим корень от модуля числа:

```
double r, a = -25.0;  
r = sqrt(fabs(a));
```

Список некоторых функций (результат — вещественное число `y`).

Прототип	Функция	Пример	Значение
<code>double fabs(double x)</code>	$y = x $	<code>y = fabs(-2.5);</code>	2.5
<code>double floor(double x)</code>	$y = \lfloor x \rfloor$	<code>y = floor(-2.2);</code>	-3
		<code>y = floor(2.2)</code>	2
<code>double ceil(double x)</code>	$y = \lceil x \rceil$	<code>y = ceil(-2.2);</code>	-2
		<code>y = ceil(2.2);</code>	2
<code>double sqrt(double x)</code>	$y = \sqrt{x}$	<code>y = sqrt(6.25);</code>	2.5
<code>double exp(double x)</code>	$y = e^x$	<code>y = exp(1.0);</code>	2.718..
<code>double sin(double x)</code>	$y = \sin x$	<code>y = sin(1.5);</code>	0.997..
<code>double cos(double x)</code>	$y = \cos x$	<code>y = cos(1.5);</code>	0.071..
<code>double tan(double x)</code>	$y = \operatorname{tg} x$	<code>y = tan(1.5);</code>	14.101..
<code>double asin(double x)</code>	$y = \arcsin x$	<code>y = asin(1.0);</code>	1.571..
<code>double acos(double x)</code>	$y = \arccos x$	<code>y = acos(1.0);</code>	0.000..
<code>double atan(double x)</code>	$y = \arctan x$	<code>y = atan(1.0);</code>	0.785..
<code>double log(double x)</code>	$y = \ln x$	<code>y = log(10.0);</code>	2.303..
<code>double log10(double x)</code>	$y = \lg x$	<code>y = log10(10.0);</code>	1.0

Несколько функций требует пояснений. Функция `floor` — целая часть числа (округляет вниз), а функция `ceil` — округляет до ближайшего целого числа. Стоит помнить, что возвращаемый тип у функций — вещественный, поэтому, например, такая запись некорректна:

```
int a = floor(13.6) % 10;
```

Если необходимо найти остаток, то результат следует привести к целочисленному типу:

```
int a = (int)floor(13.6) % 10;
```

Функции, которые реализуют прямую и обратную тригонометрию, работают не с градусами, а с радианами. Для удобства в этой же библиотеке описаны константы, часто встречающиеся при работе с тригонометрией и логарифмами:

- `M_PI` (соответствует $\pi = 3.1415\dots$);
- `M_PI_2` (соответствует $\frac{\pi}{2} = 1.5707\dots$);
- `M_E` (соответствует $e = 2.71828\dots$).

Таже имеются функции с несколькими аргументами. Например, функция

```
double atan2(double a, double b);
```

позволяет вычислить полярный угол, соответствующий радиус-вектору точки с координатами $(x; y)$. Ответ вычисляется по модулю 2π , в то время как функция `double atan(double a);` вычисляет арктангенс по модулю π . Например,

```
double a = atan(5.00 / -5.00);  
double a2 = atan2(5.00, -5.00);
```

Здесь $a = \frac{3}{4}\pi$, а $a_2 = \frac{7}{4}\pi$.

Подробное описание всех функций можно узнать в справке. Например, если необходимо получить информацию о функции `sqrt`, то достаточно набрать:

```
$ man sqrt
```

Выход из справки осуществляется нажатием клавиши `q`.

4.11 Неэффективное использование «math.h»

Часто используемая начинающими программистами функция для возведения в степень:

```
double pow(double a, double b);
```

вычисляет значение $y = a^b$. Стоит помнить, что данная функция не предназначена для возведения в целые степени b , хотя и позволяет это сделать. Вычисления через функцию `pow(a, b)` медленнее в сравнении с обычным умножением и может давать худший результат с точки зрения точности. Для возведения в целые степени лучше использовать технику из темы «Цикл `while`», а эту функцию — только для возведения в нецелые степени.

Второй типичной ошибкой является применение функции там, где она не нужна в принципе. Например, сравнение длины вектора с какой-либо величиной:

```
double x, y, r;  
if (sqrt(x * x + y * y) < r)  
    ...
```

Приведенный выше код работает дольше, чем следующий вариант:

```
double x, y, r;
if (x * x + y * y < r * r)
    ...
```

4.12 Линковка библиотеки «math.h»

Для работы с данной библиотекой необходимо подключить её заголовочный файл `math.h` в начале программы:

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

Хотя компилятор и компилирует программы без данного включения, но это приводит к предупреждениям вида:

```
$ warning: implicit declaration of function 'sqrt'
```

Особенностью библиотеки «math.h» является то, что её необходимо прилинковать (присоединить) добавлением флага `-lm` при компиляции:

```
$ gcc prog.c -o prog -Wall -lm
```

Без флага `-lm` можно получить вот такую ошибку:

```
In function `main':
prog.c:(.text+0x1d): undefined reference to 'sqrt'
collect2: error: ld returned 1 exit status
```

4.13 Примеры

Пример 4. Даны три целых числа a , b , c от -1000 до 1000. Определить, существует ли треугольник со сторонами a , b , c . Если такой треугольник существует, то вывести его площадь с 3 знаками после запятой, в противном случае — текст «Not exist».

Ввод	1.0 1.1 1.2	3.0 4.0 5.0	3.0 4.0 8.0
Вывод	0.27	6.00	Not exist

```
1 #include <stdio.h>
2 #include <math.h>
3
```

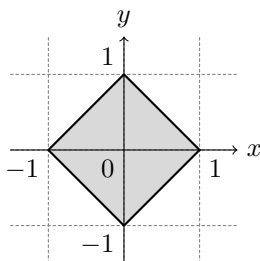
```

4  int main()
5  {
6      int a, b, c;
7      double p, S;
8      if (a + b > c && b + c > a && c + a > b)
9      {
10         p = (a + b + c) / 2.0;
11         S = sqrt(p * (p - a) * (p - b) * (p - c));
12         printf("%.3lf", S);
13     }
14     else
15         puts("Not exist.");
16     return 0;
17 }

```

Обратите внимание, что если записать $(a + b + c) / 2$ вместо $(a + b + c) / 2.0$, то оператор деления будет целочисленным. Тогда, например, для треугольника со сторонами 4, 5 и 6 полупериметр и, соответственно, площадь будут вычислены неверно. Также не стоит вычислять площадь до условного оператора: во-первых, лишние действия в случае с несуществующим треугольником, во-вторых, извлечение корня из отрицательного числа.

Пример 5. Даны координаты точки $(x; y)$. Определить положение (внутри, на границе или снаружи) данной точки относительно следующего квадрата:



Ввод	0.4 0.4	0.7 -0.3	-0.9 0.8
Вывод	Inside	Border	Outside

```

1  #include <stdio.h>
2  #include <math.h>

```

```

3
4 int main()
5 {
6     double x, y, d;
7     scanf("%lf %lf", &x, &y);
8     d = fabs(x) + fabs(y);
9     if (d < 1.0)
10         puts("Inside");
11     else if (d == 1.0)
12         puts("Border");
13     else
14         puts("Outside");
15     return 0;
16 }

```

Обратите внимание, что представленной программе функция для вычисления модуля числа вызывается только два раза — это эффективнее, чем вычислять модули в каждом условном операторе:

```

...
if (fabs(x) + fabs(y) < 1.0)
    puts("Inside");
if (fabs(x) + fabs(y) == 1.0)
    puts("On the border");
if (fabs(x) + fabs(y) > 1.0)
    puts("Outside");
...

```

Пример 6. Дано целое число — угол α в градусах от 0 до 360. Вычислить $\sin \alpha$, $\cos \alpha$, $\operatorname{tg} \alpha$ и $\operatorname{ctg} \alpha$ с двумя знаками после запятой.

Ввод	45	300
Вывод	0.71	-0.87
	0.71	0.5
	1.00	-1.73
	1.00	-0.58

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main()

```

```

5 {
6     double alpha, t;
7     scanf("%lf", &alpha);
8
9     alpha = M_PI * alpha / 180.0;
10    printf("%.2lf\n", sin(alpha));
11    printf("%.2lf\n", cos(alpha));
12
13    t = tan(alpha);
14    printf("%.2lf\n", t);
15    printf("%.2lf\n", 1.0 / t);
16    return 0;
17 }

```

В программе угол переведён из градусов в радианы, так как аргументы тригонометрических функций измеряются в радианах. Отдельной функции для котангенса в библиотеке нет, поэтому котангенс выражается через тангенс.

Пример 7. Даны 3 вещественных числа a, b, c . Определить число корней квадратного уравнения $ax^2 + bx + c = 0$ ($a \neq 0$). Если корни есть, то вывести их, в противном случае вывести текст **No solution**.

Ввод	1.0 2.0 3.0	1.0 2.0 1.0	2.0 3.0 1.0
Вывод	No solution	-1.00	-1.00 -0.50

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      double a, b, c, D, x1, x2;
7      scanf("%lf %lf %lf", &a, &b, &c);
8      D = b * b - 4 * a * c;
9      if (D > 0)
10     {
11         x1 = (- b + sqrt(D)) / (2.0 * a);
12         x2 = (- b - sqrt(D)) / (2.0 * a);
13         printf("%.2lf %.2lf\n", x1, x2);
14     }
15     else if (D == 0)

```



```

16      {
17          x1 = -b / (2.0 * a);
18          printf("%.2lf\n", x1);
19      }
20      else
21          puts("No solution");
22      return 0;
23  }

```

Пример 8. Даны 2 целых числа r от 1 до 100 — радиус и α от 1 до 359 — угол в градусах. Вычислить площадь сегмента радиуса r с углом α .

Ввод	10 90
Вывод	28.54

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      int r, a;
7      double sector, triangle, segment, alpha;
8      scanf("%d %d", &r, &a);
9
10     alpha = M_PI * a / 360.0;
11     sector = radius * radius * a / 360.0;
12     triangle = 0.5 * radius * radius * sin(alpha);
13     segment = sector - triangle;
14
15     printf("%.2lf\n", segment);
16     return 0;
17 }

```

Обратите внимание, что если в 12 строке вместо $0.5 * \dots$ поставить $1 / 2 * \dots$, то результат вычислений будет равен нулю (так как $1 / 2$ — это целочисленное деление 1 на 2). Соответственно, площадь будет вычислена неверно.

Пример 9. Даны вещественные координаты двух точек $(x_1; y_1)$ и $(x_2; y_2)$. Необходимо найти площадь пересечения квадратов с центрами в данных точках и стороной 1.

Ввод	0.25 0.25 2.55 3.55	0.1 0.1 0.8 0.7
Вывод	0.00	0.12

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      double x1, y1, x2, y2, dx, dy;
6      double S = 0;
7      scanf("%lf %lf", &x1, &y1);
8      scanf("%lf %lf", &x2, &y2);
9
10     dx = fabs(x2 - x1);
11     dy = fabs(y2 - y1);
12     if (dx <= 1 && dy <= 1) {
13         S = (1 - dx) * (1 - dy);
14     }
15     printf("%.2lf\n", S);
16     return 0;
17 }
```

Выражения $|x_2 - x_1|$ и $|y_2 - y_1|$ встречается в коде два раза (в условном операторе и в формуле площади). Для того чтобы избавиться от дублирования кода, добавлены вспомогательные переменные dx и dy .

Пример 10. Даны три вещественных числа от -1000 до 1000. Определить, существует ли треугольник с данными сторонами. Если он существует, то определить его тип: равнобедренный, равносторонний, прямоугольный или обычный. Теорему Пифагора проверять с точностью до 1 знака после запятой, то есть $|a^2 + b^2 - c^2| < 0.01$.

Ввод	3.00 4.00 5.00	3.00 4.00 3.00	1.00 1.00 1.41
Вывод	right	isosceles	right isosceles

Ввод	0.03 0.03 0.03	4.00 5.00 6.00	1.00 3.00 5.00
Вывод	equilateral	simple	not exist

```
1  #include <stdio.h>
2
3  int main()
4  {
5      double a, b, c;
6      scanf("%lf %lf %lf", &a, &b, &c);
7      if (a + b > c && b + c > a && c + a > b)
8      {
9          if (a == b && b == c && c == a)
10             puts("equilateral");
11         else
12         {
13             double a2 = a * a;
14             double b2 = b * b;
15             double c2 = c * c;
16             const double eps = 0.01;
17             int abc = fabs(a2 + b2 - c2) < eps;
18             int bca = fabs(b2 + c2 - a2) < eps;
19             int cab = fabs(c2 + a2 - b2) < eps;
20             int is_rect = (abc || bca || cab);
21             int is_iso = (a == b ||
22                           b == c ||
23                           b == a);
24             if (is_rect)
25                 puts("right");
26             if (is_iso)
27                 puts("isosceles");
28             if (!is_rect && !is_iso)
29                 puts("simple");
30         }
31     }
32     else
33         puts("not exist");
34     return 0;
35 }
```

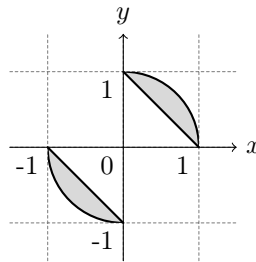
В коде введены две вспомогательные переменные, которые выполняют роль флага: `is_rect` (равно 1, если треугольник прямоугольный, 0 иначе) и `is_iso` (равно 1, если треугольник равнобедренный, 0 иначе). Варианты без использования флагов либо содержат повторные вычисления логических выражений. Также удобно выводить ответ для прямоугольного

равнобедренного треугольника.

Также обратите внимание, что если строки не помещаются на экран, то можно делать переносы как на строках 17-19.

4.14 Задания для самостоятельной работы

- Дано целое число x . Выписать соответствующие логические выражения, которые принимают значение «истина», если целое число x :
 - является нечетным отрицательным числом;
 - не принадлежит множеству $[0; 30) \cup \{50\}$;
 - является четырехзначным палиндромом (например, 2002 или 9999).
- Написать выражение, которое принимает значение «истина», если шахматный король в клетке (x_1, y_1) бьет поле (x_2, y_2) , где x_1, x_2, y_1, y_2 — целые числа от 1 до 8.
- Написать выражение, которое принимает значение «истина», если точка с координатами $(x; y)$ лежит в следующей области:



- Дано целое число n из всего допустимого диапазона типа `int`. Если существует правильный n -угольник со стороной 1, найти его площадь, иначе вывести `Does not exist`:

```
#include <stdio.h>
#include <math.h>
int main()
{
    int n;
    double S;
    scanf("%d", &n);

    return 0;
}
```

5. Программа должна печатать сумму двух максимальных чисел из трёх заданных. Найти все ошибки и оставить короткие комментарии к ним:

```
int main()
{
    int a, b, c, int ans;
    scanf("%d", a);
    scanf("%d", b);
    scanf("%d", c);
    if a < b && a < c
        ans = b + c
    if (b < c) && (b < a)
        ans == a + c
    if (c < a && c < b);
        ans += a + b;
    printf("%d/n", ans);
    return;
}
```

4.15 Практикум на ЭВМ

1. Дано целое число x от -1000 до 1000. Если x принадлежит множеству $[0; 10) \cup (20; 30]$, вывести **yes**, иначе вывести **no**.

Ввод	5	15	30
Вывод	yes	no	yes

2. Даны 2 целых числа x, y . от -1000 до 1000. Если числа разной чётности, вывести **yes**, иначе вывести **no**.

Ввод	1 2	-2 4	-5 -7
Вывод	yes	no	no

3. Даны 3 целых числа x, y, z от -1000 до 1000. Если среди них есть хотя бы одно отрицательное, вывести сумму чисел, иначе вывести произведение чисел.

Ввод	2 12 -3	2 12 3	-2 -12 -3
Вывод	11	72	-17

4. Даны 2 вещественных числа — координаты точки А. Определить, где лежит точка А относительно единичного круга. Вывести **in** (внутри), **out** (снаружи) или **boundary** (на границе).

Ввод	1.0 0.0	0.5 -0.6	-1.0 1.0
Вывод	boundary	in	out

5. Даны 3 вещественных числа a, b, c . Определить число корней уравнения $ax^2 + bx + c = 0$ (a может быть нулем). Если корней бесконечно много, вывести **infinity**.

Ввод	1.0 5.0 6.0	0.0 0.0 0.0	0.0 0.0 1.0
Вывод	2	infinity	0

6. Даны 2 вещественных числа — координаты точки А. Если точка А находится внутри кольца с радиусами 1 и 2 и центром в начале координат, то вывести **yes**, иначе **no**.

Ввод	1.0 1.0	2.0 2.0
Вывод	yes	no

7. Даны 4 целых числа от 1 до 8, которые задают две клетки шахматного поля. Вывести **knight**, если с первой клетки можно попасть на вторую ходом коня. Вывести **queen**, если с первой клетки можно попасть на вторую ходом ферзя. Иначе вывести **nothing**.

Ввод	3 4 7 8	3 4 1 3	3 4 5 7
Вывод	queen	knight	nothing

8. Даны 2 вещественных числа x, y . Проверить, находится ли точка с координатами (x, y) строго внутри квадрата с вершинами $(1, 1)$, $(1, -1)$, $(-1, -1)$, $(-1, 1)$.

Ввод	0.6 -0.6	-1.4 0.4	0.7 1.2
Вывод	yes	no	no

9. Даны 3 вещественных положительных числа от 0 до 1000 с 1 знаком после запятой — стороны треугольника. Если такой треугольник существует, то найти радиус вписанной и описанной окружностей (с точностью 2 знака после запятой). Иначе вывести «do not exist».

Ввод	3.0 4.0 5.0	1.0 2.0 4.0
Вывод	1.00 2.50	do not exist

10. Даны 3 целых числа a, b, c от 1 до 1000, вычислить среднее гармоническое $\frac{3}{\frac{1}{a} + \frac{1}{b} + \frac{1}{c}}$, арифметическое $\frac{a+b+c}{3}$, геометрическое $\sqrt[3]{abc}$ и квадратическое $\sqrt{\frac{a^2+b^2+c^2}{3}}$ этих чисел.

Ввод	3 3 3	2 4 8
Вывод	3.00 3.00 3.00 3.00	3.43 4.67 4.00 5.29

5 Цикл while

В текущей теме рассматриваются циклы с оператором `while`. Также разобраны некоторые примеры алгоритмов, которые удобно реализовывать с применением данных циклов.

5.1 Цикл с предусловием while

В предыдущих темах были разобраны программы, где действия выполнялись последовательно друг за другом и программы с условным выбором действий. Теперь рассмотрим способ многократного повторения операций.

Начнем с простой задачи: дано целое положительное число n от 1 до 15. Найти минимальную степень 2^k такую, что $2^k > n$. Например, для $n = 5$ ответом будет 8, а для $n = 8$ ответ — 16.

Попробуем решить эту задачу с помощью условного оператора. Решение построим следующим образом: будем перебирать все степени двойки 2, 4, 8 и 16, пока не найдём подходящую.

```
1  int p = 2, n = 5;
2  if (p <= n)
3      p *= 2;
4  if (p <= n)
5      p *= 2;
6  if (p <= n)
7      p *= 2;
8  if (p <= n)
9      p *= 2;
10 printf("%d", n);
```

В последних двух условных операторах условие «ложно» и соответствующие ветки с умножением не выполнились (строки 7 и 9).

строка	код	p	n
1	<code>int p = 2, n = 5;</code>	2	5
2	<code>if (p <= n)</code>	2	5
3	<code>p *= 2;</code>	4	5
4	<code>if (p <= n)</code>	4	5
5	<code>p *= 2;</code>	8	5
6	<code>if (p <= n)</code>	8	5
8	<code>if (p <= n)</code>	8	5
10	<code>printf("%d ", n);</code>	8	5

Два важных наблюдения: во-первых, условный оператор и его тело повторяется абсолютно без изменений; во-вторых, как только условие становится ложным его выполнение прекращается. Как раз для таких повторяющихся операторов можно использовать цикл с предусловием. Для этого заменим оператор `if` на оператор `while`:

```
int p = 2, n = 5;
while (p <= n)
    p *= 2;
printf("%d", n);
```

Тело цикла `p *= 2;` будет выполняться до тех пор, пока условие `p <= n` будет истинно.

строка	код	p	n
1	<code>int p = 2, n = 5;</code>	2	5
2	<code>while (p <= n)</code>	2	5
3	<code>p *= 2;</code>	4	5
2	<code>while (p <= n)</code>	4	5
3	<code>p *= 2;</code>	8	5
2	<code>while (p <= n)</code>	8	5
4	<code>printf("%d", n);</code>	8	5

Рассмотрим ещё один пример. Дано целое положительно число n от 1 до 10^6 . Необходимо определить, на сколько нулей оканчивается число. Для $n = 103000$ ответом будет 3, для $n = 1000000$ ответ — 6.

По умолчанию будем считать, что ответ равен нулю. Пока число оканчивается на нуль, отбрасываем последнюю цифру и увеличиваем ответ на единицу.

```
int ans = 0, n = 103000;
while (n % 10 == 0)
{
    n /= 10;
    ans++;
}
```

Как и в случае с условным оператором, если тело состоит из нескольких операторов, то их объединяют составным оператором в виде фигурных скобок { ... }.

Циклы с предусловием в общем виде выглядят следующим образом:

```
while (condition)
    action;
```

и

```
while (condition)
{
    action1;
    action2;
    ...
}
```

5.2 Цикл с постусловием do while

Напишем программу-эхо. Что делает эхо? Повторяет то, что слышит. Передавать мы будем в программу числа в столбик, а она будет выводить их на экран до тех пор, пока не встретится ноль. Решение можно оформить с помощью цикла с постусловием:

```
int a;
do
{
    scanf("%d", &a);
    printf("%d", a);
} while (a != 0);
```

Циклы с постусловием в общем виде выглядят следующим образом:

```
do
    action;
while (condition);
```

или

```
do {  
    action1;  
    action2;  
    ...  
} while (condition);
```

Хороший стиль оформления кода. Открывающаяся скобка должна быть либо сразу после слова `do`, либо на новой строке (в зависимости от стиля оформления остального кода).

Хороший стиль оформления кода. Ключевое слово `while` оператора `do-while` записывается на той же строке, что и закрывающая скобка, чтобы избежать путаницы с оператором `while`.

5.3 Переменная-счётчик

Часто в циклах используется специальная переменная-счётчик, которая изменяется определенным образом в теле цикла (обычно увеличивается или уменьшается на 1) и при этом является ограничителем количества действий.

Например, вывести текст «Hello!» 5 раз. Конечно, можно написать 5 одинаковых команд `puts("Hello!")`. Но мы сделаем лучше!

Давайте заведём переменную-счетчик `i`. Исторически сложилось, что именно такое имя переменной используется для счетчиков (от англ. `index`). Данная переменная будет хранить информацию о том, сколько слов «Hello!» осталось вывести. Начнём с `i = 5`, а после каждого вывода будем уменьшать значение `i` на 1.

```
int i = 5;  
while (i > 0)  
{  
    puts("Hello!");  
    i--;  
}
```

Стоит отметить, что условие $(i > 0)$ будет проверено 6 раз. После завершения цикла значение переменной `i` будет равно 0.

Ещё один пример: дано целое положительное число n , необходимо вывести все целые неотрицательные числа от 0 до $n - 1$. В этом случае удобно использовать счётчик i , изначально равный 0, который будет увеличиваться на 1 после каждого вывода.

```
int i = 0, n;
scanf("%d", &n);
while (i < n)
{
    printf("%d\n", i);
    i++;
}
```

5.4 Заикливание

При работе с циклами иногда возникает проблема: цикл не останавливается (программа заикливается).

```
1 int p = 2, n = 5;
2 while (p <= n);
3     p *= 2;
4 printf("%d", n);
```

Проблема может быть вызвана всё тем же пустым оператором, который начинающие программисты по ошибке ставят в виде точки с запятой после условия (на строке 2). Это не является синтаксической ошибкой и аналогична коду представленному ниже:

```
1 int p = 2, n = 5;
2 while (p < n) {
3
4 }
5 p *= 2;
6 printf("%d", n);
```

В терминале заикливание проявляется в том, что программа ничего не выводит и приветственная строка не появляется. В таких случаях необходимо завершить программу комбинацией: **ctrl + c**.

```
$ ./prog

^C
$ ./prog
```

Рассмотрим еще одну типичную ошибку: забытые фигурные скобки. Например:

```
1  #include <stdio.h>
2  int main() {
3      int ans = 0, n = 103000;
4      while (n % 10 == 0)
5          ans++;
6          n /= 10;
7      printf("%d\n", ans);
8      return 0;
9  }
```

Программа тоже заикнется. Прежде чем продолжить работу, необходимо завершить программу комбинацией `ctrl + c`.

Далее есть несколько способов найти причину заикливания: отладочная печать и специализированная программа отладчик.

5.5 Отладочная печать

Отладочная печать представляет собой дополнительный вывод значений переменных с помощью функций `puts` и `printf`.

```
1  #include <stdio.h>
2  int main() {
3      int ans = 0, n = 103000;
4      while (n % 10 == 0)
5          printf("%d\n", n);
6          ans++;
7          n /= 10;
8      printf("%d\n", ans);
9      return 0;
10 }
```

Стоит помнить, что функция `printf()` без переноса строки не гарантирует вывод на экран, поэтому обязательно добавляем символ переноса строки `'\n'` вручную. На экране появится:

```
$ gcc prog.c -o prog -Wall
$ ./prog
103000
103000
103000
```

```
103000
...
```

В этом случае становится понятно, что оператор `n /= 10` не выполняется по причине того, что в цикле отсутствуют фигурные скобки.

5.6 Отладка в gdb

Инструментом `gdb` (GNU Debugger) пользуются при разработке серьезных программ. Тем не менее можно попробовать использовать его и для простых задач. Для начала скомпилируем программу с дополнительным флагом `-g`:

```
$ gcc prog.c -o prog -Wall -g
```

Далее запустим отладчик:

```
$ gdb prog
```

Посмотрим код программы начиная с первой строки:

```
(gdb) list 1
```

Получим сообщение с кодом программы и номерами строк:

```
1  #include <stdio.h>
2  int main() {
3      int ans = 0, n = 103000;
4      while (n % 10 == 0)
5          printf("%d\n", n);
6          n /= 10;
7          ans++;
8      printf("%d\n", ans);
9      return 0;
10 }
```

Укажем точку останова (англ. breakpoint), то есть номер строки, до которой программа выполнится, а в этой строке действия программы выполняться уже не будут. Ставить точку останова лучше непосредственно перед началом цикла:

```
(gdb) break 3
```

Получим сообщение о добавлении точки останова:

```
Breakpoint 1 at 0x652: file prog.c, line 3.
```

Цикл while

После запускаем выполнение программы до точки останова (в нашем примере код первой и второй строки):

```
(gdb) run
```

Получим сообщение о том, что программа выполнена до точки останова и готова начать пошаговое выполнение действий оставшейся части кода:

```
Starting program: /home/alien/a  
  
Breakpoint 1, main () at prog.c:3  
3      int ans = 0, n = 100;
```

Добавляем переменные, за которыми будем наблюдать:

```
(gdb) display n  
(gdb) display ans
```

Выполняем команду на строке с точкой останова (в нашем примере код третьей строки):

```
(gdb) next
```

Получаем сообщение с кодом следующей строки и со значениями наблюдаемых переменных:

```
4      while (n % 10 == 0)  
1: n = 100  
2: ans = 0
```

Далее нажимаем **enter** (запустить последнюю команду gdb, то есть **next** в нашем случае):

```
5      printf("%d\n", n);  
1: n = 100  
2: ans = 0
```

Снова нажимаем **enter**:

```
100  
4      while (n % 10 == 0)  
1: n = 100  
2: ans = 0
```

Снова нажимаем **enter**:


```

5           printf ("%d\n", n);
1: n = 100
2: ans = 0

```

Заметим, что значение переменной n не меняется. Значит, тело цикла определено некорректно из-за отсутствия фигурных скобок.

Теперь выйдем из gdb:

```
(gdb) quit
```

Подтвердим свои намерения клавишей у.

5.7 Примеры

Пример 1. Дано целое n от 1 до 10000. Найти максимальный квадрат целого числа k , который меньше n , то есть $k^2 < n$.

Ввод	30	100
Вывод	25	81

Стоит обратить внимание, что попытка решения данной задачи в одно действие `ans = pow((int)sqrt(n), 2);` имеет два недостатка: некорректная обработка полных квадратов и переход в вещественную арифметику.

Для начала найдем перебором целое число k , начиная с 1, пока $k^2 < n$. Невыполнение условия $k^2 < n$ означает, что $k^2 \geq n$ (то есть мы перескочили правильный ответ на 1). И правильным ответом будет $(k - 1)^2$.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int n, k = 1, ans;
6      scanf ("%d", &n);
7      while (k * k < n)
8          k++;
9      ans = (k - 1) * (k - 1);
10     printf ("%d\n", ans);
11     return 0;
12 }

```

Несложно убедиться, что данный алгоритм совершит не более 100 операций умножения целых чисел при $n \leq 10000$. Ниже будет приведено решение данной задачи методом бинарного поиска, который найдёт ответ не более чем за 7 операций умножения.

Пример 2. Дано целое число от 1 до 10^{18} . Посчитать количество цифр этого числа.

Ввод	2019	123456789012345
Вывод	4	15

Идея решения заключается в пошаговом вычёркивании последних цифр данного числа до тех пор, пока они не закончатся. Вычёркивание последней цифры числа n можно описать как целочисленное деление на 10, то есть $n /= 10$.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int ans = 0;
6      long long n;
7      scanf("%lld", &n);
8
9      while (n != 0)
10     {
11         n /= 10;
12         ans++;
13     }
14     printf("%d\n", ans);
15     return 0;
16 }
```

Пример 3. Дано целое число от 1 до 10^{18} . Вывести цифры числа в обратном порядке.

Ввод	2019	1000000
Вывод	9102	0000001

Решение отличается от решения предыдущей задачи только тем, что перед вычёркиванием каждой последней цифры числа она выводится на

экран. Данный подход позволяет найти цифры числа в других системах счисления, если вместо деления на 10 использовать основание соответствующей системы.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int digit;
6      long long n;
7      scanf("%lld", &n);
8
9      while (n != 0)
10     {
11         digit = n % 10;
12         printf("%d", digit);
13         n /= 10;
14     }
15     printf("\n");
16     return 0;
17 }
```

Пример 4. Дано целое n от 1 до 30000. Вывести все делители числа n через пробел.

Ввод	30	31
Вывод	1 2 3 5 6 10 15 30	1 31

Решение заключается в переборе всех чисел d от 1 до n и выводе таких d , что n делится на d без остатка. Обратите внимание на вывод: чтобы числа выводились через пробел, достаточно выводить пробел после каждого числа, а завершить весь вывод переносом строки.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int d = 1;
6      scanf("%d", &n);
7      while (d <= n)
8      {
9          if (n % d == 0)
```

```

10             printf("%d ", d);
11             d++;
12         }
13         printf("\n");
14         return 0;
15     }

```

Пример 5. Дано целое n от 1 до 30000. Проверить, является ли данное число — простым (вывести `prime` или `not prime` соответственно).

Ввод	30	101
Вывод	not prime	prime

Заметим, что вместо очевидного перебора потенциальных делителей d от 2 до $n - 1$ можно перебрать все целые числа d от 2 до \sqrt{n} . Действительно, если у числа n есть делитель $d > \sqrt{n}$, то у него же есть делитель $\frac{n}{d} < \sqrt{n}$. Если не нашлось ни одного такого делителя d , то число n простое.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int d = 2, divisors = 0, n;
6      scanf("%d", &n);
7      while (d * d <= n)
8          if (n % d == 0)
9              divisors++;
10         if (divisors == 0)
11             puts("prime");
12         else
13             puts("not prime");
14         return 0;
15     }

```

Пример 6. Дано целое положительное число от 1 до 20. Вычислить $n! = 1 \cdot 2 \cdot 3 \dots n$ (факториал).

Ввод	5	20
Вывод	120	2432902008176640000

Ничего не изменится, если переставить множители: $20! = 1 \cdot 20 \cdot 19 \cdot \dots \cdot 3 \cdot 2$. Это даст возможность использовать саму переменную n как счетчик. Стоит отметить, что результат $20!$ не помещается в тип `int`, но помещается в тип `long long`.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int n;
6      long long ans = 1;
7      scanf("%d", &n);
8      while (n > 1)
9      {
10         ans *= n;
11         n--;
12     }
13     printf("%lld\n", ans);
14     return 0;
15 }
```

В следующих примерах мы рассмотрим два основных способа ввода набора чисел:

1. ввод чисел завершается числом со специальным значением, отличным от остальных чисел (например, вводятся положительные числа, а ввод заканчивается нулём);
2. вводится количество чисел, а затем сами числа.

Пример 7. Дана последовательность целых чисел (их количество заранее не задано). Каждое число принимает значение от 1 до 1000. Ввод заканчивается нулём. Найти сумму всех заданных чисел. Гарантируется, что ответ не превосходит 10^9 .

Ввод	1 2 3 4 5 0	2 0
Вывод	15	2

Первый вариант решения (цикл с предусловием).

```

1  #include <stdio.h>
2
3  int main()
```

```

4 {
5     int a, sum = 0;
6     scanf("%d", &a);
7     while (a != 0)
8     {
9         sum += a;
10        scanf("%d", &a);
11    }
12    printf("%d\n", sum);
13    return 0;
14 }

```

Второй вариант решения (цикл с постусловием):

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int a, sum = 0;
6     do
7     {
8         scanf("%d", &a);
9         sum += a;
10    } while (a != 0);
11    printf("%d\n", sum);
12    return 0;
13 }

```

Стоит обратить внимание, что последний считанный нуль здесь также складывается, что не совсем удобно для решения аналогичной задачи с произведением.

В общем случае для вычисления результата $a_1 \circ a_2 \circ \dots \circ a_n$ (где \circ — произвольный ассоциативный оператор, например, сложение, умножение, максимум или минимум) алгоритм будет следующим:

```

ans = e;
scanf("%d", &a);
while (a != 0)
{
    ans = ans o a;
    scanf("%d", &a);
}

```

Здесь e является нейтральным элементом по отношению к ассоциативному оператору \circ , то есть $e \circ a = a \circ e == a$. Такими элементами могут быть: 0 — для сложения, 1 — умножения, $-\infty$ — для максимума, $+\infty$ — для минимума.

Пример 8. Дана последовательность целых чисел (их количество заранее не задано). Каждое число последовательности принимает значение от -1000 до 1000 и отлично от нуля. После последовательности вводится нуль. Найти минимальное число в заданной последовательности.

Ввод	2 3 1 4 5 0	2 -3 1 -4 5 0
Вывод	1	-4

Первый вариант решения: определим нейтральный элемент для оператора минимум по ограничениям значений чисел из условия. Так как все числа меньше 1000 , то в качестве нейтрального элемента можно использовать число 1001 .

```

1  #include <stdio.h>
2
3  int main()
4  {
5      const int inf = 1001;
6      int a, min = inf;
7      scanf("%d", &a);
8      while (a != 0)
9      {
10         if (min > a)
11             min = a;
12         scanf("%d", &a);
13     }
14     printf("%d\n", min);
15     return 0;
16 }
```

Второй вариант решения: определим нейтральный элемент для оператора минимум по ограничениям числового типа `int`. Максимальное значение в типе равно:

$$2^{31} - 1 = 8 \cdot 16^7 - 1 = 80000000_{16} - 1 = 7FFFFFFF_{16}.$$

Код изменится только в пятой строке:

```
const int inf = 0x7FFFFFFF;
```

Для аналогичной задачи с поиском максимума можно использовать минимальное отрицательное число типа `int` равное `0x80000000`.

Третий вариант решения — инициализация ответа по первому числу. Это наиболее аккуратный вариант, так как в нём отсутствуют явные константы. Для этого следует изменить алгоритм поиска минимума: минимум без условного оператора инициализировать как первое введённое число.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a, min;
6      scanf("%d", &min);
7      scanf("%d", &a);
8      while (a != 0)
9      {
10         if (min > a)
11             min = a;
12         scanf("%d", &a);
13     }
14     printf("%d\n", min);
15     return 0;
16 }
```

Пример 9. В первой строке вводится одно целое число n от 1 до 1000. На второй строке дана последовательность из n различных целых чисел от -1000 до 1000 . Найти минимальное число из данной последовательности и вывести номер его позиции.

Ввод	5 12 13 11 14 15
Вывод	11 3

Пусть переменная-счётчик `i` отмечает, сколько уже чисел обработано. Переменные `min` и `imin` равны значению и позиции минимального числа из первых `i` чисел соответственно. Первое число последовательности обрабатывается отдельно, поэтому сразу можно инициализировать эти две переменных.


```

1  #include <stdio.h>
2
3  int main()
4  {
5      int n, a, min, imin, i;
6      scanf("%d", &n);
7
8      scanf("%d", &a);
9      i = 1;
10     min = a;
11     imin = 1;
12     while (i < n){
13         scanf("%d", &a);
14         i++;
15         if (a < min)
16         {
17             min = a;
18             imin = i;
19         }
20     }
21     printf("%d %d\n", min, imin);
22     return 0;
23 }
```

Пример 10. Даны два целых числа от 1 до 10^9 . Найти наибольший общий делитель этих чисел.

Ввод	40 12	20 17
Вывод	4	1

Используем известный алгоритм Евклида для нахождения наибольшего общего делителя:

$$(a; b) = (a - b; b).$$

В этом алгоритме из большего числа вычитается меньшее. Например, наибольший общий делитель чисел 40 и 12 будет равен 4:

$$(40; 12) = (28; 12) = (16; 12) = (4; 12) = (4; 8) = (4; 4) = 4$$

Существует модифицированная версия алгоритма Евклида, в которой вместо вычитания используется деление с остатком:

$$(a; b) = (b; a \bmod b)$$

Как только число b станет нулем, число a и будет наименьшим общим делителем исходных чисел. Например:

$$(40; 12) = (12; 4) = (4; 0) = 4$$

Модифицированный алгоритм будет работать быстрее и позволит не думать о том, какое из чисел больше. Действительно, если первое число меньше, то операция $(a; b) = (b; a \bmod b) = (b; a)$ просто переставляет аргументы местами.

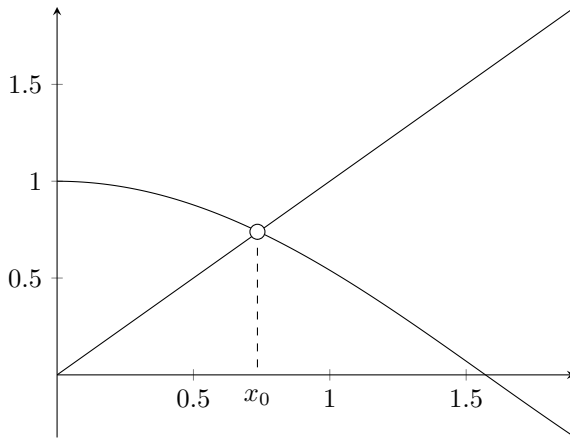
```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      int a, b, d;
7      scanf("%d %d", &a, &b);
8      while (b != 0)
9      {
10         d = a % b;
11         a = b;
12         b = d;
13     }
14     printf("%d\n", a);
15     return 0;
16 }
```

Пример 11. Дано вещественное число ε от 0.0001 до 0.1. Найти минимальное положительное решение уравнения $\cos x = x$ с абсолютной погрешностью не более ε . Вывести ответ с точностью 6 знаков после запятой.

Ввод	0.01
Вывод	0.739085

Решение основано на идее бинарного поиска. Заметим, что минимальное положительное решение уравнения лежит на отрезке $[0; \frac{\pi}{2}]$. Обозначим его x_0 .



Тогда верно, что:

$$\begin{cases} \cos x > x, & x \in [0, x_0) \\ \cos x < x, & x \in (x_0, \frac{\pi}{2}] \end{cases}$$

Начнем поиск корня на интервале от $l = 0$ до $r = \frac{\pi}{2}$. Определим расположение точки $m = \frac{l+r}{2}$ — середины интервала $[l; r]$. Сравним $\cos(m)$ и m . Если $\cos(m) > m$, то $m < x_0$ и область поиска сузим до интервала $[m; r]$. В противном случае область поиска — интервал $[l; m]$. После каждой такой итерации область поиска будет сокращаться в два раза. Поиск продолжим до тех пор, пока размер области не станет меньше ε . Тогда корень найдём с необходимой точностью.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      double left = 0.0, right = M_PI_2, middle;
7      double eps;
8      scanf("%lf", &eps);
9      while (right - left > eps)
10     {
11         middle = (left + right) / 2;
12         if (cos(middle) > middle)
13             left = middle;
14         else
15             right = middle;

```

```

16     }
17     printf("%.6lf\n", middle);
18     return 0;
19 }
```

Оценим n — количество итераций при $\varepsilon = 0.0001$. Так как на каждой итерации интервал сокращается в два раза, то

$$\frac{\pi}{2^n} < 0.0001.$$

Неравенство верно для $n \geq 12$. Значит, необходимая точность будет достигнута за 12 итераций.

Пример 12. Приведём альтернативное решение задачи из примера 1, используя бинарный поиск.

Дано целое n от 1 до 10000. Найти максимальный квадрат целого числа k , который меньше n , то есть $k^2 < n$.

Ввод	30	200	10000
Вывод	25	196	9801

Метод бинарного поиска может применяться не только для монотонных вещественных функций, но и для монотонных последовательностей. Рассмотрим его на примере возрастающей последовательности из квадратов целых чисел $0^2, 1^2, 2^2, \dots, k^2, \dots, 100^2$.

По условию задачи необходимо найти такое максимальное k , что $k^2 < n$. Так как k — целое число, то это означает, что

$$k^2 < n \leq (k+1)^2.$$

Начнём поиск k на интервале от $l = 0$ до $r = 100$ (ответ точно больше 0 и меньше 100 для любого допустимого n). Будем уменьшать область поиска $[l; r]$, поддерживая инвариант:

$$l^2 < n \leq r^2.$$

Как только значения l и r станут отличаться на 1, поиск остановим. Это означает, что l — искомое число.

Подробнее разберем процесс сжатия области поиска. Определим расположение целочисленной точки $m = \left\lfloor \frac{l+r}{2} \right\rfloor$ сравнением m^2 и n . Если $m^2 < n$, то верно неравенство:

$$m^2 < n \leq r^2,$$

и область поиска можно сузить до $[m; r]$. В противном случае верно неравенство:

$$l^2 < n \leq m^2,$$

и область поиска можно сузить до $[l; m]$.

Отличие от вещественного поиска состоит в том, что итерации продолжаютсЯ до тех пор, пока не останется единичный отрезок.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      unsigned int ans = 1, left = 0, right = 101;
6      int n;
7      scanf("%u", &n);
8
9      while (right - left > 1)
10     {
11         middle = (left + right) / 2;
12         if (middle * middle < n)
13             left = middle;
14         else
15             right = middle;
16     }
17     printf("%u\n", left * left);
18     return 0;
19 }
```

Оценим количество итераций в бинарном поиске. При начальном размере области поиска равном 100 в худшем случае будет 7 шагов сжатия:

$$100 \rightarrow 50 \rightarrow \mathbf{25} \rightarrow \mathbf{13} \rightarrow \mathbf{7} \rightarrow 4 \rightarrow 2 \rightarrow 1.$$

Если на текущей итерации размер области поиска четное число (100, 50, 4 и 2), то он уменьшается ровно в два раза (50, 25, 2 и 1), в противном случае (25, 13 и 7) — сжимается в два раза с округлением вверх (13, 7 и 4).

5.8 Задания для самостоятельной работы

1. Описать цикл, который печатает слово **Hello!** в бесконечном цикле. Как остановить такой вывод?
2. Что будет выведено при запуске данного кода:

```
#include <stdio.h>
int main(){
    int n = 100, ans = 0;
    while (n > 0){
        n = n / 2;
        printf("%d ", n);
        ans++;
    }
    printf("%d\n", ans);
    return 0;
}
```

3. Что будет на выводе программы при $n = 12$?

```
#include <stdio.h>
int main(){
    int n, ans = 0, d = 2;
    scanf("%d", &n)
    while (d < n){
        if (n % d == 0)
            ans++;
        d++;
    }
    printf("%d\n", ans);
    return 0;
}
```

4. Вставьте пропущенный код. На вход подаются ненулевые целые числа. Ввод заканчивается нулем. Посчитать количество чисел.

```
#include <stdio.h>
int main(){
    int a, ans;
    scanf("%d", &a);
```

```
printf("%d\n", ans);  
return 0;  
}
```

5. Сколько раз выполнится тело цикла, чтобы найти корень решения уравнения $\cos x = x$ (без алгебраических преобразований) бинарным поиском с абсолютной погрешностью не более 0.1, если начальный интервал равен $(0; \pi)$?

5.9 Практикум на ЭВМ

1. Дано целое число n от 1 до 10^9 . Найти наибольшую степень тройки, на которую делится число n без остатка. Решение не должно содержать функции из библиотеки «math.h». Вывести показатель степени.

Ввод	72	15	32
Вывод	2	1	0

2. Дано целое число n от 1 до 10^9 . Перевести n в двоичную систему счисления с обратной записью цифр (от младших разрядов к старшим). Решение не должно содержать функции из библиотеки «math.h».

Ввод	13	16
Вывод	1011	00001

3. Дано целое число n от 1 до 10^6 . Вывести все чётные делители числа n .

Ввод	12
Вывод	2 4 6 12

4. Дана последовательность целых положительных чисел от 1 до 10^3 , которая заканчивается нулём. Найти произведение нечётных чисел (гарантируется, что ответ не превосходит 10^9).

Ввод	1 3 2 5 0
Вывод	15

5. Дана последовательность целых положительных чисел от 1 до 10^3 , которая заканчивается нулём. Найти количество положительных нечётных чисел (гарантируется, что ответ не превосходит 10^9).

Ввод	-1 2 3 -4 5 -6 0
Вывод	2

6. В первой строке дано целое число n от 1 до 1000. На второй строке дана последовательность из n целых отличных друг от друга чисел со значениями от 1 до 1000. Вывести на первой строке минимальное число и его позицию. А на второй строке — максимальное число и его позицию.

Ввод	5 104 220 101 132 109
Вывод	101 3 220 2

7. Дано целое число n от 1 до 10^{18} . Вывести максимальную цифру числа n .

Ввод	1546323
Вывод	6

8. Даны два целых числа a и b от 1 до 10^9 . Вывести наименьшее общее кратное чисел a и b .

Ввод	40 12	20 17
Вывод	120	340

9. Дано целое положительное число n от 2 до 10^9 . Бинарным поиском найти максимальное целое число a такое, что $a^2 + 3a + 2 \leq n$.

Ввод	12	21
Вывод	2	3

10. Дано целое положительное число n от 1 до 100. Бинарным поиском найти n -ый по величине вещественный корень уравнения $\operatorname{tg} x = x$. Ответ вывести с точностью в два знака после запятой.

Ввод	1	3
Вывод	4.49	10.90

6 Символьный тип

В данной теме речь пойдет об одном специфичном целочисленном типе — символьном, который используется для работы с текстом. Компьютеры на низком уровне работают только с числами. Чтобы работать с буквами, необходимо договориться об общих правилах их представления в числовом виде. О таком автоматическом средстве кодирования–декодирования букв и пойдет речь ниже.

6.1 Таблица кодов ASCII

Таблица кодов ASCII (англ. American standard code for information interchange) — это таблица, в которой представлены печатные и непечатные символы. Каждый символ кодируется 7-битным числом, но для хранения используется 1 байт. В таблице закодированы печатные символы с 32 по 126 и непечатные символы с 0 по 31.

Список печатных символов с 32 по 126:

номер	0	1	2	3	4	5	6	7	8	9
30				!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

Как можно заметить, коды цифр лежат в диапазоне от 48 до 57. Заглавные буквы — от 65 до 90, а строчные буквы — от 97 до 122. Нет никакой необходимости запоминать все коды. Печатные символы достаточно заключать в одинарные кавычки. В частности, чтобы получить код буквы «А» достаточно написать её в одинарных кавычках:

```
'A'
```

32-й символ — пробел — является печатным. Его код можно получить так:

```
' '
```

Стоит обратить внимание, что символ обратного слепа «\» имеет специализированное назначение — экранирование. Это означает, что он сам по себе игнорируется, зато символ после него трактуется специальным образом. Например, символ новой строки:

```
'\n'
```

Символ табуляции:

```
'\t'
```

Символ непосредственно самого обратного слепа:

```
'\\'
```

Также специальное назначение имеет символ одинарной кавычки '. Поэтому объявлять одинарную кавычку необходимо в экранированном виде:

```
'\''
```

Стоит отметить, что большинство непечатных символов были нужны для управления первыми системами обмена информацией. Поэтому на текущий момент используются далеко не все из них. Список непечатных символов (неполный):

номер	константа	пояснение
0	'\0'	нулевой символ (символ окончания строки)
9	'\t'	Tab — табуляция
10	'\n'	Line Feed — перевод строки (LF)
12	'\r'	Carriage Return — возврат каретки (CR)

Последние два символа используются для генерации переноса строки в разных операционных системах. Новая строка в UNIX: LF; в Windows: CR + LF; в Mac OS: CR.

6.2 Арифметика и сравнение

Для хранения символов из ASCII-таблицы обычно используется тип `char` или `unsigned char`. Например:

```
char a = 'M';
unsigned char b = 'N';
```

В переменную **a** будет записано число 77, а в переменную **b** — число 78.

Для данных типов применимы все арифметические операции и операции сравнения, но в рамках диапазонов их значений: у типа **char** это целые числа от (-128) до 127, а у типа **unsigned char** — от 0 до 255.

В переменной **letter** записана строчная буква английского алфавита от 'a' до 'y'. Запишем в переменную **next** следующую за ней букву:

```
char letter = 'e';
char next = letter + 1;
```

В ответе получим $101 + 1 = 102$, то есть код буквы *f*.

Определим количество букв в алфавите между буквами *D* и *J*.

```
int answer = 'J' - 'D' - 1;
```

В переменной **answer** получим $74 - 68 - 1$, то есть 5 (их действительно пять: *E, F, G, H, I*).

Рассмотрим пример, иллюстрирующий работу с символами для цифр. Составим из двух символов '2' и '5' двузначное число 25.

```
char ch1 = '2', ch2 = '5';
int x = ch1 - '0';
int y = ch2 - '0';
int num = 10 * x + y;
```

Заметим тот факт, что символы цифр в таблице идут по порядку от 0 до 9. То есть чтобы из символа '2' (код 50) получить число 2, достаточно вычесть символ '0' (код 48). В переменной **x** будет значение $50 - 48 = 2$, в переменной **y** — значение $53 - 48 = 5$, а в переменной **num** — значение $10 * 2 + 5 = 25$.

Дана любая строчная буква из диапазона от 'a' до 'z'. Определим симметричную с конца алфавита букву. Например, для буквы 'd' (4-я по счету с начала алфавита) симметричной буквой будет 'w' (4-я по счету с конца алфавита).

```
char ch = 'd';
char position = ch - 'a';
char sym = 'z' - position;
```

В переменной **position** будет значение $100 - 97 = 3$. А в переменной **sym** — значение $122 - 3 = 119$, то есть код буквы *w*.

Важно запомнить: чтобы найти порядковый номер строчной буквы в алфавите, достаточно вычесть из неё 'a'. Аналогично, чтобы найти порядковый номер заглавной буквы в алфавите, достаточно вычесть из неё 'A', а чтобы получить значение символьной цифры — вычесть '0'.

Непрерывные блоки букв и цифр в ASCII-таблице позволяют легко проверять символ на принадлежность какому-либо множеству.

<code>ch >= '0' && ch <= '9'</code>	цифра
<code>ch >= 'a' && ch <= 'z'</code>	строчная буква
<code>ch >= 'A' && ch <= 'Z'</code>	заглавная буква

Символ является буквой латинского алфавита тогда и только тогда, когда следующее выражения является истинным:

```
(ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')
```

Еще один интересный пример:

```
char a;  
a = '2' + '3';
```

Получаем, что $a = 101$. В этом случае константы '2' и '3' заменяются на их коды из ASCII-таблицы, то есть 50 и 51.

6.3 Смена регистра

Дана строчная буква. Получим соответствующую заглавную букву. Для этого стоит вспомнить, что код буквы 'd' больше кода буквы 'a' на столько, на сколько код буквы 'D' больше кода буквы 'A'.

```
char low = 'd';  
char position = low - 'a';  
char up = 'A' + position;
```

В переменной `position` будет $100 - 97 = 3$. В переменной `up` будет $65 + 3 = 68$, что соответствует заглавной букве 'D'.

Более компактный код выглядит следующим образом:

```
char low = 'd';  
char up = low - 'a' + 'A';
```

6.4 Битовые операции

Стоит отметить, что расположение символов именно в таком порядке обусловлено некой логикой в битовом представлении чисел. Код цифры в двоичной системе счисления представляет собой запись $0011XXXX_2$, где $XXXX_2$ — сама цифра в двоичной системе счисления. Например, 7 кодируется 00110111_2 , а 0 кодируется как 00110000_2 .

Чтобы получить число 7, можно не только вычитать из символа '7' символ '0', но и применять битовый оператор исключающего ИЛИ (он же XOR). Данный оператор одинаковые биты заменяет на 0, а разные биты — на 1. Применим исключающее ИЛИ для кодов символов '7' и '0':

'7'	00110111
'0'	00110000
7	00000111

В результате получим 7. Соответствующий код выглядит так:

```
char ch = '7';
int d = ch ^ '0';
```

Оператор исключающего ИЛИ применим и для смены регистра букв. Например, код буквы 'a' (01100001_2) отличается от кода буквы 'A' (01000001_2) только в одном бите. Для смены регистра символа в переменной `ch` вместо вычитания и сложения достаточно воспользоваться битовым оператором исключающего ИЛИ для кода `ch` и кода пробела:

'a'	01100001
' '	00100000
'A'	01000001

Соответствующий код выглядит так:

```
char ch = 'a';
char ch2 = ch ^ ' ';
```

6.5 Ввод–вывод символьного типа

Для ввода символов из терминала и вывода в терминал используются специализированные функции из библиотеки `stdio.h`:

```
int putchar(int);
int getchar();
```

Первая функция выводит символ с данным кодом на экран. Вторая — считывает символ со стандартного ввода (терминала) и возвращает код этого символа. Ввод и вывод чисел работает с типом `int`, но допускается работа и с `char`, и с `unsigned char`.

Пример программы, которая считывает один символ и выводит его обратно в терминал:

```
#include <stdio.h>

int main()
{
    char ch;
    ch = getchar();
    putchar(ch);
    return 0;
}
```

С помощью `putchar` и `getchar` можно работать и с непечатными символами. Например, данный вызов делает перенос строки:

```
putchar('\n');
```

Стоит отметить, что в знакомых нам функциях `scanf` и `printf` также есть возможность вывести и ввести отдельный символ. Спецификатор форматной строки в таком случае будет `%c`. Например, данный код считывает символ и выводит его:

```
char ch;
scanf("%c", &ch);
printf("%c", ch);
```

При работе с вводом-выводом необходимо помнить, что `char` — это всего лишь числовой тип. И выводить его можно и как символ, и как код. Например, вот так можно вывести символ, зная его код:

```
char code = 90;
printf("%c", code);
```

В терминале появится:

```
$ ./prog
Z
```

Выведем код в числовом виде для данного символа:

```
char ch = 'Z';  
printf("%d", (int)ch);
```

В терминале появится:

```
$ ./prog  
90
```

Выведем и символ, и его код:

```
char ch = 'Z';  
printf("%c %d", ch, (int)ch);
```

В терминале появится:

```
$ ./prog  
Z 90
```

Функцию `scanf` не следует использовать для ввода отдельного символа — применяйте `getchar`, так как `scanf` сам использует `getchar`. Функция `scanf` лучше подойдет для ввода нескольких символов по определенному формату. Например, вводятся две пары шахматных координат через двоеточие:

```
A5:B7
```

Соответствующий ввод выглядит так:

```
char c1, r1, c2, r2;  
scanf("%c%c:%c%c", &c1, &r1, &c2, &r2);
```

Обратите внимание на двоеточие: оно присутствует и на вводе, и в форматной строке.

6.6 Кириллица

Возможно, у многих возникает вопрос: почему всё пишется только на английском языке, где же кириллица? Самая главная причина заключается в том, что кодирование и декодирование кириллицы производится не по ASCII-таблице, а по более сложным таблицам, которые зависят от вида операционной системы. А точнее от настроек кодовой страницы, неправильный выбор которой является причиной появления «кракозябр» (нечитаемого текста). Чтобы увидеть русские буквы в терминале, необходимо согласовать кодировку вывода программы с кодировкой терминала.

В Windows по умолчанию используется таблица кодировки cp1251. Буквы русского алфавита хранятся в 1 байте с кодами в диапазоне 192 – 255 (сначала заглавные, потом строчные). То есть команда `putchar(193)` выведет букву Б. Код, приведённый ниже, выводит буквы русского алфавита только в том случае, если в терминале по умолчанию задана кодировка cp1251:

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 192; i <= 255; i++)
        putchar(i);
    putchar('\n');
    return 0;
}
```

В Linux по умолчанию используется таблица кодировки utf8. Полное название кодировки терминала можно узнать, выполнив команду:

```
locale
```

На экран выведется название:

```
ru_RU.UTF-8
```

Данная кодировка подразумевает хранение символов в 1, 2, 3 или 4 байтах. Буквы русского алфавита хранятся в 2 байтах в диапазоне: 0x0410 – 0x044F (шестнадцатиричные константы). Соответственно, для вывода каждого символа необходимо по 2 байта. Для хранения таких символов можно воспользоваться расширенным символьным типом `wchar_t` из библиотеки `wchar.h`. Но для этого придётся использовать расширенные константы (перед одинарными кавычками необходимо ставить букву L), задав локальную кодировку по умолчанию функцией

```
void setlocale(LC_ALL, "")
```

из библиотеки `locale.h`. Код, приведённый ниже, выводит буквы русского алфавита только в том случае, если в терминале по умолчанию задана кодировка utf8:

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
int main()
```

```
{
    wchar_t ch;
    setlocale(LC_ALL, "");
    for (i = 0x0410; i <= 0x044F; i++)
        putwchar(i);
    putwchar('\n');
    return 0;
}
```

Кодировка utf8 является более универсальной по двум причинам. Во-первых, если cp1251 не является кодировкой по умолчанию, то первый пример работать не будет, а второй код — будет, так как символы в utf8 имеют уникальные номера вне зависимости от операционной системы и настроек языка по умолчанию. Во-вторых, она является стандартом для web-программирования. Тем не менее, вывод русскими буквами средствами C лучше не осуществлять в рамках нашего курса.

6.7 Bash: перенаправление ввода-вывода

Терминал позволяет работать с текстовым файлом с помощью перенаправления стандартного потока ввода и вывода.

Вывод программы через функции `printf`, `puts`, `putchar` попадёт не на экран терминала, а в файл `out.txt` при запуске команды (предыдущее содержимое файла сотрётся):

```
$ ./prog > out.txt
```

Если необходимо вывод программы дописать в конец файла, сохранив его содержимое, то программа запускается следующим образом:

```
$ ./prog >> out.txt
```

Для ввода данных через функции `scanf`, `getchar` не из терминала, а из файла `in.txt`, программа запускается так:

```
$ ./prog < in.txt
```

Одновременный ввод из одного файла и вывод в другой:

```
$ ./prog < in.txt > out.txt
```

Таким образом можно отлаживать программы с большими входными данными, не набирая их вручную при каждом запуске программы. Входной текст для программы можно один раз сохранить в текстовом файле и потом запускать многократно как показано выше.

Также имеется очень удобная команда `cat` для просмотра текстовых файлов прямо в терминале:

```
$ cat in.txt
```

6.8 Примеры

Пример 1. Дана последовательность символов с точкой в конце. Вывести все ASCII-коды символов и общее количество символов.

Ввод	Astana 2019.
Вывод	65 115 116 97 110 97 32 50 48 49 57 46 12

Обработку каждого символа (считать, вывести код, прибавить к ответу 1) можно оформить в виде цикла `do ... while`. Цикл будет выполняться до тех пор, пока считанный символ отличается от точки.

```
1  #include <stdio.h>
2  int main()
3  {
4      int n = 0;
5      char ch;
6      do
7      {
8          ch = getchar();
9          printf("%d ", (int)ch);
10         n++;
11     } while (ch != '.');
12     printf("\n%d\n", n);
13     return 0;
14 }
```

Пример 2. Дана последовательность символов с переносом строки в конце. Вывести все символы, кроме цифр.

Ввод	Astana 2019!
Вывод	Astana !

Символы, отличные от цифр, имеют коды либо меньше '0', либо больше '9'. Поэтому их легко определить, сравнив текущий введённый символ с данными границами. А продолжать ввод будем до тех пор, пока символы отличны от '\n'.

```
1  #include <stdio.h>
2  int main()
3  {
4      char ch;
5      do
6      {
7          ch = getchar();
8          if (ch < '0' || ch > '9')
9              putchar(ch);
10     } while (ch != '\n');
11     return 0;
12 }
```

Пример 3. Дана строка. Заменить все строчные буквы на заглавные.

Ввод	Astana 2019!
Вывод	ASTANA 2010!

Смену регистра со строчного на заглавный можно произвести обычными арифметическими операциями `ch = ch - 'a' + 'A'` или более компактной формой `ch += 'A' - 'a'`. Для разнообразия обработку можно обернуть в цикл `while` Стоит не забыть, что первый символ вводится до начала цикла.

```
1  #include <stdio.h>
2  int main()
3  {
4      char ch;
5      ch = getchar();
6      while (ch != '\n')
7      {
8          if (ch >= 'a' && ch <= 'z')
9              ch += 'A' - 'a';
10         putchar(ch);
11         ch = getchar();
12     }
13     putchar('\n');
14     return 0;
15 }
```

Пример 4. Дана строка. Посчитать, сколько в этой последовательности букв.

Ввод	Astana 2019 and 2020!
Вывод	9

Немного модифицируем ввод из предыдущей программы. Для это стоит отметить, что оператор присваивания, как и другие операторы, возвращает значение. Например, `ch = '\n'` возвращает `'\n'`. Поэтому `ch = getchar()` возвращает значение только что введённого символа. Можно сразу проверить его на символ конца строки.

```

1  #include <stdio.h>
2  int main()
3  {
4      char ch;
5      int ans = 0;
6      while ((ch = getchar()) != '\n')
7      {
8          if (ch >= 'A' && ch <= 'Z' ||
9              ch >= 'a' && ch <= 'z')
10         {
11             ans++;
12         }
13     }
14     printf("%d\n", ans);
15     return 0;
16 }
```

Приведённое выше замечание про оператор присваивания объясняет, почему компилируется код `if (a = 10) a = 20;`. Оператор присваивания `a = 10` возвращает значение 10, которое условный оператор трактует как истину. Соответственно, этот код всегда присваивает `a` значение 20.

Пример 5. Дана строка. Посчитать количество слов (слово — непрерывная последовательность букв).

Ввод	Hello, Julia! It's me!	2007-2018 years
Вывод	5	1

Сразу стоит отметить, что если бы все слова были разделены одним пробелом, то можно было бы просто посчитать количество пробелов и

прибавить единицу. Но здесь в качестве разделителей могут быть любые символы, причем в любом количестве.

Будем считать количество мест, где заканчиваются слова: текущий символ — не буква, предыдущий символ — буква. Для этого заведем 2 флага: `current` и `prev`. Где

$$\text{current} = \begin{cases} 1, & \text{если текущий символ — буква,} \\ 0, & \text{если текущий символ — не буква,} \end{cases}$$

`prev` — значение `current` на предыдущем шаге. Важно не забыть инициализировать `current = 0`, так как первая буква не может быть разделителем для предыдущего слова.

```

1  #include <stdio.h>
2  int main()
3  {
4      char ch, current = 0, prev;
5      int ans = 0;
6      do
7      {
8          ch = getchar();
9          prev = current;
10         current = ch >= 'A' && ch <= 'Z' ||
11                ch >= 'a' && ch <= 'z';
12         if (prev && !current)
13             ans++;
14     } while (ch != '\n');
15     printf("%d\n", ans);
16     return 0;
17 }
```

Пример 6. Вводится 3 символа. Первый символ — цифра от 0 до 9. Второй символ — знак арифметического действия (+, − или *). Третий символ — снова цифра от 0 до 9. Необходимо произвести арифметическое вычисление.

Ввод	2+3	3*4
Вывод	5	12

```

1  #include <stdio.h>
2  int main()
```

```

3 {
4     char digit1, sign, digit2;
5     int ans = 0;
6     digit1 = getchar();
7     sign = getchar();
8     digit2 = getchar();
9
10    digit1 -= '0';
11    digit2 -= '0';
12
13    if (sign == '+')
14        ans = digit1 + digit2;
15    if (sign == '-')
16        ans = digit1 - digit2;
17    if (sign == '*')
18        ans = digit1 * digit2;
19
20    printf("%d\n", ans);
21    return 0;
22 }

```

Пример 7. Дана последовательность положительных целых чисел от -10^{100} до 10^{100} , разделённых между собой знаками $+$ или $-$. Ввод заканчивается символом $=$. Произвести арифметическое вычисление, игнорируя числа, по модулю превосходящие 10^6 .

Ввод	2019+2018-1000=
Вывод	3037

Числа будем собирать по цифрам. Это легко сделать, используя схему Горнера:

$$2019 = (((0 * 10 + 2) * 10 + 0) * 10 + 1) * 10 + 9.$$

Алгоритм схемы следующий:

1. обнуляем **ans**;
2. считываем символ **ch**, если **ch** — не цифра, **ans** является ответом, в противном случае:
 - (а) к предыдущему результату **ans** прибавляем текущую цифру;
 - (б) результат **ans** умножаем на 10;

(с) переходим к шагу 2.

Для корректного ввода установим такой порядок: определяем знак перед числом (по умолчанию +), собираем число из цифр, производим арифметическое действие.

```
1  #include <stdio.h>
2  int main()
3  {
4      int ans = 0, current, sign = '+', overflow;
5      char ch = getchar();
6      do
7      {
8          current = 0;
9          overflow = 0;
10         while ('0' <= ch && ch <= '9')
11         {
12             current = 10 * current + (ch - '0');
13             if (current > 1000000)
14                 overflow = 1;
15             ch = getchar();
16         }
17         if (overflow == 0)
18         {
19             if (sign == '+')
20                 ans += current;
21             if (sign == '-')
22                 ans -= current;
23         }
24         sign = ch;
25     } while (sign != '=');
26     printf("%d\n", ans);
27     return 0;
28 }
```


6.9 Задания для самостоятельной работы

1. В переменной `char ch`; записан код некоторого символа. Написать один условный оператор, который изменяет символ на строчную букву, если это заглавная буква, иначе — ничего не делает.

2. Дана команда

```
printf("\ntnt\tntn\n\n\\\\\\n");
```

Сколько видимых символов будет напечатано? Сколько строк это займет? Напишите с отступами напечатанный текст.

3. Выведите все символы ASCII-таблицы, коды которых расположены между блоками кодов заглавных и строчных букв, в следующем формате:

```
код:символ
код:символ
...
код:символ
```

Вывод перенаправьте средствами терминала в файл `ascii.txt`.

4. Чему равны выражения:

- а) `'0' / '0'`;
- б) `'0' / 0`;
- в) `0 / '0'`;
- г) `3 * '2' - 2 * '3' - '1'`;
- д) `'+' - '+'`?

Замечание: не используйте явные значения кодов ASCII-таблицы.

5. Приведенный ниже код должен вычислять в тексте количество слов, оканчивающихся на символ `'a'` (слово — непрерывная последовательность букв). Приведите два примера текста, для которых программа будет выдавать верный и неверный результат, соответственно. Тексты примеров должны заканчиваться точкой.

```
1 #include <stdio.h>
2 int main(){
3     int ans = 0;
```

```
4      char ch;
5      while ((ch = getchar()) != '.')
6          if (ch == 'a'){
7              ch = getchar();
8              if (ch != 'a')
9                  ans++;
10         }
11     printf("%d\n", ans);
12     return 0;
13 }
```

6.10 Практикум на ЭВМ

1. Дана последовательность символов с точкой в конце. Вывести все символы, кроме цифр.

Ввод	Start: 10 June 2017; finish: 10 September 2017.
Вывод	Start: June ; finish: September .

2. Дана последовательность символов с точкой в конце. Посчитать, сколько в этой последовательности заглавных и строчных букв.

Ввод	Start: 10 June 2017; finish: 10 September 2017.
Вывод	3 21

3. Дано целое число k от 1 до 26. Вывести все буквы латинского алфавита с номерами, кратными k (нумерация букв от начала алфавита начинается с единицы).

Ввод	2
Вывод	bdfhjlnprtvxz

4. Дана последовательность символов с точкой в конце. Заменить все заглавные буквы на строчные.

Ввод	Start: 10 June 2017; finish: 10 September 2017.
Вывод	start: 10 june 2017; finish: 10 september 2017.

5. На первой строке дано целое число n от 1 до 100. На второй строке дана последовательность из n координат шахматных полей. Вывести цвет каждого поля, если поле A1 чёрное.

Ввод	3 A2 B3 C3
Вывод	white white black

6. Вводится буква от 'a' до 'z' и положительное целое число n от 1 до 1000. Вывести символ, циклически сдвинутый на n позиций по алфавиту.

Ввод	a 4	y 262
Вывод	e	a

7. Вводится чётное количество чередующихся символов подряд без пробелов: цифра (от 0 до 9) и знак (+, − или =). Ввод заканчивается знаком =. Необходимо произвести арифметические действия в данном выражении.

Ввод	2+7=	1-2+3-4+5-6=
Вывод	9	-3

8. Дано четырехзначное шестнадцатиричное число. Перевести его в десятичное число.

Ввод	0012	0100
Вывод	18	256

9. В первой строке дана отсечка времени t в формате hh:mm:ss, во второй строке — целое число d от 1 до 86399. Необходимо вывести время спустя d секунд после момента времени t .

Ввод	00:01:02 3661	00:00:00 86399	23:59:59 2
Вывод	01:02:03	23:59:59	00:00:01

10. Дано целое число от 2 до 10^6 . На следующей строке дано целое число от 1 до 10^{10000} . С помощью схемы Горнера найти остаток при делении второго числа на первое.

Ввод	2019 12345678901234567890123456789
Вывод	240

7 Цикл for

Тема занятия посвящена ещё одному типу циклов: циклу `for`, который является компактной записью цикла `while`. Также будут рассмотрены операторы `break`, `continue` и «запятая».

7.1 Цикл for

Рассмотрим код, который выводит все двузначные числа по возрастанию. Решение с применением цикла `while`:

```
int i = 10;
while (i < 100)
{
    printf("%d ", i);
    i++;
}
```

Структурно в коде можно выделить 4 компоненты:

1. инициализация переменной (`i = 10`);
2. условие выполнения цикла (`i < 100`);
3. тело цикла (`printf("%d i");`);
4. оператор итерирования (`i++`).

Этот же код имеет альтернативную и более компактную версию записи с помощью оператора `for`:

```
int i;
for (i = 10; i < 100; i++)
    printf("%d ", i);
```

Обобщая, можно написать следующий формат преобразования цикла `while`:

```
init;
while (condition)
{
    action;
    iteration;
}
```

в цикл `for`:

```
for (init; condition; iteration)
    action;
```

В качестве инициализации (`init`) и оператора итерирования (`iteration`) может выступать любой оператор (или набор операторов). Рассмотрим несколько примеров с простыми вариантами условий (`condition`) выполнения цикла `for`:

1. Вывести все нечётные числа от 1 до n .

```
int i, n;
scanf("%d", &n);
for (i = 1; i <= n; i += 2)
    printf("%d ", i);
```

2. Вывести строчные буквы английского алфавита:

```
char ch;
for (ch = 'a'; ch <= 'z'; ch++)
    putchar(ch);
```

3. Вывести все натуральные числа, квадраты которых не превосходят целого числа n .

```
int i, n;
scanf("%d", &n);
for (i = 1; i * i <= n; i++)
    printf("%d ", i);
```

4. Для данного натурального числа n вывести все натуральные числа от 1 до n по убыванию.

```
int i, n;
scanf("%d", &n);
for (i = n; i > 0; i--)
    printf("%d ", i);
```

5. Для данного натурального числа n вывести $n!!$ (двойной факториал), где $n!! = n \cdot (n - 2) \cdot (n - 4) \dots$

```
int i, n, ans = 1;
scanf("%d", &n);
for (i = n; i > 0; i -= 2)
    ans *= i;
```

6. Найти сумму цифр натурального числа n .

```
int n, sum = 0;
scanf("%d", &n);
for (sum = 0; n > 0; n /= 10)
    sum += n % 10;
```

7. Посчитать количество делителей натурального числа n .

```
ans = 0;
scanf("%d", &n);
for (d = 1; d <= n; d++)
    if (n % d == 0)
        ans++;
```

7.2 Условие продолжения цикла

Условие (condition) может быть не простым оператором сравнения, а представлять собой результат логических операторов.

Пример 1. Даны 2 целых положительных числа a , b от 1 до 10^5 . Необходимо на первой строке распечатать все целые числа от 1 до $\min(a, b)$ включительно, а на второй строке — все целые числа от 1 до $\max(a, b)$ включительно.

Ввод	3 5	5 4
Вывод	1 2 3 1 2 3 4 5	1 2 3 4 1 2 3 4 5

Целое число i меньше минимума из a и b тогда и только тогда, когда оно меньше каждого из них, то есть

```
i <= a && i <= b
```

Аналогично, число i меньше максимума из a и b тогда и только тогда, когда оно меньше хотя бы одного из них, то есть

```
i <= a || i <= b
```

```
1  #include <stdio.h>
2  int main()
3  {
4      int a, b, i;
5      scanf("%d %d", &a, &b);
6      for (i = 1; i <= a && i <= b; i++)
7          printf("%d ", i);
8      putchar('\n');
9      for (i = 1; i <= a || i <= b; i++)
10         printf("%d ", i);
11     putchar('\n');
12     return 0;
13 }
```

7.3 Переменные счётчики

Рассмотрим код с циклом `for`. Ответим на два вопроса. Сколько раз будет выведено слово «Hello»? Чему равно значение переменной `i` после завершения цикла?

```
int i;
for (i = 10; i < 25; i++)
    puts("Hello");
printf("%d", i);
```

Здесь `i` принимает целые значения из $[10; 25)$. Количество таких элементов можно вычислить как разность границ: $25 - 10 = 15$. Таким образом, слово «Hello» будет выведено 15 раз. А значение `i` после завершения цикла — 25.

Значение переменных счётчиков могут быть использованы после завершения цикла. В случае естественного завершения цикла значение счётчика определяется из условия в цикле. Например:

```
char ch;
for (ch = 'a'; ch <= 'f'; ch++)
    putchar(ch);
char next = ch;
```


После выполнения кода значение переменной `ch` будет таким, что неравенство `ch <= 'f'` не выполняется, то есть в переменной `ch` будет значение `'g'`.

Не стоит забывать, что переменную счётчик можно изменять внутри цикла. Начинающие программисты чаще делают такие действия неосознанно. За этим бывает очень сложно уследить, поэтому данный приём следует применять крайне аккуратно. Например:

```
int x, y = 0;
for (x = 3; x < 7; x++)
{
    printf("%d %d\n", x, y);
    y++;
    x += 2;
}
```

Чтобы проследить за происходящим, перепишем код через `while`:

```
int x, y = 0;
x = 3;
while (x < 7)
{
    printf("%d %d\n", x, y);
    y++;
    x += 2;
    x++;
}
```

Вывод:

```
$ ./prog
3 0
6 1
```

7.4 Вложенные циклы

Выведем таблицу умножения размером 5×5 с помощью цикла `for`:

```
1  2  3  4  5
2  4  6  8 10
3  6  9 12 15
4  8 12 16 20
5 10 15 20 25
```

Вывести таблицу умножения можно двумя разными способами: с помощью одинарного цикла и с помощью вложенных циклов. Сравним эти два подхода.

Первый способ. Используем переменную счётчик i , которая изменяется от 0 до 24. Выразим через неё номер строки (row) и номер столбца ($column$):

$$\begin{cases} row = \left[\frac{i}{5} \right] + 1 \\ column = i \bmod 5 + 1 \end{cases}$$

Например:

i	row	$column$
0	1	1
1	1	2
2	1	3
3	1	4
4	1	5
5	2	1
6	2	2
..
23	5	4
24	5	5

Тогда i -е по счету выводимое значение равно $row * column$. Переносы строк можно выводить после последнего элемента в строке, то есть как только $column$ равно 5.

```
int i, row, column;
for (i = 0; i < 25; i++)
{
    row = i / 5 + 1;
    column = i % 5 + 1;
    printf("%3d", row * column);
    if (column == 5)
        putchar('\n');
}
```

Второй способ с использованием вложенных циклов является более компактным. Для этого необходимо завести две переменные типа счётчик: row — для строк, $column$ — для столбцов.

Внешний цикл будет перебирать строки row . Для фиксированной строки row будем перебирать столбцы $column$ вложенным циклом.

1. Сначала распечатаем всю первую строку $row = 1$:
 - (a) выводим значение при $column = 1$,
 - (b) выводим значение при $column = 2$,
 - (c) выводим значение при $column = 3$,
 - (d) выводим значение при $column = 4$,
 - (e) выводим значение при $column = 5$,
 - (f) для перехода на новую строку печатаем перенос строки.
2. Далее распечатаем всю вторую строку $row = 2$:
 - (a) выводим значение при $column = 1$,
 - (b) выводим значение при $column = 2$,
 - (c) выводим значение при $column = 3$,
 - (d) выводим значение при $column = 4$,
 - (e) выводим значение при $column = 5$,
 - (f) для перехода на новую строку печатаем перенос строки.
3. и так далее.

Заметим, что для решения задачи потребуется вложить цикл, перебирающий столбцы, в цикл, перебирающий строки.

```
int row, column;
for (row = 1; row <= 5; row++)
{
    for (column = 1; column <= 5; column++)
        printf("%3d", row * column);
    putchar('\n');
}
```

Вывод: в рассмотренном примере решение с вложенным циклом оказалось более оптимальным.

Рассмотрим пример, который демонстрирует обратную ситуацию: решение задачи с помощью одинарного цикла более предпочтительно, чем решение с вложенным циклом.

Пример 2. Дано вещественное число x и натуральное число n . Вычислить частичную сумму ряда экспоненты e^x :

$$1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!}$$

Ответ вывести с 3 знаками после запятой.

Ввод	1.0 2	1.0 10
Вывод	2.500	2.718

Одно из возможных решений с вложенным циклом заключается в следующем: для вычисления частичной суммы ряда используем внешний цикл, а для вычисления значения отдельного слагаемого — внутренний цикл.

1. Сначала вычислим первое слагаемое:

- (a) числитель равен x ,
- (a) знаменатель равен 1,

2. Далее вычислим второе слагаемое:

- (a) числитель равен x ,
- (b) числитель равен x^2 .
- (a) знаменатель равен 1,
- (b) знаменатель равен $1 \cdot 2$,

3. Прибавим его к сумме.

4. Далее вычислим третье слагаемое:

- (a) числитель равен x ,
- (b) числитель равен x^2 ,
- (c) числитель равен x^3 ,
- (a) знаменатель равен 1,
- (b) знаменатель равен $1 \cdot 2$,
- (c) знаменатель равен $1 \cdot 2 \cdot 3$,

5. Прибавим его к сумме.

6. и так далее.

```
1  #include <stdio.h>
2
3  int main() {
4      int i, j, n;
5      double sum = 1, add;
```

```

6     scanf("%f %d", &x, &n);
7     for (i = 1; i <= n; i++)
8     {
9         add = 1;
10        for (j = 1; j <= i; j++)
11            add *= x;
12        for (j = 1; j <= i; j++)
13            add /= j;
14        sum += add;
15    }
16    printf("%.3lf", sum);
17    return 0;
18 }

```

Таким образом решена задача с помощью вложенного цикла. Заметим, что неоптимально вычислять степень x^i на каждом шаге во вложенном цикле: можно использовать результат с предыдущего шага $x^i = x^{i-1} \cdot x$ и обойтись без вложенного цикла. Аналогично с факториалом. Ниже приведен код с одинарным циклом:

```

1  #include <stdio.h>
2
3  int main() {
4      int i, j, n;
5      double sum = 1.0, add = 1.0;
6      scanf("%f %d", &x, &n);
7      for (i = 1; i <= n; i++)
8      {
9          add *= x;
10         add /= j;
11         sum += add;
12     }
13     printf("%.3lf", sum);
14     return 0;
15 }

```

Вывод: вложенных циклов в ряде задач можно избежать при использовании данных с предыдущих итераций.

7.5 Перебор всех пар

Пример 3. Дано целое число n от 1 до 100. Вывести все сочетания неупо-

рядоченных пар различных чисел от 1 до n . То есть пару (a, b) вывести до (c, d) в одном из двух случаев: либо $a < c$, либо $a = c, b < d$.

Ввод	5
Вывод	(1; 2) (1; 3) (1; 4) (1; 5) (2; 3) (2; 4) (2; 5) (3; 4) (3; 5) (4; 5)

Сначала необходимо вывести все пары $(1; j)$, где $j \geq 2$. Потом все пары $(2; j)$, где $j \geq 3$ и так далее. В этом случае границы вложенного цикла зависят от переменной-счётчика внешнего цикла.

```
#include <stdio.h>

int main() {}
    int i, j, n;
    scanf("%d", &n);
    for (i = 1; i < n; i++)
        for (j = i + 1; j <= n; j++)
            printf("(%d; %d)\n", i, j);
    return 0;
}
```

7.6 Проблема общего счётчика во вложенных циклах

При использовании и модификации одной и той же переменной во вложенных циклах может возникнуть проблема: внутренний цикл искажит значение счётчика внешнего цикла. Например:

```
int i;
for (i = 0; i < 5; i++)
    for (i = 0; i < 5; i++)
        printf("%d", i);
```

Определим, как будет меняться переменная i . Для этого перепишем код через два цикла `while`:

```

1  int i;
2  i = 0;
3  while (i < 5)
4  {
5      i = 0;
6      while (i < 5)
7      {
8          printf("%d", i);
9          i++;
10     }
11     i++;
12 }

```

На первом шаге внешнего цикла $i = 0$. Внутренний цикл выведет числа 0 1 2 3 4 и остановится при $i = 5$. Далее оператор инкремента внешнего цикла на 11 строке увеличит i на 1 и получится, что $i = 6$. Условие $i < 5$ на 3 строке не выполнится и внешний цикл завершится, выполнив всего одну итерацию вместо ожидаемых пяти.

7.7 Оператор «запятая»

Инициализация и итерирование цикла может выполняться в несколько действий. Разделить такие действия точкой с запятой не получится, так как точка с запятой является внутренней конструкцией оператора `for`. Для этого можно применить оператор «запятая» в качестве разделителя.

Рассмотрим пример обмена значениями двух переменных с разделителем в виде точки с запятой:

```

int a = 1, b = 2, c = 3;
c = a;
a = b;
b = c;

```

и в виде запятой:

```

int a = 1, b = 2, c = 3;
c = a, a = b, b = c;

```

Стоит помнить, что три оператора присваивания, записанные в одну строку, читаются хуже, чем три оператора, записанные в три строки.

Мы будем использовать оператор «запятая» внутри цикла `for`. Например, вывести максимальное число Фибоначчи, которое не превосходит n .

```
int f0, f1, f2, n;
scanf("%d", &n);
for (f0 = 0, f1 = 1; f1 < n; f0 = f1, f1 = f2)
    f2 = f0 + f1;
printf("%d\n", f0);
```

В языках программирования (как и в естественных языках) следует аккуратно относиться к пробелам.

Хороший стиль оформления кода. Пробелы ставятся только после знаков препинания: запятой и точки с запятой.

Например:

```
for (a = 0, b = 5; a < b; a++, b--)
    ...;
```

— хорошо;

```
for (a = 0,b = 5 ; a < b ;a++ , b-- )
    ...;
```

— плохо.

Не рекомендуется в инициализации и завершении цикла использовать функции. Например, данный код считывает и выводит символы, пока не встретит точку:

```
for (c = getchar(); c != '.'; c = getchar())
    putchar(c);
```

Но он является плохо читаемым, поэтому стоит избегать такого использования цикла for.

7.8 Условный оператор и условие продолжение цикла

Всегда необходимо чётко различать условие в условном операторе внутри цикла и условие продолжения цикла (condition).

Например, необходимо распечатать все числа от 1 до 10, кроме 7. Неправильно объединять эти оба условия в условие цикла:

```
int i;
for (i = 1; i <= 10 && i != 7; i++)
    printf("%d ", i);
```


В этом случае условие цикла будет выполняться до тех пор, пока числа не больше 10 и не равны 7. Соответственно, для 7 это условие ложно и на экране будет выведено:

```
1 2 3 4 5 6
```

Верным решением будет:

```
int i;
for (i = 1; i <= 10; i++)
    if (i != 7)
        printf("%d ", i);
```

Тогда на экране появится:

```
1 2 3 4 5 6 8 9 10
```

В следующем примере необходимо вывести все буквы (и заглавные, и строчные):

```
char ch;
for (ch = 'A'; ch <= 'z'; ch++)
    if (ch <= 'Z' || ch >= 'a')
        putchar(ch);
```

7.9 Бесконечный цикл, break, continue

Стоит отметить, что каждая из трёх компонент цикла

```
for (init; condition; iteration)
```

не является обязательной.

Например, отсутствие инициализации (init) и итерирования (iteration):

```
for ( ; condition; )
    action;
```

соответствует простому циклу **while**.

Отсутствие условия выполнения цикла:

```
for ( ; ; )
    action;
```

приводит к бесконечному циклу. Для цикла **while** отсутствие условия недопустимо, а бесконечный цикл запускается следующим образом:

```
while (1)
    action;
```

Для досрочного завершения цикла (в том числе и `while`, и `do while`) есть специальный оператор `break`. Он прерывает текущий цикл и переходит к следующему за циклом оператору. Например:

```
int i;
for (i = 0; i < 5; i++)
{
    printf("start d\n", i);
    if (i == 2)
        break;
    printf("finish %d\n", i);
}
printf("last index is %d\n", i);
```

На выходе будет:

```
$ ./prog
start 0
finish 0
start 1
finish 1
start 2
last index is 2
```

Также имеется возможность пропустить оставшуюся содержательную часть тела цикла (аналогично у `while`, и `do while`). Для этого есть оператор `continue`. Он прерывает тело цикла, выполняет итерирование и переходит к следующему шагу. Например:

```
int i;
for (i = 0; i < 5; i++)
{
    printf("start d\n", i);
    if (i == 2)
        continue;
    printf("finish %d\n", i);
}
printf("last index is %d\n", i);
```

На выходе будет:

```
$ ./prog
start 0
finish 0
start 1
finish 1
start 2
start 3
finish 3
start 4
finish 4
last index is 5
```

Операторы **break** и **continue** влияют только на тот цикл, в котором они находятся непосредственно. Например, такой код:

```
int i, j;
for (i = 0; i < 5; i++)
{
    for (j = 0; j < 5; j++)
        if (j == 2)
            break;
    printf("%d ", i);
}
```

выведет 0 1 2 3 4.

7.10 Примеры

Пример 4. В первой строке дано целое число n от 1 до 1000. Во второй строке дана последовательность из n целых чисел от -10^6 до 10^6 . Найти сумму чисел последовательности.

Ввод	3 3 -4 2	1 5
Вывод	1	5

```
1 #include <stdio.h>
2 int main()
3 {
4     int n, i, x, sum = 0;
5     scanf("%d", &n);
```

```

6     for (i = 0; i < n; i++)
7     {
8         scanf("%d", &x);
9         sum += x;
10    }
11    printf("%d\n", sum);
12    return 0;
13 }
```

Пример 5. В первой строке дано целое число n от 1 до 1000. Во второй строке дана последовательность из n целых чисел от -10^6 до 10^6 . Найти минимальное значение в последовательности.

Ввод	3	1
	3 -4 2	5
Вывод	-4	5

```

1  #include <stdio.h>
2  int main()
3  {
4      int n, i, x, min;
5      scanf("%d", &n);
6      for (i = 0; i < n; i++)
7      {
8          scanf("%d", &x);
9          if (i == 0 || min > x)
10             min = x;
11      }
12      printf("%d\n", min);
13      return 0;
14 }
```

Проследим за переменной `min`, которая изначально ничем не инициализируется (значение является мусорным). Но при этом её значение используется в условном операторе:

```
if (i == 0 || min > x)
```

Не возникает ли здесь проблема? Заметим, что на первом шаге сравнение будет таким: `if (0 == 0 || ? > 3)`. Первый аргумент `0 == 0` оператора ИЛИ является истинным, то есть результат оператора уже точно известен — тоже «истина». Так как язык C поддерживает «ленивые» вычисления,

то второй аргумент $? > 3$ не вычисляется вообще. Следовательно, всё корректно. Но переставить операнды местами нельзя:

```
if (i == 0 || min > x)
```

Здесь будут вычисления с мусорным значением.

Пример 6. В первой строке дано целое число n от 1 до 1000. Во второй строке дана последовательность из n целых чисел от -10^6 до 10^6 . Найти среднее арифметическое $\frac{\sum_{i=1}^n a_i}{n}$ и квадратическое $\sqrt{\frac{\sum_{i=1}^n a_i^2}{n}}$.

Ввод	5 1 2 3 4 5
Вывод	3.00 3.32

Результаты могут быть нецелыми, поэтому все вычисления будем производить в вещественной арифметике. Сумму чисел и сумму их квадратов можно вычислить в процессе ввода.

```
1  #include <stdio.h>
2  #include <math.h>
3  int main()
4  {
5      int n, i, x;
6      double sum = 0, sum2 = 0;
7      scanf("%d", &n);
8      for (i = 0; i < n; i++)
9      {
10         scanf("%d", &x);
11         sum += x;
12         sum2 += x * x;
13     }
14     sum /= n;
15     sum2 = sqrt(sum2 / n);
16     printf("%.2f %.2f", sum, sum2);
17     return 0;
18 }
```

Пример 7. В первой строке дано целое число n от 1 до 1000. Во второй строке дана последовательность из n целых чисел от -10^6 до 10^6 . Найти длину наибольшей возрастающей подстроки. Подстрокой последовательности называется любая подпоследовательность подряд идущих элементов.

Ввод	9 15 13 14 12 14 16 17 1 2	3 2 3 1
Вывод	4	2

На первый взгляд не очень простая задача. Разделим её на две более простых: найти длины возрастающих подстрок и найти максимум из них. Первая задача решается так:

```

1  #include <stdio.h>
2  int main()
3  {
4      int n, i, length = 1;
5      int current, previous;
6
7      scanf("%d", &n);
8      scanf("%d", &current);
9      for (i = 2; i <= n; i++)
10     {
11         previous = current;
12         scanf("%d", &current);
13
14         if (previous < current)
15             length++;
16         else
17             {
18                 printf("%d ", len);
19                 length = 1;
20             }
21     }
22     printf("%d\n", length);
23     return 0;
24 }
```

Обратите внимание, что первое число последовательности считывается вне цикла. Переход к следующему числу реализуется в два действия: записываем текущее значение в переменную `previous`, считываем новое значение в переменную `current`. Печать внутри цикла производится только в том случае, когда нашлась пара чисел, расположенных по убыванию (в примере выше это: (15, 13), (14, 12) и (17, 1)). То есть для последней подстроки данный вывод не будет выполняться. Поэтому её длину выводим вне цикла в конце.

Осталось найти максимум из значений `length`. Первый вариант: добавим условный оператор `if (max_length < length)` на 18 и 22 строках,

где находится вывод длины строки. Второй вариант: поставить условный оператор сразу же после операции `length++`. Первый вариант будет в среднем немного эффективней с точки зрения скорости работы, а второй — короче с точки зрения кода.

```
1  #include <stdio.h>
2  int main()
3  {
4      int n, i, length = 1, max_length = 1;
5      int current, previous;
6
7      scanf("%d", &n);
8      scanf("%d", &current);
9      for (i = 2; i <= n; i++)
10     {
11         previous = current;
12         scanf("%d", &current);
13
14         if (previous < current) {
15             length++;
16             if (max_length < length)
17                 max_length = length;
18         }
19         else
20             length = 1;
21     }
22     printf("%d\n", max_length);
23     return 0;
24 }
```

Пример 8. Дана позиция ферзя на шахматной доске. Распечатать шахматную доску, отметив все клетки которые бьёт ферзь 'X', а остальные — символом '.'. Между символами выводить пробел.

Ввод	F3
Вывод	<pre> X X . . . X . . . X X . . X X . X . X X X X . X X X X X X X X X X X X . X . X </pre>

Если доску печатать со стороны белых, то необходимо будет строки нумеровать в обратном порядке от 8 до 1, а столбцы — в прямом от A до H.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      char r, c;
6      char row, col;
7      col = getchar();
8      row = getchar();
9      for (r = '8'; r >= '1'; r--)
10     {
11         for (c = 'A'; c <= 'H'; c++)
12             if (abs(r - row) == abs(c - col) ||
13                 r == row || c == col)
14                 printf("X ");
15             else
16                 printf(". ");
17         putchar('\n');
18     }
19     return 0;
20 }
```

Пример 9. Дано целое положительное число n от 1 до 10^9 . Разложить n на простые множители с учётом кратности.

Ввод	30	12	13
Вывод	2 3 5	2 2 3	13

За основу возьмем алгоритм получения всех делителей числа n . Для этого переберём d от 2 до n .

```
for (d = 2; d <= n; d++)
    if (n % d == 0)
        printf("%d ", d);
```

Чтобы не учитывать составные делители, необходимо исходное число делить на найденные делители:

```
for (d = 2; d <= n; d++)
    if (n % d == 0)
    {
        printf("%d ", d);
        n /= d;
    }
```

Теперь изменяются обе части сравнения $d \leq n$ в условии выполнения цикла. Причём d растёт, а n уменьшается. Посмотрим, что будет выведено на примерах. При $n = 30$ вывод: 2 3 5 — верно. А вот при $n = 12$ вывод: 2 3 — неверно. Проблема заключается в том, что к делителю равному 3 необходимо переходить только в том случае, если число больше не делится на 2. А оно может делиться на 2 несколько раз. Поэтому меняем `if` на `while`:

```
1  #include <stdio.h>
2
3  int main() {
4      int d, n;
5      scanf("%d", &n);
6      for (d = 2; d <= n; d++)
7          while (n % d == 0)
8              {
9                  printf("%d ", d);
10                 n /= d;
11             }
12     return 0;
13 }
```

7.11 Задания для самостоятельной работы

1. Что будет выведено на печать при выполнении данного кода?

```
int i;
for (i = 30; i > 1; i /= 3)
    printf("%d ", i);
```

2. Что будет выведено на печать при выполнении данного кода? Чем равны значения переменных x и y после завершения внешнего цикла?

```
int x, y;
for (x = 1; x < 5; x++)
{
    for (y = 5; y >= 3; y--)
        printf("%d ", x + y)
    printf("\n");
}
```

3. Выберите циклы, которые выводят числа: 0 2 4 6 8. Переменная x имеет тип `int`.

- a)

```
for (x = 0; x <= 10; x += 2)
    printf("%d ");
```
- б)

```
for (x = 0; x != 10; x += 2)
    printf("%d ");
```
- в)

```
for (x = 0; x % 2 == 0 && x < 9; x++)
    printf("%d ");
```
- г)

```
for (x = 0; x < 9; x++)
    if (x % 2 == 0)
        printf("%d ");
```
- д)

```
for (x = 0; x < 20; x += 2)
    if (x == 10)
        break;
    else
        printf("%d ");
```
- е)

```
for (x = 0; x < 20; x += 2)
    if (x == 10)
        continue;
    else
        printf("%d ");
```

4. На первой строке дано целое положительное число n от 1 до 1000. На второй строке дана последовательность из n целых чисел от 1 до 1000. Найти количество чётных чисел в последовательности. Дополните код.

```
#include <stdio.h>
int main() {
    int n, i, a, answer = 0;
    scanf("%d", &n);

    printf("%d\n", answer);
    return 0;
}
```

5. Дано целое положительное число n . Распечатать шахматную доску размера $n \times n$, где чёрные клетки обозначены 'X', а белые '0' (A1 — черная). Дополните код.

```
#include <stdio.h>
int main(){
    char row, column;

    return 0;
}
```

7.12 Практикум на ЭВМ

1. Дано целое число n от 1 до 20. Напечатать $n!$.

Ввод	5	20
Вывод	120	2432902008176640000

2. Дано натуральное число n от 1 до 10^9 . Посчитать количество делителей и сумму делителей числа n .

Ввод	28	101	100
Вывод	6 56	2 102	9 217

3. Дано целое число n от 1 до 10^9 . Разложить n на простые множители и вывести их.

Ввод	12	101	2020
Вывод	2 2 3	101	2 2 5 101

4. На первой строке дано целое число n от 1 до 1000. На второй строке дана последовательность из n целых чисел от -1000 до 1000 . Найти и вывести сумму всех положительных и произведение всех отрицательных чисел последовательности. Гарантируется, что ответы по модулю не превосходят 10^{18} .

Ввод	6 1 -2 5 -4 -3 0	3 0 0 0	3 -1 0 1
Вывод	6 -24	0 1	1 -1

5. На первой строке дано целое число n от 1 до 1000. На второй строке дана последовательность из n целых чисел от -1000 до 1000 . Найти и вывести минимальный элемент последовательности и его позицию. Если минимальных чисел несколько, вывести позицию первого минимального элемента.

Ввод	5 4 8 2 7 3	5 -2 -1 0 1 2 3	3 -3 -3 -3
Вывод	2 3	-2 1	-3 1

6. На первой строке дано целое число n от 1 до 1000. На второй строке дана последовательность из n целых чисел от -1000 до 1000 . Найти и вывести максимальный элемент последовательности и количество элементов, равных максимальному.

Ввод	6 4 7 2 7 7 4	3 -2 -3 -2	3 1 3 2
Вывод	7 3	-2 2	3 1

7. На первой строке дано целое число n от 2 до 1000. На второй строке дана последовательность из n целых чисел от 1 до 1000. Найти и вывести среднее геометрическое

$$\sqrt[n]{a_1 \cdot a_2 \cdot \dots \cdot a_n}$$

и гармоническое

$$\frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \dots + \frac{1}{a_n}}$$

всех элементов последовательности.

Ввод	4 1 2 2 4	4 2 2 2 2	2 4 6
Вывод	2.00 1.78	2.00 2.00	4.90 4.80

8. Дано целое положительное число n от 1 до 100. Вывести матрицу размера $n \times n$, где $a_{ij} = |i - j|$.

Ввод	3	1	2
Вывод	0 1 2 1 0 1 2 1 0	0	0 1 1 0

9. Даны целые положительные числа a и b ($a \leq b$). Найти сумму цифр у каждого числа от a до b включительно.

Ввод	98 102	549 551	17 17
Вывод	17 18 1 2 3	18 10 11	8

10. На первой строке дано целое число n от 1 до 1000. На второй строке дана последовательность из n целых чисел a_i от -1000 до 1000 . Найти максимальную сумму среди всех подотрезков $a_k + a_{k+1} + \dots + a_{k+m}$. Идея решения: http://e-maxx.ru/algo/maximum_average_segment.

Ввод	5 2 -3 4 5 -1	5 2 3 4 5 1	6 -2 1 2 -1 3 -4
Вывод	9	15	5

8 Массивы

В данной теме будет подробно рассмотрены одномерные и многомерные статические массивы. А также впервые будет затронут вопрос о самой главной отличительной особенности языка С от многих других языков — об указателях.

8.1 Одномерный массив

Программы для задач с обработкой последовательности чисел, которые были рассмотрены ранее, использовали обработку чисел «на лету» (англ. online algorithm). Некоторые задачи таким образом решить невозможно. Например: найти наиболее часто встречающееся число, посчитать количество чисел равных последнему, вывести числа в обратном порядке. Такие задачи можно решить с помощью массивов.

Массив — это упорядоченный набор однотипных элементов, к которым можно обращаться по номерам (индексам). Объявить массив можно следующим образом:

```
int a[5];
```

После такого объявления в памяти выделяется блок из 20 байт (5 чисел по 4 байта), к которому можно обратиться по имени **a**. В языке С нумерация начинается с нуля. Чтобы работать с отдельным элементом массива, достаточно указать индекс в квадратных скобках: **a[0]** — первый элемент, **a[1]** — второй элемент, ..., **a[4]** — пятый (последний) элемент.

После объявления массива, как и в случае с обычной переменной, в элементах лежат мусорные значения. Перед началом работы с массивом следует присвоить переменным некоторые значения.

```
a[0] = 20;  
a[1] = 18;  
a[2] = 0;  
a[3] = a[0];  
a[4] = a[1] + 1;
```

После такой инициализации массив в памяти будет представлен следующим образом:

Имя	a[0]	a[1]	a[2]	a[3]	a[4]
Значение	20	18	0	20	19

Массив можно инициализировать непосредственно при объявлении одним из двух способов. Первый — с явным указанием размера:

```
int a[5] = {20, 18, 0, 20, 19};
```

Второй — без явного указания размера:

```
int a[] = {20, 18, 0, 20, 19};
```

Размер будет вычислен автоматически по количеству элементов.

Также имеется возможность инициализировать массивы частично:

```
int a[5] = {20, 18};
```

что приводит к заполнению массива числами:

Имя	a[0]	a[1]	a[2]	a[3]	a[4]
Значение	20	18	0	0	0

Таким приёмом можно легко обнулить весь массив при инициализации:

```
int a[5] = {0};
```

8.2 Применение массивов

Для последовательного доступа к элементам массива используют циклы. Например, следующим образом можно считать 5 целых чисел и распечатать их в обратном порядке:

```
1  #include <stdio.h>
2
3  int main() {
4      int i, a[5];
5      for (i = 0; i < 5; i++)
6          scanf("%d", &a[i]);
7      for (i = 4; i >= 0; i--)
8          printf("%d ", a[i]);
9      return 0
10 }
```

Массивы позволяют разделить программу на 3 логические части: ввод данных, обработка, вывод ответа. Например, дано целое число n от 1 до 100 и последовательность из n целых чисел. Необходимо найти сумму чисел последовательности.

```
1  #include <stdio.h>
2
```

```
3  int main() {
4      int n, i, sum = 0, a[100];
5      scanf("%d", &n);
6      for (i = 0; i < n; i++)
7          scanf("%d", &a[i]);
8
9      for (i = 0; i < n; i++)
10         sum += a[i];
11
12     printf("%d\n", sum);
13     return 0;
14 }
```

Ввод данных осуществляется на строках 5–7, обработка – на строках 9–10, вывод – на 12-й строке.

Массивы необходимы для случаев, когда данные невозможно обработать в порядке ввода. Например, дано целое число n от 1 до 100 и последовательность из n вещественных чисел. Необходимо вывести те числа, которые больше последнего числа последовательности.

```
#include <stdio.h>

int main() {
    int n, i;
    double a[100];

    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%f", &a[i]);

    for (i = 0; i < n; i++)
        if (a[i] > a[n-1])
            printf("%d ", a[i]);
    reutrn 0;
}
```

8.3 Размер массива в задачах

Массивы можно использовать даже в том случае, когда неизвестно точное количество элементов последовательности, но есть ограничение на максимальный размер последовательности. Например, если дано не

более 100 целых чисел, то подразумевается, что можно выделить массив сразу на 100 элементов:

```
int a[100];
```

Можно использовать константную переменную для размера:

```
const int nmax = 100;
int n;
int a[nmax];
scanf("%d", &n);
...
```

Правда здесь отсутствует возможность инициализировать массив.

В целях экономии можно выделять память под массив размером n . Ниже представлена типичная ошибка при попытке объявить такой массив:

```
int n;
int a[n];
scanf("%d", &n);
...
```

В данном случае размер массива будет определён случайным образом, потому что в переменной n содержится мусорное значение. Это может привести к неопределённому поведению программы (англ. *undefined behaviour*), то есть ошибка проявляется от случая к случаю. Иногда программа даже может вывести верный ответ, если n окажется больше размера массива. Это является случайностью.

Правильным вариантом будет выделение массива на n элементов следующим образом:

```
int n;
scanf("%d", &n);
int a[n];
```

Обратите внимание, что сначала инициализируется n , а только потом выделяется память. В рамках изучаемого курса рекомендуется не использовать такое выделение памяти, а использовать настоящее динамическое выделение памяти, о котором будет подробно рассказано в следующих темах.

8.4 Размеры переменных

Прежде чем осветить очень популярную ошибку (выход за пределы массива), поговорим немного о размерах переменных.

Определить количество байт, которое занимает переменная данного типа, можно при помощи оператора `sizeof`:

```
int s1 = sizeof(char);
int s2 = sizeof(double);
```

В переменной s_1 окажется 1, а в переменной s_2 — 8.

Аналогично можно узнать размер массива:

```
int s3 = sizeof(int[5]);
```

В переменной s_3 будет 20.

Также можно определить размер любой переменной:

```
char ch;
double x;
int a[5];
int s1 = sizeof(ch);
int s2 = sizeof(x);
int s3 = sizeof(a);
```

8.5 Адреса

Когда в программе объявляются переменные, операционная система выделяет для них память. Например, при таком коде:

```
char ch = 'z';
short n = 7;
int x = 123, y = 456;
```

память может быть распределена может быть распределена следующим образом (вместо полного адреса выписаны только последние две цифры адреса, в предположении, что остальные цифры совпадают):

Адрес	..41	..42	..43	..44	..45	..46	..47	..48	..49	..50	..51
Имя	ch	n			x				y		
Значение	'z'	7			123				456		

Обратите внимание, что ячейки для переменных расположены последовательно друг за другом. Адрес первой из них (..41) может меняться при

запусках программы. Размер каждой ячейки равен одному байту. Адрес следующей переменной можно определить, если к адресу текущей переменной прибавить её же размер.

Для того чтобы получить адрес, достаточно использовать унарный левосторонний оператор `&` (амперсанд) перед именем переменной. Например `&ch` — адрес переменной `ch`. С этим амперсандом мы уже знакомы из функции `scanf`. Итак, посмотрим на адреса переменных:

```
long ch_ptr = (long)&ch;
long n_ptr = (long)&n;
long x_ptr = (long)&x;
long y_ptr = (long)&y;
printf("%ld\n", ch_ptr);
printf("%ld\n", n_ptr);
printf("%ld\n", x_ptr);
printf("%ld\n", y_ptr);
```

Для печати в десятичном виде приведем адреса к типу `long int`. При запуске программы на выводе будет следующее:

```
$ ./prog
140723082296941
140723082296942
140723082296944
140723082296948
```

Разумеется, цифры могут различаться, но смещения адресов друг относительно друга будут аналогичными. Адреса могут меняться при перезапусках программы, так как операционная система может выделять разные блоки памяти при каждом новом запуске.

Теперь посмотрим, как можно работать с адресами массива:

```
int a[5] = {20, 18, 0, 20, 19};
```

Расположение в памяти выглядит следующим образом:

Адрес	..00	..04	..08	..12	..16
Имя	a[0]	a[1]	a[2]	a[3]	a[4]
Значение	20	18	0	20	19

Выведем адреса массива и некоторых элементов массива:

```
long a_ptr = (long)&a;
long a0_ptr = (long)&a[0];
long a1_ptr = (long)&a[1];
```

```
long a3_ptr = (long)&a[3];
printf("%ld\n", a_ptr);
printf("%ld\n", a0_ptr);
printf("%ld\n", a1_ptr);
printf("%ld\n", a3_ptr);
```

Оказывается, адрес самого массива — это адрес его первого элемента!

```
140727744864000
140727744864000
140727744864004
140727744864012
```

Теперь можно ответить на вопрос, как работает индексация массива: `a[i]` означает, что от адреса `a` необходимо отступить `i` шагов по 4 байта (размер шага определяется типом элемента массива).

8.6 Копирование и сравнение

Копировать массивы целиком оператором присваивания нельзя:

```
int a[3] = {10, 11, 12};
int b[3];
b = a;
```

Код выше приведет к ошибке:

```
error: invalid array assignment
    b = a;
```

Копировать массивы нужно поэлементно:

```
int a[3] = {10, 11, 12};
int b[3];
int i;
for (i = 0; i < 3; i++)
    b[i] = a[i];
```

Сравнивать массивы целиком тоже нельзя. Если мы сравним два массива `a` и `b` с одинаковыми элементами следующим образом:

```
int a[3] = {10, 11, 12};
int b[3] = {10, 11, 12};
if (b == a)
    puts("equal");
```

```
else
    puts("not equal");
```

то на экран будет выведено не то, что ожидает автор программы:

```
$ ./prog
not equal
```

Связано это с тем, что вместо **a** и **b** подставляются адреса начала массивов, а не элементы. А адреса, в отличие от элементов массивов, различаются.

8.7 Ваш первый segmentation fault

Обращение к несуществующим ячейкам памяти может привести к неопределенному поведению программы (англ. *undefined behavior*):

1. к ошибке сегментации данных (англ. *segmentation fault*);
2. к искажению значений других переменных;
3. к ожидаемому поведению программы.

Рассмотрим код, приводящий к ошибке сегментации данных:

```
1  #include <stdio.h>
2  int main() {
3      int a[16];
4      a[1000000] = 10;
5      printf("a[1000000] = %d", a[1000000]);
6      return 0;
7  }
```

Программа аварийно завершится во время выполнения:

```
$ gcc prog.c -o prog
$ ./prog
segmentation fault (core dumped)
$
```

В терминале появится сообщение: «ошибка сегментирования (стек памяти сброшен на диск)». Для массива `int a[16]` мы запросили 64 байта, которые образуют сегмент данных программы из 64 подряд идущих ячеек памяти. При попытке изменить элемент `a[1000000] = 10`, произойдет

обращение к ячейке памяти, которая выходит за пределы сегмента данных (её адрес сдвинут на $4 \cdot 1000000$ байт относительно реального начала массива). Это и приводит к «ошибке сегментирования».

Теперь рассмотрим код, приводящий к неявному изменению других переменных программы:

```
1  #include <stdio.h>
2  int main() {
3      int a[16];
4      int b[16];
5      a[20] = 10;
6      printf("b[4] = %d\n", b[4]);
7      return 0;
8  }
```

В этом случае ошибки сегментирования нет, но изменяется значение элемента массива `b`:

```
$ gcc prog.c -o prog
$ ./prog
b[4] = 10
$
```

Обращение `a[20]` задаёт смещение на $4 \cdot 20 = 80$ байт относительно начала массива. Если считать, что массивы расположены друг за другом в памяти, то обращение происходит к ячейке со смещением $+16$ относительно начала второго массива `b`. То есть обращение к элементу `b[4]`.

Заключительный код, который приводит к неопределённому поведению программы:

```
1  #include <stdio.h>
2  int main() {
3      int a[16];
4      a[2050] = 10;
5      print("a[2050] = %d\n", a[2050]);
6      return 0;
7  }
```

Делаем несколько запусков подряд:

```
$ gcc prog.c -o prog
$ ./prog
a[2050] = 10
$ ./prog
```

```
segmentation fault (core dumped)
$ ./prog
a[2050] = 10
$
```

Программа работает через раз. Здесь смещение выходит за пределы нашего массива, но таинство реального распределения памяти позволяет нам иногда оставаться в пределах нашего сегмента.

8.8 Санитайзеры

Главным средством борьбы с неопределённым поведением программы являются внимательность и санитайзеры. Санитайзеры — это специальный режим компилятора, который использует дополнительные проверки. В частности, проверяются выходы за границы массивов. Это немного замедляет выполнение программы (и язык C начинает работать по скорости, как многие языки высокого уровня, где проверка индекса — часть языка). Но на задачах первого семестра это замедление никак не существенно.

```
$ gcc prog.c -o prog -fsanitize=undefined
$ ./prog
prog.c:4:10: runtime error:
index 2050 out of bounds for type 'int [16]'
```

8.9 Многомерные массивы

Многомерные массивы в языке C представляют возможность удобной индексации. По аналогии с одномерными массивами они имеют 0-индексацию. Например, так объявляется матрица размера 2×3 (2 строки и 3 столбца) и заполняются два её угловых элемента:

```
int a[2][3];
a[0][0] = 1;
a[1][2] = 2;
```

В данном случае получится матрица следующего вида:

```
1  ?  ?
?  ?  2
```

Например, следующий код позволяет сгенерировать таблицу умножения размера 10×10 в виде двумерного массива:

```
int a[10][10];
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        a[i][j] = (i + 1) * (j + 1);
```

Инициализировать матрицу можно сразу же при объявлении:

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

Матрица будет выглядеть таким образом:

1	2	3
4	5	6

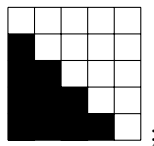
Аналогично можно создавать и массивы большей размерности:

```
int a[2][3][4];
```

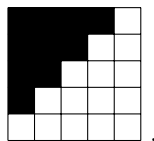
8.10 Матрицы

Одно из важных применений двумерных массивов является реализация матриц в математическом понимании. При решении задач с матрицами рекомендуется использовать естественные обозначения: i — номер строки, j — номер столбца. В практических целях часто необходимо различать расположение элемента в матрице. Например, для элементов главной диагонали матрицы верно: $i = j$; для элементов побочной диагонали: $i + j = n - 1$.

Например, найдем сумму элементов под главной диагональю ($i > j$)



и произведение над побочной диагональю ($i + j < n - 1$)



Код для решения данной задачи выглядит следующим образом:


```

int a[10][10], n;
int sum = 0, prod = 1;
scanf("%d", &n);
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        scanf("%d", &a[i][j]);

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
        if (i < j)
            sum += a[i][j];
        if (i + j < n - 1)
            prod *= a[i][j];
    }

```

8.11 Адреса в двумерном массиве

В памяти матрицы хранятся по строкам:

Адрес	..00	..04	..08	..12	..16	..20
Имя	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Значение	1	2	3	4	5	6

Чтобы узнать адрес матрицы, строки матрицы или элемента матрицы используем общее правило: адрес массива совпадает с адресом первого элемента массива:

```

long a_ptr = (long) &a;
long a0_ptr = (long) &a[0];
long a1_ptr = (long) &a[1];
long a00_ptr = (long) &a[0][0];
long a10_ptr = (long) &a[1][1];
printf("%p\n", a_ptr);
printf("%p\n", a0_ptr);
printf("%p\n", a1_ptr);
printf("%p\n", a00_ptr);
printf("%p\n", a10_ptr);

```

Вывод:

```
140727744864000
```

```
140727744864000
140727744864012
140727744864000
140727744864016
```

Это приводит к необычным обращениям, которые не вызывают ошибок при запуске без санитайзера:

```
printf("%d ", a[0][3]);
printf("%d ", a[0][4]);
printf("%d ", a[0][5]);
printf("%d ", a[1][-1]);
printf("%d ", a[1][-2]);
printf("%d ", a[1][-3]);
```

Вывод:

```
4 5 6 3 2 1
```

8.12 Первая размерность многомерных массивов

У многомерных массивов первая размерность отличается от остальных. Она может быть определена автоматически по инициализации, в то время как остальные размерности всегда задаются явно. Например, таким образом инициализировать матрицу можно:

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

А так инициализировать нельзя:

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

8.13 Примеры

Пример 1. Дано целое число n от 1 до 90. Необходимо найти и вывести первые n чисел Фибоначчи f_i , которые определяются рекуррентно:

$$\begin{cases} f_0 = 0, \\ f_1 = 1, \\ f_i = f_{i-1} + f_{i-2}, i \geq 2. \end{cases}$$

Ввод	4	7
Вывод	0 1 1 2	0 1 1 2 3 5 8

Данную задачу можно решить и без использования массивов. Но решение с массивами значительно проще, так как достаточно написать рекуррентную формулу в явном виде:

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i, f[90] = {0, 1};
6      scanf("%d", &n);
7      for (i = 2; i < n; i++)
8          f[i] = f[i - 1] + f[i - 2];
9      for (i = 1; i < n; i++)
10         printf("%d ", f[i]);
11     putchar('\n');
12     return 0;
13 }
```

Пример 2. На первой строке дано целое число n от 1 до 1000. На второй строке последовательность из n вещественных чисел от -10^9 до 10^9 . Найти позиции всех элементов, равных максимальному значению последовательности.

Ввод	5 5.5 3.1 5.5 5.5 1.2
Вывод	1 3 4

Данная задача легко решается в два этапа: сначала находим максимум, затем выводим все позиции. Так как массив индексируется с нуля, то номера позиций на выводе будут на единицу больше, чем индекс в массиве.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      double a[1000], max;
6      int i, n;
7      scanf("%d", &n);
```

```

8     for (i = 0; i < n; i++)
9         scanf("%f", &a[i]);
10    max = a[0];
11    for (i = 1; i < n; i++)
12        if (max < a[i])
13            max = a[i];
14    for (i = 0; i < n; i++)
15        if (a[i] == max)
16            printf("%d ", i + 1);
17    putchar('\n');
18    return 0;
19 }
```

Пример 3. На первой строке дано целое число n от 1 до 1000. На второй строке дана последовательность из n различных целых чисел от -10^9 до 10^9 . Поменять максимальный и минимальный элемент последовательности местами.

Ввод	5 1 4 2 5 3
Вывод	5 4 2 1 3

Так как все числа различны, то максимальный и минимальный элемент ровно один. Для этого найдём позиции максимума и минимума. Далее изменим ячейки массива.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a[1000], i, n;
6      int min, imin = 0, max, imax = 0;
7      scanf("%d", &n);
8      for (i = 0; i < n; i++)
9          scanf("%d", &a[i]);
10     min = a[0];
11     max = a[0];
12     for (i = 1; i < n; i++)
13     {
14         if (a[i] < min)
15         {
16             min = a[i];
```

```

17         imin = i;
18     }
19     if (a[i] > max)
20     {
21         max = a[i];
22         imax = i;
23     }
24 }
25 a[imin] = max;
26 a[imax] = min;
27 for (i = 0; i < n; i++)
28     printf("%d ", a[i]);
29 putchar('\n');
30 return 0;
31 }

```

Пример 4. На первой строке дано целое число n от 1 до 1000. На второй строке последовательность a_i из n различных целых чисел от -10^9 до 10^9 . Найти и вывести число инверсий (количество пар $i < j$ таких, что $a_i > a_j$).

Ввод	5 1 5 2 4 3	5 5 4 3 2 1
Вывод	4	10

Чтобы получить все пары индексов $i < j$, необходимо перебрать индексы в двойном цикле: для каждого i просмотреть все числа от $i + 1$ до $n - 1$. Зная индексы, легко сравнить соответствующие числа.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i, j, ans = 0, a[100];
6      scanf("%d", &n);
7      for (i = 0; i < n; i++)
8          scanf("%d", &a[i]);
9      for (i = 0; i < n; i++)
10         for (j = i + 1; j < n; j++)
11             if (a[i] > a[j])
12                 ans++;
13     printf("%d\n", ans);

```

```
14     return 0;
15 }
```

Пример 5. На первой строке дано целое число n от 1 до 100. На второй строке последовательность из n различных целых чисел от 0 до 1000. Напечатать по возрастанию только те числа, которые есть в этой последовательности.

Ввод	8 2 5 5 1 5 2 2 2
Вывод	1 2 5

Если решать данную задачу в общем виде, то решение сведётся к сортировке чисел и отбрасыванию повторяющихся чисел. Но здесь стоит обратить внимание на диапазон значений: он целочисленный и небольшой. Соответственно, задачу можно решать методом «подсчёта».

Для метода «подсчёта» необходимо завести дополнительный массив счётчиков `int count[1001]`, где `count[a]` равен количеству элементов исходной последовательности со значением `a`. Например, `count[5]` показывает сколько раз число 5 встретилось в исходной последовательности. Размер массива счётчиков следует выбрать на 1001 элемент, чтобы была возможность подсчитывать числа со значением от 0 до 1000. Очень важно не забыть обнулить массив счётчиков.

Чтобы такой массив заполнить значениям, рассортируем все элементы исходной последовательности на вводе: если встречается элемент последовательности со значением `a`, то в массиве счётчиков увеличим `count[a]++`. Например, как только встречается число 5, то увеличиваем `count[5]` на один. Подробно разберём, как изменяется массив счётчиков из примера в процессе обработки исходной последовательности:

a	count[0]	count[1]	count[2]	count[3]	count[4]	count[5]	...	count[1001]
2	0	0	1	0	0	0	...	0
5	0	0	1	0	0	1	...	0
5	0	0	1	0	0	2	...	0
1	0	1	1	0	0	2	...	0
5	0	1	1	0	0	3	...	0
2	0	1	2	0	0	3	...	0
2	0	1	3	0	0	3	...	0
2	0	1	4	0	0	3	...	0

После построения данного массива достаточно перебрать весь диапазон значений индексов `a` от 0 до 1000 по возрастанию и распечатать те из них, для которых значения массива `count[a]` отличны от нуля. В нашем примере это 1, 2 и 5.

Нет необходимости хранить исходную последовательность в массиве. Достаточно при считывании очередного элемента сразу изменять значение в массиве `count[]`.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i, a, count[1001] = {0};
6      scanf("%d", &n);
7      for (i = 0; i < n; i++)
8      {
9          scanf("%d", &a);
10         count[a]++;
11     }
12     for (a = 0; a <= 1000; i++)
13         if (count[a] > 0)
14             printf("%d ", i);
15     putchar('\n');
16     return 0;
17 }
```

Пример 6. На первой строке дано целое число n от 1 до 1000. На второй строке дана последовательность из n целых чисел от -10^9 до 10^9 . Вывести числа по неубыванию. Использовать сортировку «пузырьком».

Ввод	5 4 5 2 1 3	5 5 4 3 4 3
Вывод	1 2 3 4 5	3 3 4 4 5

Например, отсортируем массив:

4	5	2	1	3
---	---	---	---	---

Выполним сортировку за $(n - 1)$ итерацию. После k -й итерации k -й максимальный элемент должен занять свою позицию.

Процесс первой итерации выглядит следующим образом. Будем последовательно сравнивать два подряд идущих элемента и при необходимости

менять их местами: 4 и 5 не меняем, 5 и 2 меняем, 5 и 1 меняем, 5 и 3 меняем.

4	5	2	1	3
4	5	2	1	3
4	2	5	1	3
4	2	1	5	3
4	2	1	3	5

Процесс второй итерации аналогичен первой итерации, но последний элемент можно уже не сравнивать — это точно самый большой элемент и стоит он на своём месте. Сравниваем: 4 и 2 меняем, 4 и 1 меняем, 4 и 3 меняем.

4	2	1	3	5
2	4	1	3	5
2	1	3	4	5
2	1	3	4	5

Процесс третьей итерации для первых трёх элементов:

2	1	3	4	5
1	2	3	4	5
1	2	3	4	5

Сравнение для первых двух элементов завершает алгоритм.

1	2	3	4	5
1	2	3	4	5

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i, last, a[1000];
6      scanf("%d", &n);
7      for (i = 0; i < n; i++)
8          scanf("%d", &a[i]);
9      for (last = n - 1; last > 0; last--)
10         for (i = 0; i + 1 <= last; i++)
11             if (a[i] > a[i + 1])
12                 {

```



```

13             int t = a[i];
14             a[i] = a[i + 1];
15             a[i + 1] = t;
16         }
17     return 0;
18 }

```

Пример 7. Дана последовательность символов, которая заканчивается переносом строки. Вывести все буквы, которые встречаются в данной строке чаще остальных (буквы разных регистров считаются различными).

Ввод	Hello, Astana!!!
Вывод	a l

Здесь также реализуется идея подсчёта. Но стоит обратить внимание на диапазон значений: 52 символа (26 строчных букв и 26 заглавных букв). Чтобы не усложнять вычисления, можно не производить сжатие, а завести массив для всей ASCII-таблицы на 128 символов. Только значения для символов, отличных от букв, игнорировать. После заполнения массива добавляется стандартная задача нахождения максимума.

Стоит отметить, что для индексации использовать тип `char` не совсем хорошо. При этом выдается предупреждение:

```

$ warning: array subscript has type 'char'
[-Wchar-subscripts]

```

Чтобы избежать такого сообщения, достаточно либо явно приводить к типу большего размера (например к `int`), либо сразу использовать в качестве вводимого числа этот тип.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int count[128] = {0}, ch, max;
6
7      while ((ch = getchar()) != '\n')
8          if ((ch >= 'a' && ch <= 'z') ||
9              (ch <= 'A' && ch >= 'Z'))
10             count[ch]++;
11     max = 0;
12     for (ch = 'a'; ch <= 'z'; ch++)

```

```

13         if (max < count[ch])
14             max = count[ch];
15     for (ch = 'A'; ch <= 'Z'; ch++)
16         if (max < count[ch])
17             max = count[ch];
18     if (max > 0)
19         for (ch = 0; ch < 128; ch++)
20             if (max == count[ch])
21                 printf("%c ", ch);
22     putchar('\n');
23     return 0;
24 }

```

Пример 8. Дана последовательность символов, которая заканчивается переносом строки и не превышает 100 символов. Убрать элементы, равные последнему.

Ввод	ubuntu
Вывод	bnt

Решение заключается использовании техники с говорящим названием — «метод двух указателей».

Для начала разберём наивное решение: два массива `in[]` и `out[]`, у каждого из которых будет своя переменная-итератор `i` и `m` соответственно (те самые два указателя). Если буква `in[i]` подходящая, то копируем букву во второй массив в `out[m]` и сдвигаем оба указателя `i++` и `m++`, в ином случае ничего не делаем со вторым массивом и указателем и двигаем только первый указатель `i++`.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      char out[100], in[100], ch;
6      int n, i, m;
7      ch = getchar();
8      for (n = 0; ch != '\n'; n++)
9      {
10         in[n] = ch;
11         ch = getchar();
12     }
13     for (i = 0, m = 0; i < n; i++)

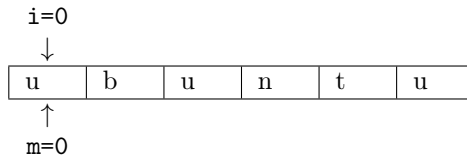
```

```

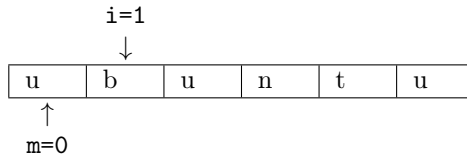
14         if (in[i] != in[n-1])
15         {
16             out[m] = in[i];
17             m++;
18         }
19         for (i = 0; i < m; i++)
20             putchar(out[i]);
21         return 0;
22     }

```

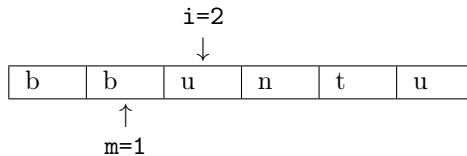
Заметим, что второй массив не нужен. Результат будем сразу же записывать в исходный массив:



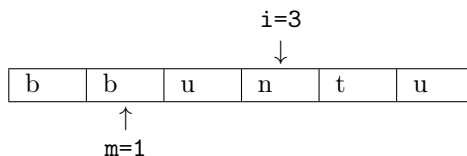
Первую букву не нужно включать в ответ, поэтому её не переписываем. Сдвигаем только указатель i .



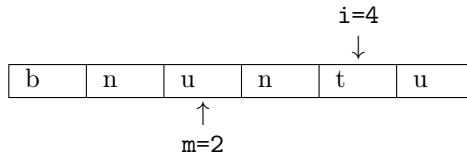
Вторую букву включить в ответ нужно, поэтому её переписываем $a[0] = a[1]$. Сдвигаем оба указателя.



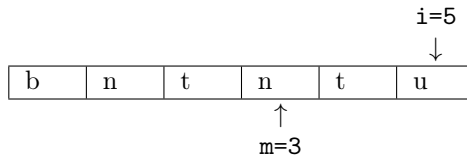
Третья буква не нужна, поэтому пропускаем её. Сдвигаем только указатель i .



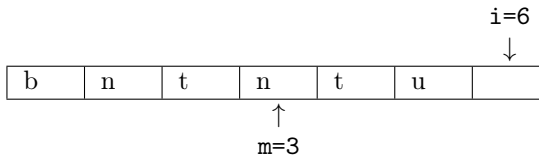
Четвертую букву включаем в ответ, поэтому её переписываем $a[1] = a[3]$. Сдвигаем оба указателя.



По аналогии включаем пятую букву $a[2] = a[4]$.



И пропускаем шестую:



В m хранится количество переписанных букв, которые находятся в начале исходного массива. Соответственно, очень легко исправить код:

```
#include <stdio.h>

int main()
{
    char a[100], ch;
    int n, i, m;
    ch = getchar();
    for (n = 0; ch != '\n'; n++)
    {
        a[n] = ch;
        ch = getchar();
    }
    for (i = 0, m = 0; i < n; i++)
        if (a[i] != a[n-1])
        {
```

```

        a[m] = a[i];
        m++;
    }
    for (i = 0; i < m; i++)
        putchar(a[i]);
    return 0;
}

```

Пример 9. Дано целое положительное число n от 1 до 10. Далее 2 матрицы размера $n \times n$ из целых чисел от -1000 до 1000 . Найти произведение матриц.

Ввод	3 1 2 1 0 1 0 2 1 0 1 -1 1 2 0 -2 0 1 0
Вывод	5 0 -3 2 0 -2 4 -2 0

Обратим внимание, что каждый из n^2 элементов произведения вычисляется следующим образом:

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}.$$

Таким образом, для решения понадобится реализовывать тройной цикл (для каждого из n^2 элементов необходимо найти скалярное произведение, которое выполняется в цикле за n умножений).

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a[100][100], b[100][100], c[100][100];
6      int i, j, k, n;
7      scanf("%d", &n);
8      for (i = 0; i < n; i++)
9          for (j = 0; j < n; j++)

```

```
10         scanf("%d", &a[i][j]);
11     for (i = 0; i < n; i++)
12         for (j = 0; j < n; j++)
13             scanf("%d", &b[i][j]);
14     for (i = 0; i < n; i++)
15         for (j = 0; j < n; j++)
16         {
17             c[i][j] = 0;
18             for (k = 0; k < n; k++)
19                 c[i][j] += a[i][k] * b[k][j];
20         }
21     for (i = 0; i < n; i++)
22     {
23         for (j = 0; j < n; j++)
24             printf("%4d", c[i][j]);
25         putchar('\n');
26     }
27     return 0;
28 }
```

Обратите внимание, что для вывода лучше использовать выравнивание по ширине вывода спецификатором `%4d`, чтобы столбцы матрицы были выровнены.

8.14 Задания для самостоятельной работы

1. Что будет напечатано?

```
int a[] = {1, 5, 2, 4, 3, 3}, i;
for (i = -10; i < 6; i += 3)
    if (i >= 0)
        printf("%d ", a[i]);
```

2. Поменять местами центральный и последний элемент массива из 99 элементов (описать ровно 3 оператора присваивания).

```
int a[99];
...
```

3. Дан массив на 26 элементов `char a[26]`. Описать цикл, который заполняет массив заглавными буквами латинского алфавита.

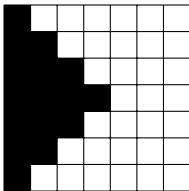
```
char a[26];
...
```

4. Что будет в массиве `count[]` после первого цикла? Что будет напечатано после второго цикла?

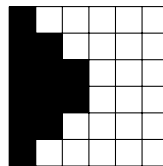
```
int a[10] = {1, 2, 2, 7, 7, 5, 7};
int count[8] = {0}, i;
for (i = 0; i < 10; i++)
    count[a[i]]++;
for (i = 0; i <= 7; i++)
    for (; count[i] > 0; count[i]--)
        printf("%d ", i);
```

5. Необходимо найти сумму элементов квадратной матрицы, которые находятся в указанных ячейках.

для нечетного порядка



для четного порядка



Запишите соответствующие действия для получения ответа в переменной `ans`.

```
int a[10][10], ans = 0, n, i, j;
scanf("%d", &n);
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        scanf("%d", &a[i][j]);

...
```


8.15 Практикум на ЭВМ

1. На первой строке дано целое число n от 1 до 10^3 . На второй строке n целых чисел от -10^6 до 10^6 . Посчитать и вывести количество чисел, равных первому и последнему соответственно.

Ввод	7 3 3 3 2 3 1 2
Вывод	4 2

2. На первой строке дано целое число n от 1 до 10^3 . На второй строке дана последовательность из n целых чисел от -10^6 до 10^6 . Вывести минимальное значение последовательности и все позиции минимальных элементов.

Ввод	7 1 3 3 2 3 1 2	5 -1 -1 -2 1 2
Вывод	1 1 6	-2 3

3. На первой строке дано целое число n от 1 до 10^3 . На второй строке дана последовательность из n целых чисел от 0 до 1000. Отсортировать последовательность по неубыванию методом «подсчёта».

Ввод	5 1 3 4 1 3	5 5 1 4 2 3	3 5 5 5
Вывод	1 1 3 3 4	1 2 3 4 5	5 5 5

4. На первой строке дано целое число n от 1 до 10^3 . На второй строке дана последовательность из n целых чисел от 0 до 1000. Вывести по возрастанию все числа, которые встречаются чаще остальных.

Ввод	10 8 1 7 8 7 1 5 3 1 7
Вывод	1 7

5. На первой строке дано целое число n от 1 до 10^3 . На второй строке дана последовательность из n целых чисел от -10^6 до 10^6 . Посчитать количество инверсий в последовательности и вывести все пары чисел, которые их образуют.

Ввод	5 1 5 2 4 3
Вывод	4 5 2 5 4 5 3 4 3

6. Дано целое положительное число n от 1 до 10. Далее дана матрица размером $n \times n$ из целых чисел от -100 до 100. Найти произведение всех ненулевых элементов над главной диагональю.

Ввод	3 2 5 0 2 2 3 2 2 2	5 1 2 3 2 1 2 1 0 1 2 0 0 0 0 0 1 2 3 2 1 0 0 0 0 0
Вывод	15	24

7. Дано целое положительное число n от 1 до 10. Далее дана матрица размером $n \times n$ из целых чисел от -100 до 100. Далее вектор размером n . Найти произведение этой матрицы на данный вектор.

Ввод	2 1 2 1 0 -3 4	3 1 2 3 1 0 1 2 1 0 1 2 3
Вывод	5 -3	14 4 4

8. На первой строке дано целое число n от 1 до 10^3 . Далее дана матрица размером $n \times n$ из целых чисел от -100 до 100. Найти все седловые точки матрицы. Седловая точка матрицы — элемент матрицы $a_{i,j}$ такой, что $a_{i,j}$ больше всех элементов i -й строки и меньше всех элементов j -го столбца. Вывести позицию седловой точки и ее значение.

Ввод	3 13 5 0 12 1 1 14 4 4	4 2 3 4 5 1 2 3 2 6 5 4 3 2 3 5 4
Вывод	2 1 12	2 3 3

9. Пусть $f_0 = 0$, $f_1 = 1$, $f_k = f_{k-1} + f_{k-2}$ при $k > 1$ (числа Фибоначчи). Дано целое x от 1 до $2 \cdot 10^{18}$. Найти n и f_n такие, что $f_{n-1} \leq x < f_n$.

Ввод	10	1234567890987654321
Вывод	7 13	89 1779979416004714189

10. Дано целое число n от 1 до 10^6 . Построить решето Эратосфена в виде массива a , где:

$$a_i = \begin{cases} 0, & \text{если число } i \text{ не простое} \\ 1, & \text{если число } i \text{ простое} \end{cases}$$

Ввод	20
Вывод	0 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0

Алгоритм построения решета:

- заполняем весь массив единицами,
- заменяем на ноль все элементы с индексом $2k$ для всех целых $k > 1$,
- заменяем на ноль все элементы с индексом $3k$ для всех целых $k > 1$,
- заменяем на ноль все элементы с индексом $4k$ для всех целых $k > 1$,
- и так далее.

9 Указатели и строки

В текущей теме будет подробно рассмотрено понятие указателя и описаны основные принципы работы со строками в языке C. Также рассмотрены библиотеки «ctype.h» и «string.h», которые могут быть полезны при работе со строками.

9.1 Взятие адреса

Вводная информация об адресах была представлена в теме «Массивы». Оператор для получения адреса обозначается символом `&` (амперсанд). Например, так можно узнать адрес произвольной переменной `a`:

```
&a
```

Адреса необходимо хранить в переменных специального типа данных, которые называются указатель. Например, таким образом можно объявить переменную `ptr` типа `(int *)` — «указатель на `int`»:

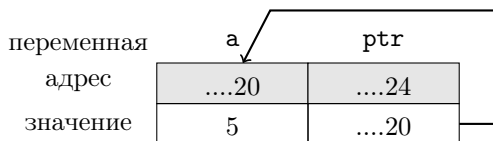
```
int *ptr;
```

Записать адрес переменной `a` в указатель `ptr` можно обычным оператором присваивания:

```
int a;
int *ptr;

a = 5;
ptr = &a;
```

В переменной `a` (с типом `int`) записано число 5, а в переменной `ptr` (с типом указатель на `int`) записан адрес переменной `a`. Соответствующая схема выглядит так:



Можно объявлять указатели на любые типы данных. Например, указатель на `double` и указатель на `char` выглядят следующим образом:

```
double *x;
char *s;
```

Размер указателя для любого типа данных фиксирован и зависит от битности операционной системы: для 32-битной системы он будет равен 4 байтам, для 64-битной — 8 байтам.

9.2 Разыменование указателя

С помощью указателя можно косвенно узнавать и изменять значение переменной, адрес которой он хранит. Для этого используется оператор разыменования `*` (звездочка). Например, так можно получить доступ к ячейке (ячейкам) памяти по указателю `ptr`:

```
*ptr
```

Рассмотрим пример:

```
1 int a = 5, b;
2 int *ptr;
```

Допустим адреса первых ячеек памяти для переменных `a`, `b` и `ptr` заканчиваются на 20, 24 и 28:

переменная	a	b	ptr
адрес	..20	..24	..28
значение	5	?	?

Сохраним в переменную `ptr` адрес переменной `a`:

```
3 ptr = &a;
```

Для наглядности указатель соединим стрелкой с ячейкой памяти, на которую он указывает:

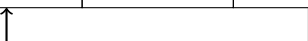
переменная	a	b	ptr
адрес202428
значение	5	?20

Запишем в переменную `b` значение, которое получим разыменовав данный указателя:

```
4 b = *ptr;
```

Таким образом в код выполнено присваивание `b = a`, обращаясь к переменной `a` косвенно через её адрес.

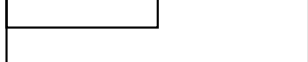
переменная	a	b	ptr
адрес202428
значение	5	520



Адрес переменной можно сохранить в нескольких указателях. Например, такой код сохраняет адрес переменной `a` сразу в двух указателях:

```
1 int a = 5;
2 int *ptr1, *ptr2;
3 ptr1 = &a;
4 ptr2 = &a;
```

переменная	a	ptr1	ptr2
адрес202432
значение	52020



Продолжим код записанный выше. Изменим значение переменной `a` в явном виде, а затем выведем на экран её значение явным и неявным способом:

```
5 a = 7;
6 printf("%d %d %d\n", a, *ptr1, *ptr2);
```

В результат программа выдаст следующие результаты:

```
7 7 7
```

А теперь изменим значение переменной `a` неявно через указатель и выведем результат на экран:

```
7 *ptr1 = a + (*ptr2);
8 printf("%d %d %d\n", a, *ptr1, *ptr2);
```

Получим:

```
14 14 14
```

9.3 Ваш второй segmentation fault

Разыменовывать можно только указатели, в которых хранятся адрес доступной памяти. В противном случае можно получить программу с неопределённым поведением:

```
int *ptr;  
*ptr = 5;
```

Изначально в переменной `ptr` лежит мусорное значение. То есть во второй строке происходит разыменование мусорного адреса (определён случайным образом). Это приведёт либо к ошибке сегментации данных:

```
segmentation fault (core dumped)
```

либо к непреднамеренному изменению других данных.

Для предотвращения подобных ошибок рекомендуется инициализировать указатели при объявлении. Например, значением специальной константы `NULL` (указатель, который никуда не указывает):

```
int *ptr = NULL;
```

Теперь в указателе не будет мусорного значения. Перед любой попыткой разыменовывать указатель `ptr` достаточно убедиться, что он отличен от значения `NULL`. Например, следующим образом:

```
int *ptr = NULL;  
...  
if (ptr != NULL)  
    *ptr = 5;
```

Такой код не приведёт к ошибке, так как пропустит попытку разыменования. А следующий вариант:

```
int *ptr = NULL;  
...  
int a;  
ptr = &a;  
...  
if (ptr != NULL)  
    *ptr = 5;
```

успешно запишет в переменную `a` значение 5.

Учитывая, что `NULL` при сравнении приводится к нулевому значению, которое трактуется как «ложь», можно использовать более компактную запись:

```
int *ptr = NULL;
...
if (ptr)
    *ptr = 5;
```

9.4 Арифметика указателей

Помимо разыменования с указателем можно производить арифметические действия: прибавление и вычитание целых чисел для смещения указателя. Самое простое смещение — на соседние ячейки:

```
ptr++;
```

При этом адрес изменится на величину равную размеру адресуемого типа в байтах. Например, указатель типа (`char *ptr`) при инкременте сместится на 1 байт, указатель типа (`int *ptr`) при инкременте — на 4 байта, указатель типа (`double *ptr`) — на 8 байт, указатель на массив из 10 целых чисел типа `int (*ptr)[10]` сдвигается на 40 байт.

Указатель можно смещать и на любое целое число:

```
...
int *ptr = &a;
...
int i = 10;
ptr = ptr + i;
ptr -= i;
```

В данном коде указатель сначала сдвинется на 40 байт в сторону увеличения адресов, а потом вернётся обратно.

9.5 Приоритет

Приоритет оператора разыменования выше приоритета арифметических операторов. Рассмотрим это на примере следующей конструкции:

```
1 int a = 5;
2 int b = 7;
3 int *ptr a, *ptr b;
4 ptr a = &a;
5 ptr b = &b;
```

В данном случае переменные в памяти раскладываются следующим образом:

переменная	a	b	ptr a	ptr b
адрес20243240
значение	5	72024

Определим, к чему приведёт данная команда:

```
5 *ptr a = *ptr a + *ptr b;
```

Оператор разыменования выполняется до оператора сложения. Поэтому данное действие является косвенным аналогом действия: $a = a + b$. То есть, в переменной **a** значение изменится с 5 на 12.

Рассмотрим ещё один пример с изменением приоритета с помощью круглых скобок. Следующий код присвоит переменной **c** значение 6:

```
int a = 5, b = 10, c;
int *ptr = &a;
c = (*ptr) + 1;
```

Рассмотрим тот же код, но с изменённым приоритетом операторов:

```
int a = 5, b = 10, c;
int *ptr = &a;
c = *(ptr + 1);
```

Сначала произойдет смещение указателя. Указатель **ptr** будет указывать на следующий адрес, то есть на переменную **b**. В итоге, $a = 5$, $b = 10$, $c = 10$.

9.6 Работа с массивами через указатели

После объявления статического массива, переменная с именем массива хранит адрес первой ячейки массива. Это позволяет косвенно получать и изменять значения элементов массива. Рассмотрим пример:

```
1 int s[5] = {11, 12, 13, 14, 15};
2 int *ptr = s;
```

В переменной **s** хранится адрес первого элемента. Поэтому в данном случае в переменной **ptr** лежит адрес четырёх байт памяти с числом 11. Изменим значение:

```
3 *ptr = 10;
```

Массив `s` теперь выглядит следующим образом:

s[0]	s[1]	s[2]	s[3]	s[4]
10	12	13	14	15

Продолжим код:

```
4 *(ptr + 2) = 20;
5 *(ptr + 3) = 20;
```

Так как `ptr` указывает на тип `int`, то смещения на два и три элемента сдвигают адрес в указателе на 8 и 12 байт соответственно. Поэтому доступ равносильен прямому доступу через квадратные скобки `[]` в массиве: `s[2]` и `s[3]`. В результате получим:

s[0]	s[1]	s[2]	s[3]	s[4]
10	12	20	20	15

9.7 Полезные соотношения

Можно заметить, что

```
*(amp a) == a
```

— верно всегда;

```
&(*a) == a
```

— верно, только если `a` — корректный указатель, который можно разыменовывать.

В общем виде для массива `a[]` верно следующее тождество:

```
a[i] == *(a + i);
```

Данное соотношение приводит еще к следующему соотношению:

```
&a[i] == &*(a + i) == a + i;
```

которое можно использовать для ввода массива:

```
int a[5], i;
for (i = 0; i < 5; i++)
    scanf("%d", a + i);
```

9.8 Приведение указателей

Оператор разыменования указателя даёт доступ к блоку из n байт, где n согласовано с размером типа, на который указывает указатель. Приведение указателей к другому типу позволяет изменить количество ячеек, к которым осуществляется доступ при разыменовании. Рассмотрим это на примере. Пусть даны три указателя и массив из 4 байт:

```
1 int *x;
2 short *y;
3 char *z;
4 char s[] = {4, 3, 2, 1};
```

Переменная `s` хранит адрес первой ячейки памяти массива. Приведём данный адрес к соответствующим типам и присвоим объявленным указателям:

```
5 x = (int *)s;
6 y = (short *)s;
7 z = (char *)s;
```

Разыменуем все три указателя и выведем полученные значения:

```
8 printf("%d\n%d\n%d\n", *x, *y, *z);
```

На выводе будут числа:

```
16909060
772
4
```

Поясним, как получаются такие значения. Начиная с адреса `s`, в памяти хранится 4 байта со значениями: 4, 3, 2, 1. Когда производится разыменование данного адреса через указатель `x` (указатель на `int`), получается четырёхбайтовое число типа `int`. Формирование четырёхбайтового значения из четырёх подряд идущих байт происходит от младших байт к старшим. То есть число 4 формирует 8 младших бит числа, 3 формирует биты с 9-го по 16-й, число 2 для битов с 17-го по 24-й и 1 отвечает за самые старшие биты. Таким образом значение при разыменовании `*x` равно:

$$4 + 256 * 3 + 256^2 * 2 + 256^3 * 1 = 16909060$$

Разыменование адреса `s` через указатель `y` (указатель на `short`) означает, что необходимо получить двухбайтовое число типа `short`, значение которого равно:

$$4 + 256 * 3 = 772$$

Если по адресу `s` мы хотим получить однобайтовое число типа `char`, то его значение определяется как 4.

9.9 Вывод адресов

Для вывода адреса в шестнадцатеричной системе счисления можно использовать спецификатор `%p` в функции `printf`:

```
int a;
int *ptr;
a = 5;
ptr = &a;
printf("%p", ptr);
```

Для вывода адреса в десятичной системе счисления указатель можно привести к типу `unsigned long` и использовать спецификатор `%lu` в функции `printf`:

```
int a;
int *ptr;
a = 5;
ptr = &a;
printf("%lu", (unsigned long)ptr);
```

9.10 Строка

Печатными символами ASCII-таблицы являются символы с кодами от 33 до 127, а непечатными — с кодами от 0 до 32. Строкой в языке C обозначают массив из печатных символов ASCII-таблицы, который обязательно заканчивается непечатным символом `'\0'` с кодом 0 (не путайте с печатным символом `'0'` с кодом 48). Размер массива должен быть не менее $(n + 1)$ байт, где n — количество печатных символов строки. Например, для строки `math` подойдет массив размером 5:

```
char s[5];
s[0] = 'm';
s[1] = 'a';
s[2] = 't';
s[3] = 'h';
s[4] = '\0';
```

Инициализировать строки при объявлении переменной можно одним из трёх вариантов:

```
char s1[] = "math";
char s2[] = {'m', 'a', 't', 'h', '\\0'};
char s3[] = {109, 97, 116, 104, 0};
```

Наиболее распространённым вариантом является первый способ. При такой инициализации строки нулевой символ в конце дописывать вручную не нужно, а размер массива определится автоматически с учётом нулевого символа.

Размер массива можно задать и вручную, главное, чтобы он был строго больше количества печатных символов строки:

```
char s[7] = "math";
```

Массив будет инициализирован следующим образом:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]
'm'	'a'	't'	'h'	'\\0'	'\\0'	'\\0'

9.11 Ввод-вывод строк с использованием `stdio.h`

Ввод и вывод строк можно производить как и при работе с обычными массивами. Главное — не забывать дописывать символ с кодом ноль в конец массива при вводе. Но можно воспользоваться готовыми функциями из библиотеки «`stdio.h`».

Рассмотрим три пары функций ввода-вывода. Первая пара функций: `printf` и `scanf` со спецификатором `%s`:

```
char str[10];
scanf("%s", str);
printf("%s", str);
```

Обратите внимание, что амперсанд при вводе не используется. Функция `scanf` считывает строку до пробельного символа (пробел, табуляция, символ новой строки) и сохраняет по указателю `str`. В конце автоматически добавляется символ `'\\0'`. Функция выполняет небезопасный ввод: если ввести больше символов, чем размер доступной памяти по указателю, то получим неопределённое поведение программы.

Вторая пара функций: `puts` и `gets`.

```
char str[10];
gets(str);
puts(str);
```

Функция `gets` считывает символы до переноса строки и записывает их по указателю `str`. Вместо символа переноса строки в конец добавляется символ `'\0'`. Ввод является небезопасным, о чём и сообщает компилятор:

```
warning: the `gets' function is dangerous
        and should not be used.
```

Функция `puts` после вывода строки добавляет перенос строки.

Третий вариант: `fputs` и `fgets`.

```
char str[10];
fgets(str, 10, stdin);
fputs(str, stdout);
```

Функция `fgets` считывает не более $(n - 1)$ символа (где n — второй аргумент) и сохраняет их по указателю `str`. Если на вводе есть перенос строки, то он добавляется к строке и ввод заканчивается досрочно. Терминальный символ `'\0'` добавляется в любом случае. Функцией `fputs` выводит только строку без добавления переноса строки.

Если применять функции из одной пары ввода и вывода, то на выводе всегда будет перенос строки. Второй вариант:

```
char s[100];
gets(s);
puts(s);
```

После ввода текста «math» в строке `s` хранится 5 значений `'m'` `'a'` `'t'` `'h'` `'\0'`. На вывод будет отправлено 5 значений `'m'` `'a'` `'t'` `'h'` `'\n'`:

```
$ ./prog
math
math
$
```

Третий вариант:

```
char s[100];
fgets(s, 100, stdin);
fputs(s, stdout);
```

После ввода текста «math» В строке `s` хранится 6 значений `'m'` `'a'` `'t'` `'h'` `'\n'` `'\0'`. На вывод также будет отправлено 5 значений `'m'` `'a'` `'t'` `'h'` `'\n'`:

```
$ ./prog
```

```
math
math
$
```

9.12 Особенности ввода–вывода

Все функции вывода схематично работают следующим образом:

```
char *ptr;
for (ptr = s; *ptr; ptr++)
    putchar(*ptr);
```

Здесь учтено то, что все ненулевые значения трактуются как «истина». Получается, что на экран выводятся все символы под указателем, а сам указатель сдвигается до тех пор, пока не встретится нулевой символ `'\0'`. Если такого символа в строке нет, то может произойти ошибка сегментации данных.

При реализации ввода вручную, то важно не забывать две правила:

1. для строки необходимо выделить на 1 байт больше, чем количестве печатных символов;
2. после печатных символов строки необходимо дописать нулевой символ.

Например, такой ввод подойдет только для текста не более 10 символов:

```
char s[11];
char *ptr;

ptr = s;
*ptr = getchar();
while (*ptr != '\n') {
    ptr++;
    *ptr = getchar();
}
*ptr = '\0';
```

Рассмотрим базовый пример работы с функциями ввода-вывода. Дана строка, состоящая не более чем из 100 символов. Необходимо вывести её длину.

Напишем код с использованием функции `gets`:

```
int n;
char a[101];
gets(a);
for (n = 0; a[n]; n++) {
}

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

В данном коде проявляется полезное применение пустого тела оператора цикла `for`. Обратите внимание, что нельзя забывать выделять память под символ `'\0'`.

А теперь выведем длину строки с вводом через функцию `fgets`:

```
int n;
int a[102];
fgets(a, 101, stdin);
for (n = 0; a[n]; n++) {
}

n--;
a[n] = '\0';

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

При вводе с применением безопасной функцией `fgets` есть два существенных отличия от небезопасной функции `gets`:

1. размер массива на 2 байта больше количества печатных символов строки, так как в конце дописывается `'\n'` и `'\0'`;
2. символ переноса строки в конце следует заменить на `'\0'` для приведения строки к стандартному виду.

9.13 Библиотека «ctype.h»

Задачи на обработку символов часто сводятся к проверке символов на принадлежность некоторым множествам символов: строчным буквам, заглавным буквам, цифрам и другим. Производить проверку можно не только вручную, но и с помощью функций, описанных в «ctype.h». В качестве аргумента передаётся проверяемый символ типа `int`. Как правило, в программе будет передаваться тип `char`, приведённый к типу `int`. Функция возвращает нулевое значение, если символ не принадлежит соответствующему множеству, и ненулевое значение — в противном случае:

Функция	Множество
<code>int isdigit(int c);</code>	цифра
<code>int isalpha(int c);</code>	буква
<code>int islower(int c);</code>	символ нижнего регистра
<code>int isupper(int c);</code>	символ верхнего регистра
<code>int isalnum(int c);</code>	буква или либо цифра
<code>int isgraph(int c);</code>	печатный символ, отличный от пробела
<code>int isprint(int c);</code>	печатный символ
<code>int ispunct(int c);</code>	пунктуация (не пробел, буква или цифра)
<code>int isspace(int c);</code>	пробельный символ (' ', '\n', '\r', '\t')

Также имеются две функции преобразования регистров букв:

Функция	Действия
<code>int tolower(int c);</code>	изменить верхний регистр на нижний
<code>int toupper(int c);</code>	изменить нижний регистр на верхний

Отличные от букв символы остаются неизменными.

Дана строка, состоящая не более чем из 100 символов. Необходимо изменить все буквы на заглавные.

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    int i = 0;
    char a[101];
    fgets(a, 101, stdin);
    for (i = 0; a[i]; i++)
        a[i] = toupper(a[i]);
    fputs(a, stdout);
    return 0;
}
```

9.14 Библиотека «string.h»

Строки в языке C не являются цельным объектом. Поэтому, в отличие от большинства языков высокого уровня, здесь нет операторов сравнения и присваивания строк. Например запись:

```
char s[7];
s = "math";
```

будет ошибочной. На экране появится соответствующее сообщение:

```
prog.c:4:7: error: assignment to expression
      with array type
      s = "math";
      ^
```

Скопировать одну строку в другую можно либо поэлементно, либо используя функции из библиотеки «string.h», о которых речь будет ниже.

Указатели, как и обычные переменные, бывают константными:

```
const char *
```

В нашем курсе такие указатели будут встречаться в качестве аргументов функций из «string.h». Это означает, что в качестве соответствующего аргумента можно использовать как обычные строки (типа `char *`):

```
char s[5] = "math";
int n = strlen(s);
```

так и константные строки в двойных кавычках (типа `const char *`):

```
int n = strlen("mech");
```

9.15 Длина строки

Длину строки можно узнать следующей функцией:

```
size_t strlen(const char *s);
```

Возвращаемый тип `size_t`, как правило, будет приводиться к типу `int`:

```
int s[] = "math";
int n = strlen(s);
```

После выполнения функции `strlen` в переменной `n` окажется количество печатных символов строки, то есть 4.

Ввиду того что для вычисления длины строки функция `strlen` просматривает всю строку от начала до конца, она может существенно увеличить время работы программы. Сравним два примера решения одной и той же задачи с неэффективным и эффективным использованием данной функции.

Дана строка длиной не более 1000 символов. Цель: посчитать количество цифр в строке.

Пример решения с неэффективным использованием функции:

```
char s[1001];
int ans = 0;
for (i = 0; i < strlen(s); i++)
    if (isdigit(s[i]))
        ans++;
```

Оценим количество действий в программе. Для вычисления длины строки функция `strlen` выполнит около 1000 действий. Сама функция будет вызвана на каждой из 1000 итераций цикла. То есть общее количество действий будет порядка 1000×1000 .

Вариант эффективного использования функции:

```
char s[1001];
int ans = 0, n = strlen(s);
for (i = 0; i < n; i++)
    if (isdigit(s[i]))
        ans++;
```

Функция `strlen` будет вызвана только один раз, то есть общее количество действий будет порядка 1000.

9.16 Сравнение строк

Пример 1. Даны два слова из строчных букв, разделенные между собой переносом строки. Суммарная длина слов составляет не более 100 символов. Необходимо определить, какое из слов лексикографически меньше.

Ввод	math math	algorithm math	algorithm algebra
Вывод	equal	first	second

Сравнение производится по первому различающемуся символу. Например, строка `algorithm` лексикографически меньше строки `math`, но больше строки `algebra`. При этом, если первое слово является началом второго слова, то он меньше второго. Например, строка `mathematic` лексикографически меньше строки `mathematics`, но больше строки `math`.

Сначала найдём первую позицию, в которой различаются строки или закончилась хотя бы одна из строк. Если обе строки закончились одновременно, то они равны. Если закончилась первая строка — то она идет раньше; если вторая — то вторая. Если же они не закончились, то достаточно просто сравнить текущие символы.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int ans, i;
6      char s1[101], s2[101];
7      scanf("%s %s", &s1, &s2);
8      for (i = 0;
9           s1[i] != '\0' && s2[i] != '\0';
10          i++)
11          if (s1[i] != s2[i])
12              break;
13
14      if (s1[i] == '\0' && s2[i] == '\0')
15          ans = 0;
16      else if (s1[i] == '\0' && s2[i] != '\0')
17          ans = -1;
18      else if (s1[i] != '\0' && s2[i] == '\0')
19          ans = 1;
20      else if (s1[i] < s2[i])
21          ans = -1;
22      else if (s1[i] > s2[i])
23          ans = 1;
24
25      if (ans < 0)
26          puts("first");
27      if (ans > 0)
28          puts("second");
29      if (ans == 0)
30          puts("equal");
31      return 0;
32 }
```

В библиотеке `string.h` предусмотрена функция для такого сравнения:

```
int strcmp(const char *s1, const char *s2);
```

Возвращаемый результат может принимать значения:

- меньше нуля, если строка `s1` меньше строки `s2`;
- больше нуля, если строка `s1` больше строки `s2`;

- равна нулю, если строка `s1` равна строке `s2`.

Теперь решение можно записать следующим образом:

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      int ans, i;
7      char s1[101], s2[101];
8      scanf("%s %s", &s1, &s2);
9
10     ans = strcmp(s1, s2);
11
12     if (ans < 0)
13         puts("first");
14     if (ans > 0)
15         puts("second");
16     if (ans == 0)
17         puts("equal");
18     return 0;
19 }
```

Обратите внимание, что в качестве аргументов можно передавать и константные строки в двойных кавычках:

```

char s[] = "algebra";
int r = strcmp("algorithm", s);
```

В переменной `r` будет положительное число.

Следующая функция позволяет сравнить не всю строку целиком, а только её часть (подстроку):

```
int strncmp(const char *a, const char *b, size_t n);
```

Данная функция абсолютно аналогична предыдущей функции за исключением того, что сравниваются не более n первых символов двух строк.

```

char a[] = "math";
char b[] = "mathematic";
int r1 = strncmp(a, b, 4);
int r2 = strncmp(a, b, 5);
```

Сравнение `'math'` и `'math'` даст ноль, а сравнение `'math'` и `'mathe'` даст отрицательный результат.

9.17 Поиск подстроки в строке

Пример 2. На первой строке дано слово w . На второй — произвольная строка s . И слово, и строка содержат не более 100 символов. Необходимо определить, есть ли слово w в строке s . Если слово есть, то вывести первую позицию в строке s , с которой начинается слово w , иначе вывести 0.

Ввод	<code>math mechmath-1</code>	<code>math mate</code>	<code>math math and math</code>
Вывод	5	0	1

Идея достаточно проста: пробуем сравнить слово со строкой, начиная с первой буквы слова и первой буквы строки. Если они не совпадают, то сравниваем с первой буквы слова и второй буквы строки. И так далее.

Код данного решения представлен ниже:

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int i, j, n, m;
6      char word[102], str[102];
7
8      fgets(word, 101, stdin);
9      for (n = 0; word[n]; n++) {
10     }
11     n--;
12     word[n] = '\0';
13
14     fgets(str, 101, stdin);
15     for (m = 0; str[m]; m++) {
16     }
17     m--;
18     str[m] = '\0';
19
20     for (i = 0; i + n <= m; i++)
21     {
22         for (j = 0; j < n; j++)
23             if (word[j] != str[i + j])
24                 break;
25         if (j == n)
```

```

26         {
27             printf("%d\n", i + 1);
28             break;
29         }
30     }
31     return 0;
32 }

```

Решим данную задачу с использованием готовых функций: для вычисления длины строки используем функцию `strlen`, а для сравнения строк — `strncmp`.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      int i, n, m;
7      char word[102], str[102];
8
9      fgets(word, 101, stdin);
10     n = strlen(word) - 1;
11     word[n] = '\0';
12
13     fgets(str, 101, stdin);
14     m = strlen(str) - 1;
15     str[m] = '\0';
16
17     for (i = 0; i + n <= m; i++)
18         if (strncmp(word, str + i, n) == 0)
19             {
20                 printf("%d\n", i + 1);
21                 break;
22             }
23     return 0;
24 }

```

Рассмотрим альтернативное решение с использованием функции поиска подстроки в строке:

```

char *strstr(
    const char *haystack,
    const char *needle

```

```
);
```

Данная функция ищет первое вхождение подстроки **needle** (с англ. игла) в строке **haystack** (с англ. стог сена). Если искомая строка есть, то функция возвращает указатель на ячейку **haystack**, с которой начинается строка **needle**. Если такой строки нет, то результатом будет **NULL** — нулевой указатель.

Рассмотрим подробнее работу данной функции на примере двух строк:

```
char needle[] = "math";
char haystack[] = "mmath1";
```

Пусть строка **haystack** представлена в памяти следующим образом:

Адрес	..20	..21	..22	..23	..24	..25	..26
Значение	'm'	'm'	'a'	't'	'h'	'1'	'\0'

Определим результаты поиска некоторых подстрок в строке:

```
char *ptr1 = strstr(haystack, needle);
char *ptr2 = strstr(haystack, "1");
char *ptr3 = strstr(haystack, "mech");
```

Значения указателей будут равны **..21**, **..26** и **NULL** соответственно. Для определения позиции вхождения подстроки вычтем из адреса, который вернула функция, адрес начала строки **haystack** — это и будет смещение относительно начала строки.

Последний вариант решения Примера 2:

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     int ans, n, m;
6     char word[102], str[102];
7
8     fgets(word, 101, stdin);
9     n = strlen(word) - 1;
10    word[n] = '\0';
11
12    fgets(str, 101, stdin);
13    m = strlen(str) - 1;
14    str[m] = '\0';
15
```



```

16     char *ptr = strstr(str, word);
17     if (ptr == NULL)
18         ans = 0;
19     else
20         ans = ptr - str + 1;
21     printf("%d\n", ans);
22     return 0;
23 }

```

9.18 Копирование строк

Ещё одной полезной функцией для строк является копирование строки.

```

char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);

```

Копируются в память данные, начиная с указателя **src** (англ. source — источник), под указатель **dest** (англ. destination — назначение). Синтаксис близок к функциям сравнения **strcmp** и **strncmp**. Стоит не забывать, что терминальный ноль тоже копируется.

Так можно реализовать простое копирование строки из массива **from** в массив **to**:

```

char from[10] = "Hello";
char to[10];
strcpy(to, from);

```

Рассмотрим более сложное копирование: строку «mechmath» преобразуем в строку «mathmech». Для этого воспользуемся арифметикой указателей. Для начала скопируем суффикс строки **from** (то есть с середины до конца строки) функцией **strcpy**:

```

1 char from[] = "mechmath";
2 char to[10];
3 strcpy(to, from + 4);

```

Получим строку **to**:

Адрес	..20	..21	..22	..23	..24	..25	..26	..27	..28	..29
Значение	'm'	'a'	't'	'h'	'\0'	?	?	?	?	?

Допишем первые 4 байта строки **from** после слова **math** в строке **to**. Воспользуемся функцией **strncpy**:

```
4 strcpy(to + 4, 4, from);
```

Получим строку to:

Адрес	..20	..21	..22	..23	..24	..25	..26	..27	..28	..29
Значение	'm'	'a'	't'	'h'	'm'	'e'	'c'	'h'	'?'	'?'

Чтобы массив символов преобразовать к строке, добавим терминальный ноль:

```
5 to[9] = '\0';
```

9.19 Заполнение массива

Функция `memset` позволяет инициализировать все элементы строки или массива одинаковым значением:

```
void *memset(void *s, int c, size_t n);
```

Ранее не упоминавшийся тип `(void *)` означает, что функция может работать с указателем любого типа. Функция заполняет значением `c` первые `n` байт, начиная с адреса `s`.

Инициализация массива из 10 букв 'z' выглядит следующим образом:

```
char s[10];
memset(s, 'z', 10);
```

Обнуление первых 70 элементов массива из 100 чисел может записать так:

```
int d[100];
memset(d, 0, 70 * sizeof(int));
```

9.20 Справочная система man

При необходимости оперативного получения подробной информации о той или иной функции можно использовать справочную систему, которая предустановлена в Ubuntu. Для этого достаточно набрать в терминале:

```
$ man string
```

На экране появится полный список функций для работы со строками. Также можно запросить информацию по конкретной функции:

```
$ man strlen
```

Выход из справки осуществляет по нажатию клавиши **q**.

9.21 Примеры

Пример 3. Дана строка, состоящая не более чем из 100 символов. Необходимо найти и вывести первую и последнюю цифру в строке. Если цифр нет, то вывести два символа 'x'.

Ввод	Astana 2019!!!	Astana
Вывод	2 9	x x

Первую цифру необходимо искать при просмотре строки слева направо. Как только будет обнаружена первая цифра, просмотр следует завершить. Для поиска последней цифры достаточно просмотреть всю строку, постоянно записывая в одну и ту же переменную текущий символ, если он является цифрой.

```
1  #include <stdio.h>
2  #include <ctype.h>
3  int main()
4  {
5      char a[100], ch1 = 'x', ch2 = 'x';
6      gets(a);
7      for (i = 0; a[i] != '\0'; i++)
8          if (isdigit(a[i]))
9              {
10                 ch1 = a[i];
11                 break;
12             }
13     for (i = 0; a[i] != '\0'; i++)
14         if (isdigit(a[i]))
15             ch2 = a[i];
16     printf("%c %c\n", ch1, ch2);
17     return 0;
18 }
```

Пример 4. Дана строка длиной не более 100 символов. Распечатать в алфавитном порядке все буквы и цифры, которые присутствуют в строке.

Дополнительно указать, сколько раз встречается соответствующий символ в строке.

Ввод	mm & cmc 2021!
Вывод	0 (1) 1 (1) 2 (2) c (2) m (3)

```

1  #include <stdio.h>
2  #include <ctype.h>
3  int main()
4  {
5      char s[100];
6      int used[128] = {0}, i;
7      fgets(s, 101, stdin);
8      for (i = 0; s[i]; i++) {
9          int ch = s[i];
10         used[ch]++;
11     }
12
13     for (i = 0; i <= 127; i++)
14         if (isalnum(i) && used[i] > 0)
15             printf("%c (%d)\n", i, used[i]);
16     return 0;
17 }
```

Компилятор выдаёт предупреждение, когда в качестве индексов используется тип `char`. Например, код `used[s[i]]` приводит к предупреждению:

```

prog.c: In function 'main':
prog.c:9:13: warning: array subscript has type 'char'
      [-Wchar-subscripts]
      used[s[i]]++;
```

Пример 5. Даны две строки s_1 и s_2 , каждая длиной не более 100 символов. Найти и вывести их общий суффикс максимальной длины. Суффикс строки — это подстрока, включающая последний символ строки.

Ввод	geometry country	likes like	hello hello
Вывод	try	like	hello

```

1  #include <stdio.h>
2  #include <string.h>
3  int main()
4  {
5      char a[101], b[101];
6      int na, nb, i;
7      gets(a);
8      gets(b);
9      na = strlen(a);
10     nb = strlen(b);
11     i = 0;
12     while (i < na && i < nb &&
13           a[na - i - 1] == b[nb - i - 1])
14     {
15         i++;
16     }
17     puts(a + na - i);
18     return 0;
19 }

```

Пример 6. Дано целое число от 1 до 10^{100} . Умножить его на 2.

Ввод	123456890123456890123456890
Вывод	246913780246913780246913780

Основная идея заключается в умножении столбиком. Но, записывая числа в столбик, стоит помнить, что действия мы производим от младших разрядов к старшим. А это, в свою очередь, удобно делать при увеличении индексов разрядов, так как в результате количество разрядов может увеличиться. Например, чтобы умножить 674 на 2, запишем цифры числа в обратном порядке:

4	7	6	0
---	---	---	---

Для начала умножим каждую цифру на 2:

8	14	12	0
---	----	----	---

Далее нормализуем цифры числа: если значение $a[i]$ больше 9, то перенесём излишек (целая часть при делении на 10) на следующий разряд $a[i + 1]$, а само значение заменим на последнюю цифру (остаток при делении на 10).

В разряде единиц число не изменяется и остается цифра 8.

8	14	12	0
---	----	----	---

В разряде десятков есть переполнение, которое переносим на сотни, а само значение меняется с 14 на 4.

8	4	13	0
---	---	----	---

В разряде сотен тоже есть переполнение, которое переносим на тысячи, а само значение меняется с 13 на 3.

8	4	3	1
---	---	---	---

В разряде тысяч переполнения нет. Ответ получен в перевёрнутом виде.

В общем виде для получения ответа необходимо продолжать данные действия до тех пор, пока есть ненулевые цифры, но не меньше чем длина исходного числа.

```
1  #include <stdio.h>
2  #include <string.h>
3  int main()
4  {
5      int i, n;
6      char a[102], ch;
7
8      scanf("%s", a);
9      n = strlen(a);
10
11     for (i = 0; i < n / 2; i++)
12     {
13         ch = a[i];
14         a[i] = a[n - 1 - i];
15         a[n - 1 - i] = ch;
16     }
17     for (i = 0; i < n; i++)
18         a[i] -= '0';
19
20     for (i = 0; i < n; i++)
21         a[i] *= 2;
22     carry = 0;
23     for (i = 0; i < n || a[i] > 0; i++)
24     {
25         a[i + 1] += a[i] / 10;
26         a[i] = a[i] % 10;
```

```

27     }
28     n = i;
29
30     for (i = 0; i < n; i++)
31         a[i] += '0';
32     for (i = 0; i < n / 2; i++)
33     {
34         ch = a[i];
35         a[i] = a[n - 1 - i];
36         a[n - 1 - i] = ch;
37     }
38     puts(a);
39     return 0;
40 }

```

Пример 7. Дано 2 целых числа a и b от 1 до 10^{100} (каждое на отдельной строке). Найти их произведение.

Ввод	123 67	1234567890 9876543210
Вывод	8241	12193263111263526900

Результат будем записывать в третий массив c , размер которого не более 200 цифр.

Каждое из чисел представим в виде многочлена с коэффициентами равными цифрам числа. Например, числа 123 и 67 соответствуют следующим многочленам:

$$A(x) = 1 \cdot x^2 + 2 \cdot x + 3$$

$$B(x) = 6 \cdot x + 7$$

При $x = 10$ получится разложение числа в 10-чной системе счисления по разрядам.

Перемножим соответствующие многочлены. Например:

$$\begin{aligned}
 C(x) &= A(x)B(x) = (1 \cdot x^2 + 2 \cdot x + 3) \cdot (6 \cdot x + 7) = \\
 &= (1 \cdot 6) \cdot x^3 + (1 \cdot 7 + 2 \cdot 6) \cdot x^2 + (2 \cdot 7 + 3 \cdot 6) \cdot x + (3 \cdot 7) = \\
 &= 6 \cdot x^3 + 19 \cdot x^2 + 32 \cdot x + 21
 \end{aligned}$$

В общем виде k -й коэффициент результата вычисляется следующим образом:

$$c_k = \sum_{i=0}^k a_i b_{k-i}.$$

Чтобы найти произведение исходных чисел достаточно вычислить

$$C(10) = A(10) \cdot B(10).$$

То есть искоемое число, представленное по разрядам в десятичной системе счисления:

$$C(10) = 6 \cdot 10^3 + 19 \cdot 10^2 + 32 \cdot 10 + 21$$

Но некоторые «цифры» больше 9. Для исправления переполненных разрядов проведём аналогичную нормализацию числа, аналогичную той, что была в предыдущей задаче:

единицы	21	32	19	6
десятки	1	34	19	6
сотни	1	4	22	6
тысячи	1	4	2	8

Ответ: 8241.

За основу можно взять программу из предыдущего примера. Далее изменить процесс умножения:

```
int c[201] = {0};
int a[101], b[101];
int na, nb, nc;
...
for (i = 0; i < na; i++)
    for (j = 0; j < nb; j++)
        c[i + j] += a[i] * b[j];
...
```

и нормализации результата (число разрядов должно быть не менее $na + nb - 1$):

```
for (i = 0; i < na + nb - 1 || a[i] > 0; i++)
{
    a[i + 1] += a[i] / 10;
    a[i] = a[i] % 10;
}
nc = i;
```


9.22 Задания для самостоятельной работы

1. Считать строку длиной не более 100 символов. Найти её длину и распечатать вторую половину строки. При нечётной длине — округлить в большую сторону.

Ввод	BigBen	Tower
Вывод	Ben	wer

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str[100];
    int n;

    return 0;
}
```

2. На ввод программе передаются две строки. Что выведет программа для произвольных двух строк *a* и *b*? Какой максимальной длины могут быть строки?

```
#include <stdio.h>
int main()
{
    char a[101], b[101];
    int i;
    gets(a);
    gets(b);
    for (i = 0; a[i] && b[i]; i++) {
        if (a[i] != b[i])
            break;
    }
    printf("%d", i);
    return 0;
}
```

3. Что будет напечатано?

```
int a[5] = {10, 20, 30, 40, 50};
int *p = a + 1;
printf("%d", *p);
(*p)++;
printf("%d", *p);
*p++;
printf("%d", *p);
```

4. Могут ли в коде программы встретиться следующие подстроки:

- а) `*==*`
- б) `/=*`
- в) `a/*`
- г) `a/ *`

Если могут, то приведите примеры, в противном случае — опишите причины, почему не могут.

5. Что будет напечатано:

```
#include <stdio.h>
#include <string.h>
int main() {}
    char a[] = "mechanics";
    char b[] = "mathematics";
    puts(strncpy(strstr(a, "a"), b, 4));
    puts(a);
    puts(b);
    if (strcmp(a + 4, b) > 0)
        puts("OK")
    else
        puts("NO");
    return 0;
}
```

9.23 Практикум на ЭВМ

1. Дана строка длиной не более 100 символов. Посчитать, чему равна длина строки и сколько раз в строке встречается последний символ (строчные и заглавные буквы считаются различными символами).

Ввод	This is Astana	Yohoho!!!	35 below zero
Вывод	14 2	9 3	13 2

2. Дана строка длиной не более 100 символов. Заменить все буквы на строчные.

Ввод	This is Astana
Вывод	this is astana

3. Дана строка длиной не более 100 символов. Посчитать количество слов. Слово — непрерывная последовательность цифр и/или букв.

Ввод	This is Astana-2018
Вывод	4

4. Дана строка длиной не более 100 символов. Посчитать длину первого и последнего слова. Слово — непрерывная последовательность цифр и/или букв.

Ввод	This is Astana	Yohoho	35 below zero
Вывод	4 6	9 9	2 4

5. Дано 2 целых числа a и b от 1 до 10^{100} (каждое на отдельной строке). Найти и вывести их сумму.

Ввод	999	1234567890987654321
	1	123456789
Вывод	1000	1234567891111111110

6. Дано целое положительное число n от 1 до 10^{100} и целое положительное число m от 1 до 1000. Найти остаток от деления n на m (схема Горнера).

Ввод	2019	1002003003002001
	100	13
Вывод	19	0

7. Даны 2 строки s_1 и s_2 длиной не более 100 символов каждая. Найти все позиции подстрок s_1 равные s_2 .

Ввод	She sells seashells on the seashore of Seychelles ell
Вывод	6 16 45

8. Даны две строки s_1 и s_2 каждая длиной не более 100 символов. Найти длину общего суффикса.

Ввод	This is Sparta! This is Parta!
Вывод	5

9. Даны 2 строки s_1 и s_2 длиной не более 100 символов каждая. Строки состоят только из букв английского алфавита. Соответствующие строчные и заглавные буквы считаются одинаковыми. Вывести, какая строка идет лексикографически раньше. Вывести 'first', 'second' или 'both'.

Ввод	Alabama Wyoming	Alaska aLASKA	Colorado color
Вывод	first	both	second

10. Даны 2 строки a и b длиной от 1 до 1000. Найти длину наибольшей общей подстроки.

Алгоритм: построить матрицу d , где значение $d[i][j]$ равно длине наибольшей общей подстроки строк $a[1..i]$ и $b[1..j]$. Если $a[i] = b[j]$, то

$$d[i][j] = \max(d[i][j - 1], d[i - 1][j], d[i - 1][j - 1] + 1),$$

в противном случае

$$d[i][j] = \max(d[i][j - 1], d[i - 1][j]).$$

Ввод	lighthouse housekeeper	baobabs baby	hakunamatataha batatas
Вывод	5	3	5

10 Функции

В данной теме будут рассмотрены вопросы реализации собственных функций и особенности работы с ними. В частности, передача аргумента функции по значению и по указателю, а также передача в качестве аргумента массивов и матриц.

10.1 Собственная функция

В предыдущих темах были рассмотрены примеры использования готовых функций: для математических вычислений (`double sqrt(double)`), для обработки символов (`int isalpha(int)`) и для работы со строками (`int strcmp(char *, char *)`). А основная функция `int main()` встречается во всех программах.

Программируя на языке C, можно не только использовать готовые функции, но и создавать свои. Основные цели создания собственных функций: многократное использование одного и того же участка кода и повышение читаемости кода.

Чтобы создать собственную функцию, необходимо описать две основные части функции:

1. прототип — описание входных и выходных параметров функции;
2. реализация — код функции.

10.2 Прототип функции

В этом примере описан прототип функции, которая на вход принимает одно вещественное число типа `double`, а возвращает значение типа `int`:

```
int f(double);
```

Функция может принимать несколько аргументов. Ниже приведен пример прототипа функции, которая принимает три целых числа и возвращает одно целое число:

```
int g(int, int, int);
```

В общем виде прототип можно описать следующим образом:

```
type function(type1, type2, ...);
```

Имя функции **function** идентифицирует функцию. Правила формирования такие же, как и у переменных: любой набор уникальных букв, цифр и символа подчёркивания. Имя функции должно отличаться от уже объявленных функций, системных функций и переменных.

Тип возвращаемого значения **type** соответствует множеству значений данной функции. Возвращаемый тип не является обязательным — вместо типа можно написать служебное слово **void**, которое означает «ничего».

Типы аргументов **type1**, **type2** и т.д. соответствует множеству определения функции. Аргументы не являются обязательными: вместо них можно вставить слово **void** или просто оставить пустое место в скобках.

Прототип функции должен быть описан до первого места, где функция вызывается. Как правило, после подключения библиотек и перед основной функцией **int main()**. Пример прототипа функции с одним аргументом и вызовом:

```
int f(double x);

int main()
{
    int a, b;
    double y = 2.5;
    a = f(y);
    b = f(2.7);
    return 0;
}
```

Пример прототипа двух функций с несколькими аргументами:

```
int powInt(int base, int power);
void printArray(int n, int a[100]);

int main()
{
    int a = powInt(3, 5);
    int s[100] = {0};
    print(5, s);
    return 0;
}
```

Пример с функцией без аргументов, с функцией без возвращаемого значения и с функцией без того и другого:

```
double getDouble(void);
```

```
void hello(int);  
void make(void);  
  
int main()  
{  
    double x = getDouble();  
    hello(5);  
    make();  
    return 0;  
}
```

10.3 Тело функции и возвращаемое значение

При попытке скомпилировать примеры выше появится ошибка линковки:

```
$ gcc prog.c -o prog
In function `main':
prog.c:(.text+0x24): undefined reference to 'f'
```

Синтаксических ошибок в коде нет, но создать исполняемый файл не получилось, так как компилятор не нашел реализацию функции. Реализацию функции можно выполнить двумя способами: вместе с прототипом или отдельно от него.

Первый вариант выглядит следующим образом:

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

Обратите внимание на то, что после прототипа точка с запятой не ставится.

Второй вариант выглядит так:

```
int f(double x);

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

Оператор **return** описывает, какое значение вернёт функция. Тип должен соответствовать типу возвращаемого значения (в указанном примере — `int`). Это может быть константа, переменная, выражение, результат вычисления другой функции или даже этой же самой (о таком необычном варианте пойдёт речь в теме «Рекурсия»). Оператор **return** завершает выполнение функции, то есть действия внутри функции после него не будут выполняться.

Пример реализации функции `inc10`, которая для данного целого числа n возвращает значение $n + 10$, выглядит следующим образом:

```
int inc10(int n)
```



```
{
    return n + 10;
}

int main()
{
    int b = inc10(25);
    return 0;
}
```

Функция, которая вычисляет длину двумерного вектора $(x; y)$:

```
double length(double x, double y)
{
    return sqrt(x * x + y * y);
}

int main()
{
    double a = 3, b = 4, l;
    l = length(3, 4);
    l = length(a, y);
    l = length(a, 4);
    l = length(3, b);
    return 0;
}
```

Пример 1. Даны два целых числа a и b от 1 до 1000. Определить, у какого из чисел количество делителей больше. Вывести это число и количество его делителей. Описать вспомогательную функцию, которая возвращает количество делителей данного числа.

Ввод	10 25	11 25	12 18
Вывод	10 4	25 3	12 6 18 6

```
1 #include <stdio.h>
2
3 int tau(int n)
4 {
5     int ans = 0, d;
6     for (d = 1; d <= n; d++)
```

```

7         if (n % d == 0)
8             ans++;
9     return ans;
10 }
11
12 int main()
13 {
14     int a, b, tau_a, tau_b;
15     scanf("%d %d", &a, &b);
16     tau_a = tau(a);
17     tau_b = tau(b);
18     if (tau_a >= tau_b)
19         printf("%d %d\n", a, tau_a);
20     if (tau_b >= tau_a)
21         printf("%d %d\n", b, tau_b);
22     return 0;
23 }

```

В нашем курсе мы будем придерживаться ориентира в 20 строк на каждую функцию. Если функция, в том числе и `main`, становится длиннее 20 строк, рекомендуется задуматься о её «декомпозиции», т.е. выделить некоторую логическую часть функции в отдельную функцию. Данный подход имеет сразу 2 полезных свойства:

1. упрощение процесса поиска ошибок (можно тестировать функции поочередно);
2. использование кода для других программ.

10.4 Функции без аргументов

Пример 2. Даны два вещественных числа с одним знаком после запятой. Причём разделителем целой и дробной части является запятая. Найти сумму этих чисел. Описать функцию для ввода вещественного числа числа с разделителем в виде запятой.

Ввод	23,4 100,2
Вывод	123.6

```

1  #include <stdio.h>
2
3  double getDouble()
4  {
5      int a, b;
6      scanf("%d,%d",&a, &b);
7      return a + 0.1 * b;
8  }
9
10 int main()
11 {
12     double x, y;
13     x = getDouble();
14     y = getDouble();
15     printf("%lf\n", x + y);
16     return 0;
17 }
```

10.5 Функции без возвращаемого значения

Функции без возвращаемого значения часто называют процедурами. Основная цель функции — выделить повторяющийся участок кода. Так как функция не возвращает никаких значений, то оператор **return** можно опустить.

Пример 3. Дано два целых числа n и m от 1 до 10. Вывести текст «hello» n раз в первой строке и m раз — на второй строке.

Ввод	2 4
Вывод	hello hello hello hello hello hello

```

1  #include <stdio.h>
2
3  void hello(int n)
4  {
5      int i;
6      for (i = 0; i < n; i++)
7          printf("hello ");
8      putchar("\n");
```

```

9  }
10
11 int main()
12 {
13     int n, m;
14     scanf("%d %d", &n, &m);
15     hello(n);
16     hello(m);
17     return 0;
18 }

```

10.6 Несколько операторов return

Функция может содержать несколько операторов `return`. Как правило, это встречается внутри условного оператора или цикла.

Разберем пример функции, которая возвращает знак вещественного числа x :

```

int sign(double x)
{
    if (x > 0)
        return 1;
    if (x < 0)
        return -1;
    return 0;
}

```

При вызове функции `f(4.2)` сработает первый оператор `return 1`; и код далее выполняться не будет. При вызове функции `f(-4.2)` первый условный оператор даёт «ложь» и сработает второй оператор `return -1`;, и код далее выполняться не будет. При вызове функции `f(0.0)` ни один из условных операторов не выполнится, соответственно, вернется `return 0`;

Рассмотрим ещё один пример функции, которая возвращает максимум из двух чисел:

```

int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

```

```
}
```

При компиляции будет получено следующее предупреждение:

```
$ gcc prog -o prog -Wall
Warning: no return value by default
```

Проблема заключается в том, что рекомендуется всегда использовать явный оператор `return` (не вложенный в условный оператор или цикл) для функций с возвращаемым типом. Решить её можно следующим образом:

```
int max(int a, int b)
{
    if (a > b)
        return a;
    return b;
}
```

10.7 Использование результатов вычисления функции

В качестве аргументов функции можно передавать результаты вызова этой же или другой функции. Ниже приведён пример, демонстрирующий такую ситуацию:

```
int max(int a, int b)
{
    if (a > b)
        return a;
    return b;
}
int max3(int a, int b, int c)
{
    return max(max(a, b), c);
}
```

Чтобы найти максимум из 3 чисел, используется функция максимума 2 чисел:

10.8 Приведение типа аргумента

Если аргумент функции не соответствует заявленному в прототипе, то компилятор осуществит попытку неявного преобразования типа:

```
int f(double x);
int main()
{
    int x, y = 2;
    x = f(y);
    return 0;
}
```

В данном примере вызов `f(2)` преобразуется к вызову `f(2.0)`.

Если неявное преобразование недопустимо (например, `(int *) → double`):

```
x = f(&y);
```

то при компиляции возникнет следующая ошибка:

```
$ gcc prog.c -o prog -Wall
error: incompatible type for argument 1 of 'f'
    x = f(&y);
        ^
```

Аналогичную ошибку наблюдали те, кто забывал ставить оператор адреса в функции `scanf`.

```
int x;
scanf("%d", x);
```

10.9 Локальные переменные

Каждая функция может иметь свои переменные, которые называются локальными. Время «жизни» этих переменных ограничено временем работы данного вызова функции. То есть при каждом новом вызове функции такие переменные будут заново создаваться, а при выполнении оператора `return` уничтожаться:

```
int next(int a)
{
    int ans = a + 1;
    return ans;
}

int main()
{
```

```

    int x, y;
    x = 5;
    y = next(x);
    x = 10;
    y = next(x);
    return 0;
}

```

В данном примере переменная **ans** является локальной для функции **next**, а переменные **x** и **y** — локальными переменными функции **main**. Два важных замечания:

1. при каждом из двух вызовов функции **next** будет создана её локальная переменная **ans**, и после каждого возврата из функции (оператор **return**) — уничтожена;
2. функция **next** не имеет прямого доступа к переменным **x** и **y**, а функция **main** — к переменной **ans**.

Также стоит отметить, что аргументы функции тоже являются локальными переменными этой функции. Отличие от локальных переменных, определённых внутри функции, в том, что они инициализируются значениями из вызова функции. То есть, вызов **y = next(5)** приводит к вызову функции **next** с двумя локальными переменными **a** и **x**:

```

    int a = 5;
    int x = a + 1;
    return x;

```

В двух разных функциях можно использовать переменные с одинаковым именем абсолютно независимо друг от друга:

```

int next(int a)
{
    int x = a + 1;
    return x;
}

int main()
{
    int x, y;
    x = 5;
    y = next(x);
    return 0;
}

```

```
}

```

В данном примере переменной `x` функции `main` присвоено значение 5. Далее при вызове `next` будет создана локальная переменная `x` со значением $5 + 1 = 6$ и уничтожена после возврата в функцию `main`. При этом локальная переменная `x` функции `main` сохранила своё значение 5.

10.10 Глобальные переменные

Язык C позволяет объявлять глобальные переменные, которые существуют от начала и до конца выполнения всей программы. К ним можно получить доступ из всех функций. Отличительной особенностью глобальных переменных является то, что они по умолчанию инициализированы нулевыми значениями:

Глобальные переменные могут создавать значительные трудности при использовании одинакового имени у глобальной и локальной переменной функции:

```
1  #include <stdio.h>
2
3  int x;
4
5  void next()
6  {
7      x = 7
8  }
9
10 int main()
11 {
12     int x;
13     x = 5;
14     next();
15     printf("%d", x);
16     return 0;
17 }
```

В примере глобальная переменная `x` равна 7, а локальная переменная `x` в функции `main` равна 5. На выводе появится: 5.

Если убрать всего одну строку:

```
1  #include <stdio.h>
2
3  int x;
```



```

4
5 void next()
6 {
7     x = 7
8 }
9
10 int main()
11 {
12     x = 5;
13     next();
14     printf("%d", x);
15     return 0;
16 }

```

то программа выдаст: 7.

В большой программе сложно отловить ошибку такого рода. Также придётся помнить актуальное значение глобальной переменной, которую изменяют несколько функций. В рамках данного курса рекомендуется отказаться от использования любых глобальных переменных.

10.11 Передача аргументов по значению и по указателю

Попробуем описать функцию, которая изменяет значение своего аргумента, как, например, функция `scanf`. Передача аргумента по значению не приведёт успеху:

```

void next(int d) {
    d = d + 1;
}

int main() {
    int a = 7;
    next(a);
    printf("%d", a);
    return 0;
}

```

В данном примере в функцию `next` передается аргумент по значению. Вызов функции схематически работает так:

```

void next() {
    int d = 7;
}

```

```
d = d + 1;
}
```

Выполним все действия по шагам:

стр	main()	<i>a</i>	стр	next(int)	<i>d</i>
6	int a = 7;	7			
7	next(a);	7			
		7	1	int d = 7;	7
		7	2	d = d + 1;	8
8	printf("%d ", a);	7			

В итоге на экране будет число 7. То есть, изменить переменную не удалось.

Воспользуемся косвенным изменением значения переменной, передавая её адрес:

```
void next(int *p)
{
    *p = *p + 1;
}

int main()
{
    int a = 7;
    next(&a);
    printf("%d", a);
    return 0;
}
```

В переменной *p* окажется адрес переменной *a* (допустим это ..20). Выполним все действия по шагам:

стр	main()	<i>a</i>	стр	next(int)	<i>p</i>
6	int a = 7;	7			
7	next(&a);	7			
		7	1	int *p = ..20;	..20
		8	2	*p = *p + 1;	..20
8	printf("%d ", a);	8			

В итоге на экране окажется число 8.

Таким образом, передача аргумента по указателю позволяет изменить значение локальной переменной из другой функции.

10.12 Массив как аргумент функции

Рассмотрим пример, где в качестве аргумента функции передаётся массив.

```

1  #include <stdio.h>
2
3  void set(double b[5])
4  {
5      b[2] = 0.0;
6  }
7
8  int main()
9  {
10     double a[5] = {1.1, 2.2, 3.3, 4.4, 5.5}, i;
11     set(a);
12     for (i = 0; i < 5; i++)
13         printf("%d ", a[i]);
14     return 0;
15 }
```

На выводе будут значения: 1.1, 2.2, 0.0, 4.4, 5.5. То есть, массив `a` изменился. Что же произошло? Посмотрим на адрес и размер переменных `a` и `b`:

```

1  #include <stdio.h>
2
3  void set(double b[5])
4  {
5      printf("b: %p %lu\n", b, sizeof(b));
6  }
7
8  int main()
9  {
10     double a[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
11     printf("a: %p %lu\n", a, sizeof(a));
12     set(a);
13     return 0;
14 }
```

На выводе:

```

a: 0x10002000 40
b: 0x10002000 4
```

Теперь ясно, что **a** — это массив на 40 байт, а **b** — это лишь указатель на начало этого массива, то есть **double *b = a**. Соответственно, любые изменения массива через указатель **b** равносильны изменениям массива **a**. Таким образом, все три прототипа функций:

```
void set(double b[5]);
void set(double b[]);
void set(double *b);
```

приводят к одинаковому результату: передачи указателя (размер массива в первом примере игнорируется). Из соображений читаемости кода лучше использовать второй вариант, чтобы можно было отличить передачу указателя на число от передачи массива.

Для практического закрепления опишем функции для ввода и вывода массива:

```
1  #include <stdio.h>
2
3  int scan_array(int a[])
4  {
5      int i, n;
6      scanf("%d", &n);
7      for (i = 0; i < n; i++)
8          scanf("%d ", &a[i]);
9      return n;
10 }
11
12 void print_array(int n, int a[])
13 {
14     int i;
15     for (i = 0; i < n; i++)
16         printf("%d ", a[i]);
17 }
18
19 int main()
20 {
21     int x[100], n;
22     n = scan_array(x);
23     print_array(n, x);
24     return 0;
25 }
```

Передачу аргумента функции по указателю мы будем использовать для трех целей:

1. для изменения значения внешней переменной;
2. для работы с массивами и матрицами;
3. для экономии памяти.

10.13 Матрица как аргумент функции

Рассмотрим пример передачи матрицы в качестве аргумента функции:

```
void set(double b[2][3])
{
    b[0][1] = 0.0;
}

int main()
{
    double a[2][3] = {
        {1.0, 2.0, 3.0},
        {4.0, 5.0, 6.0}
    };
    set(a);
    return 0;
}
```

В этом случае будет передана не матрица, а указатель на начало матрицы. Определим тип данного указателя. Если передать массив из 5 чисел типа `double`, то получим `(double *)` — указатель на тип `double`. А что такое матрица? `double [2][3]` — это массив из 2 элементов, каждый из которых типа `double[3]`. Значит, этот тип будет преобразован к `double (*)[3]` — указателю на массив из 3 чисел типа `double`. То есть, матрицу можно принимать по одному из следующих описаний:

```
void set(double b[2][3]);
void set(double b[][3]);
void set(double (*b)[3]);
```

В последнем варианте скобки обязательны, чтобы не путать с типом `double *b[3]` — массив из 3 элементов типа `double *`. Последние 2 варианта прототипа позволяют передавать матрицы размера $n \times 3$ для любого n . Например:

```
void set(double b[][3]);
...
double a[2][3], c[5][3], d[7][3];
set(a);
set(c);
set(d);
```

Стоит обратить внимание, что если попытаться передать матрицы с другой второй размерностью:

```
void set(double b[][3]);
...
double a[2][5];
set(a);
```

то мы получим ошибку:

```
$ gcc prog -o prog
expected 'double (*)[3]' but argument is
of type 'double (*)[5]'
```

Можно ли полностью избавиться от второй размерности? Существует несколько способов для такой реализации.

При использовании статических массивов можно либо принимать указатель произвольного типа `void *` и преобразовывать его к нужному типу, либо использовать одномерный массив, где элементы разложены по строкам вручную. Второй вариант приведёт к необходимости замены двойной индексации на одинарную: вместо `a[i][j]` используется `a[i * n + j]`.

Для избавления второй размерности можно использовать динамическую память, создавая указатель на массив указателей: `double b**`. Последний тип принципиально отличается от рассмотренных выше `double (*)[3]` и подробнее будет рассмотрен в теме «Динамические матрицы».

10.14 Макроопределение типа

Для сокращения кода можно использовать макроопределения типов с помощью макроса `typedef`. Например, длинное определение `unsigned long long` можно заменить на короткое `int64`:

```
typedef unsigned long long int64;

int main()
```

```
{
    int64 a, b, c;
    return 0;
}
```

Рассмотрим ещё один пример макроопределения типа. Введём определение типа для массива и вектора фиксированного размера, предположив, что реальный размер матрицы и вектора не превосходит 100:

```
typedef double Vector[100];
typedef double Matrix[100][100];

void f(int n, Vector a);
void g(int n, int m, Matrix a);

int main() {
    Vector a;
    Matrix b;
    f(2, a);
    g(2, 5, b);
    return 0;
}
```

10.15 Примеры

Пример 4. Дано целое число n от 0 до 10^9 . Если число n простое, то вывести «prime». Если число составное — «composite». Если число не является ни простым, ни составным, то вывести «no type». Описать функцию, которая проверяет, является ли целое число n простым или составным и возвращает результат: 1 — простое, 0 — составное, −1 — иное.

```
int isPrime(int n);
```

Ввод	2017	2018	1
Вывод	prime	composite	no type

Числа, которые не являются ни простыми, ни составными ($n < 2$), легко выявить сразу. Для остальных чисел переберём все потенциальные делители — это числа от 2 до $n - 1$. Если хотя бы одно из них является делителем, то дальнейший перебор можно прекратить.

```
1  #include <stdio.h>
2
3  int isPrime(int n)
4  {
5      int d;
6      if (n < 2)
7          return -1;
8      for (d = 2; d < n; d++)
9          if (n % d == 0)
10             return 0;
11     return 1;
12 }
13
14 int main()
15 {
16     int n, answer;
17     scanf("%d", &n);
18     answer = isPrime(n);
19     if (answer == 1)
20         puts("prime");
21     else if (answer == 0)
22         puts("composite");
23     else
24         puts("nothing");
25     return 0;
26 }
```

Каждый вызов и выполнение функции требуют дополнительных расходов памяти и времени. Поэтому не стоит делать вызов одной и той же функции, если этого можно избежать. Например, ни в коем случае не стоит писать такой код:

```
if (isPrime(n) == 1)
    puts("prime");
if (isPrime(n) == 0)
    puts("composite");
if (isPrime(n) == -1)
    puts("nothing");
```

В таком коде одинаковые вычисления выполняются три раза, что крайне неэффективно.

Любители теории чисел могут отметить, что перебор можно сократить до \sqrt{n} :

```
for (d = 2; d <= sqrt(n); d++)
    if (n % d == 0)
        return 0;
```

Теперь для $n = 10^6$ число проверок будет порядка 1000. Но здесь опять же проявляется проблема, описанная выше: функция `sqrt(n)` с одним и тем же результатом будет вызываться 1000 раз. От данной проблемы можно избавиться двумя способами:

1. вычислить и сохранить значение \sqrt{n} перед циклом:

```
int dmax = sqrt(n);
for (d = 2; d <= dmax; d++)
    if (n % d == 0)
        return 0;
```

2. вместо условия $d < \sqrt{n}$ проверять условие $d^2 < n$, так как любая простая арифметическая операция выполняется быстрее, чем вычисление функции:

```
for (d = 2; d * d <= n; d++)
    if (n % d == 0)
        return 0;
```

Пример 5. На первой строке дано целое число n от 1 до 100. На второй строке — последовательность их n целых чисел. Найти максимальное значение последовательности и позиции всех элементов, которые ему равны. Описать функции для ввода массива, поиска максимума и вывода позиций:

```
int scanArray(int n);
int findMax(int n, int a[]);
void printPosition(int n, int a[], int value);
```

Ввод	5 1 5 5 4 5
Вывод	5: 2 3 5

```
1  #include <stdio.h>
2
3  int scanArray(int a[])
4  {
5      int i, n;
6      scanf("%d", &n);
7      for (i = 0; i < n; i++)
8          scanf("%d ", &a[i]);
9      return n;
10 }
11
12 int findMax(int n, int a[])
13 {
14     int i, max = a[0];
15     for (i = 1; i < n; i++)
16         if (max < a[i])
17             max = a[i];
18     return max;
19 }
20
21 void printPosition(int n, int a[], int value)
22 {
23     for (i = 0; i < n; i++)
24         if (value == a[i])
25             printf("%d ", i + 1);
26     putchar('\n');
27 }
28
29 int main()
30 {
31     int a[100], n, max;
32     n = scanArray(a);
33     max = findMax(n, a);
34     printf("%d: ", max);
35     printPosition(n, a, max);
36     return 0;
37 }
```

Пример 6. Дано целое число n от 2 до 1000. Для всех вычетов, обратимых по модулю n , построить и вывести обратные элементы. То есть, для всех

a от 1 до $n - 1$ вывести b такое, что $ab \equiv 1 \pmod n$. Описать функции для поиска обратного элемента и вывода ответа:

```
int reverse(int n, int a);
void printAllReverse(int n);
```

Ввод	10	7
Вывод	1 1	1 1
	3 7	2 4
	7 3	3 5
	9 9	4 2
		5 3
		6 6

```
1  #include <stdio.h>
2
3  int reverse(int n, int a)
4  {
5      int b;
6      for (b = 1; b < n; b++)
7          if (a * b % n == 1)
8              return b;
9      return 0;
10 }
11
12 void printAllReverse(int n)
13 {
14     int a, b;
15     for (a = 1; a < n; a++)
16     {
17         b = reverse(n, a);
18         if (b > 0)
19             printf("%d %d\n", a, b);
20     }
21 }
22
23 int main()
24 {
25     int n;
26     scanf("%d", &n);
27     printAllReverse(n);
```

```
28     return 0;
29 }
```

Пример 7. Дано целое число n от 2 до 10^6 . Вычислить $\varphi(n)$ — количество натуральных чисел меньших n и взаимнопростых с n . Описать функции для вычисления наибольшего общего делителя двух чисел и для вычисления ответа:

```
int gcd(int a, int b);
void phi(int n);
```

Ввод	10	7
Вывод	4	6

```
1  #include <stdio.h>
2
3  int gcd(int a, int b)
4  {
5      while (b != 0)
6      {
7          t = a % b;
8          a = b;
9          b = t;
10     }
11     return a;
12 }
13
14 int phi(int n)
15 {
16     int answer = 0, a;
17     for (a = 1; a < n; a++)
18         if (gcd(a, n) == 1)
19             answer++;
20     return answer;
21 }
22
23 int main()
24 {
25     int n;
26     scanf("%d", &n);
```

```

27     printf("%d\n", phi(n));
28     return 0;
29 }

```

Пример 8. Дано целое число n от 1 до 100. Далее матрица $n \times n$ из различных целых чисел от -1000 до 1000 . Необходимо найти минимальный элемент матрицы и его позицию. Описать функции для ввода матрицы (возвращает размер матрицы) и для поиска минимума (возвращает минимум по значению, а позицию через указатели).

```

int scanMatrix(int a[][100]);
int findMin(
    int n,
    int a[][100],
    int *ptri,
    int *ptrj
);

```

Ввод	3 3 4 5 6 2 1 9 8 7
Вывод	1: 2 3

Если у функции часть аргументов входные, а часть аргументов — указатели на выходные параметры, то сначала принято перечислять входные параметры, а потом — выходные, как в данном примере:

```

1  #include <stdio.h>
2
3  typedef int Matrix[100][100];
4
5  int scanMatrix(Matrix a)
6  {
7      int i, j, n;
8      scanf("%d", &n);
9      for (i = 0; i < n; i++)
10         for (j = 0; j < n; j++)
11             scanf("%d", a[i][j]);
12     return 0;
13 }

```

```

14
15 int findMin(int n, Matrix a, int *ptri, int *ptrj)
16 {
17     int i, j, n, min;
18     min = a[0][0];
19     index = 0;
20     for (i = 0; i < n; i++)
21         for (j = 0; j < n; j++)
22             if (min > a[i][j])
23                 {
24                     min = a[i][j];
25                     *ptri = i + 1;
26                     *ptrj = j + 1;
27                 }
28     return min;
29 }
30
31 int main()
32 {
33     int n, i, j, min;
34     Matrix a;
35     n = scanMatrix(a)
36     min = findMin(n, a, &i, &j);
37     printf("%d: %d %d\n", min, i, j);
38     return 0;
39 }

```

10.16 Задания для самостоятельной работы

1. Чему равны значения x , y и z :

```
int f(int a)
{
    if (a % 3 == 0)
        return 0;
    if (a % 3 == 1)
        a = 1;
    if (a % 3 == 2)
        return 2;
    return 3;
}

int main()
{
    x = f(14);
    y = f(13);
    z = f(12);
    return 0;
}
```

2. Чему равны значения локальной переменной a функции `main` и глобальной переменной a после каждой строки функции `main`:

```
int a = 10;
int f(int a)
{
    a = a + 1;
    return a;
}

int g()
{
    a = a + 5;
    return a;
}

int main()
{
    int a = 0;
```

```

    a = f(10);
    a = f(a);
    a = g();
    return 0;
}

```

3. Описать прототипы, которые можно использовать для передачи в качестве аргумента:

1. статического массива

int a[10];

а) void set(int b[10]);

б) void set(double b[10]);

в) void set(int b[20]);

г) void set(int b[5]);

д) void set(int b);

е) void set(int b[]);

ж) void set(unsigned int b[]); ж) void set(int (*d)[3]);

з) void set(int *b[]);

и) void set(int *b);

2. статической матрицы

int c[2][3];

а) void set(int d[2][3]);

б) void set(int d[2][5]);

в) void set(int d[5][3]);

г) void set(int d[][3]);

д) void set(int d[2][]);

е) void set(int *d[3]);

ж) void set(int (*d)[3]);

з) void set(int d[][3]);

и) void set(int **d);

4. В каких случаях значения локальных переменных функции main изменятся?

```

1 void f1(int x)
2 {
3     x = x + 1;
4 }
5 void f2(int x[1])
6 {
7     x[0] = x[0] + 1;
8 }
9 void f3(int *x)
10 {
11     *x = *x + 1;
12 }

```



```
13 int main()
14 {
15     int a = 5, b[1] = {5}, c = 5;
16     f1(a);
17     f2(b);
18     f3(&c);
19     return 0;
20 }
```

5. Написать функции

```
1 int max2(int, int)
2 int max4(int, int, int, int)
```

которые находят максимум из 2 и 4 чисел, соответственно. Реализация второй функции должна содержать вызовы первой функции.

10.17 Практикум на ЭВМ

1. Дано целое положительное число n от 1 до 1000000. Найти сумму цифр числа. Описать функцию:

```
int digitSum(int n);
```

Ввод	1234	2018	1
Вывод	10	11	1

2. Даны 3 вещественных числа a , b , c . Проверить, можно ли составить треугольник с такими сторонами. Если можно, то найти его площадь, иначе — вывести ноль. Описать функцию:

```
double square(double a, double b, double c);
```

Ввод	4.0 5.0 6.0	3.0 10.0 20.0	-3.0 -4.0 -5.0
Вывод	9.9	0.0	0.0

3. Дано целое положительное число n от 1 до 1000000. Найти количество простых чисел, не превосходящих n . Описать функцию, которая возвращает 0, если число простое, 1 — если число составное и -1 — иначе. Описать функцию $\pi(n)$ — количество простых чисел, не превосходящих n .

```
int isprime(int n);
int pi(int n);
```

Ввод	5	100
Вывод	3	25

4. Дано целое положительное число n от 1 до 10000. Найти количество чисел, меньших n взаимно простых с n (функция Эйлера) и вывести эти числа. Описать функции для вычисления наибольшего общего делителя и функции Эйлера:

```
int gcd(int a, int b);
int phi(int n);
```

Ввод	12
Вывод	4 1 5 7 11

5. Дано 2 целых положительных числа a и p от 1 до 10000. Найти обратный элемент к a по модулю p , если такой элемент существует (то есть такое число x , что $x * a = 1 \pmod{p}$), иначе вывести 0. Описать функцию:

```
int inverseElement(int a, int p);
```

Ввод	3 11	10 99	2 10
Вывод	4	10	0

6. Дано целое положительное число p от 1 до 10000. Вывести по возрастанию все квадратичные вычеты q по модулю p (то есть существует такое число x , что $x * x = q \pmod{p}$). Описать функцию, которая возвращает: 1, если q является квадратичным вычетом по модулю p , 0 — иначе:

```
int isQuadraticResidues(int q, int p);
```

Ввод	7	28
Вывод	1 2 4	1 4 8 9 16 21 25

7. На первой строке дано целое положительное n от 1 до 26. На второй строке дана строка из строчных символов, длиной не более 100 символов. Зашифруйте строку из строчных букв шифром Цезаря: каждую букву циклически сдвинуть на n позиций по алфавиту. Опишите функцию, которая изменяет данную строку s (при этом она не должна ничего выводить или считывать):

```
void caesarCrypt(char s[], int n);
```

Ввод	3 hello	23 khoor	1 zoo
Вывод	khoor	hello	app

8. Даны две строки s_1 и s_2 каждая длиной не более 100 символов. Найти длину общего префикса. Описать функцию:

```
int prefix(char a[], char b[]);
```

Ввод	Ice Age 2: The Meltdown Ice Age 3: Dawn of the Dinosaurs
Вывод	8

9. Дано целое положительное число n от 1 до 100. Далее матрица из $n \times n$ целых чисел от -1000 до 1000. Найти след матрицы (сумма диагональных элементов). Описать функцию:

```
int trace(int n, int a[][100])
```

Ввод	2 1 2 3 4
Вывод	5

10. Дано целое положительное число n от 1 до 100. Построить все простые числа, которые не превосходят n с помощью решета Эратосфена (вычеркиваем все числа больше 2 и кратные 2, вычеркиваем все числа больше 3 и кратные 3, вычеркиваем все числа больше 4 и кратные 4 и т.д.). Описать функцию, которая заполняет массив $p[k] = 0$, если k — простое, и $p[k] = 1$, если k — непростое. Описать функцию, которая получает на вход решето Эратосфена и заполняет массив из простых чисел, возвращая его размер.

```
void getSieve(int n, int p[]);  
int filterIndex(int n, int p[], int primes[]);
```

Ввод	20
Вывод	8 3 5 7 11 13 17 19

11 Функция в качестве аргумента.

Рекурсия

В данной теме будет рассмотрен вопрос об использовании функций внутри другой функции, передавая её через аргумент. Также будет затронут вопрос о системном стеке вызовов функций для организации рекурсивных вызовов.

11.1 Функция как аргумент

Рассмотрим задачу, в которой необходимо вычислить 2 суммы:

$$\frac{1}{1^2} + \frac{1}{2^2} + \dots + \frac{1}{100^2}$$
$$\sqrt{1} + \sqrt{2} + \dots + \sqrt{1000}$$

Обе подзадачи можно решить, используя простые циклы, причём они будут весьма похожи друг на друга. Для этого достаточно описать код нахождения суммы

$$f(1) + f(2) + \dots + f(n)$$

для функций $f(x) = \frac{1}{x^2}$ и $f(x) = \sqrt{x}$, передавая функцию в качестве аргумента:

```
double sum(int n, double f(double))
{
    double s = 0, x;
    for (x = 1.0; x <= n; x++)
        s += f(x);
    return s;
}
```

Обратите внимание, что в качестве второго аргумента задаётся прототип функции f . Чтобы вычислить сумму обратных квадратов, нужно описать соответствующую функцию:

```
double reverse(double x)
{
    return 1.0 / (x * x);
}
```

Для извлечения корня будем использовать библиотечную функцию `sqrt`. Теперь можно получить две исходные суммы, используя одну и ту же функцию `sum`:

```
double res1 = sum(100, reverse);  
double res2 = sum(1000, sqrt);
```

Часто в литературе можно найти рекомендацию передавать в качестве аргумента указатель на функцию:

```
double sum(int n, double (*f)(double))  
{  
    double s = 0;  
    int i;  
    for (i = 1; i <= n; i++)  
        s += f(i);  
    return s;  
}  
...  
sum(1000, &sqrt);
```

В действительности разницы никакой нет, так как в любом случае тип приводится к указателю на функцию. Поэтому можно использовать любой из этих вариантов. Единственное достоинство второй записи — напоминание о том, что передаётся именно указатель на функцию.

Для более компактной записи можно использовать макрос для определения прототипа функции:

```
typedef double (*funcptr)(double);  
  
double sum(int n, funcptr f)
```

11.2 Системный стек

Прежде чем переходить к рекурсии, подробно разберём, как выполняется функция. Перед её запуском необходимо запомнить номер строки, к которой необходимо вернуться после завершения функции — адрес возврата. Когда в коде функции есть вызов другой функции, то адресов возврата нужно несколько — каждому вызову свой адрес. Рассмотрим пример:

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

```

int f(int n)
{
    int i, s = 0;
    for (i = 1; i <= n; i++)
        s += g(i);
    return s;
}

int main()
{
    int ans;
    ans = f(3);
    return 0;
}

```

Функция *main* вызывает функцию *f*, которая несколько раз вызывает функцию *g*. Значит, необходимо запоминать сразу несколько адресов: в какой адрес возвращается *f*, вызванная из *main*, и в какой адрес возвращается *g*, вызванная из *f*. Причём адреса возврата нужны в определенном порядке:

1. сохранить адрес возврата *f*(3) к *main*;
 - (a) сохранить адрес возврата *g*(1) к *f*;
 - (b) извлечь адрес возврата *g*(1) и вернуться к *f*;
 - (a) сохранить адрес возврата *g*(2) к *f*;
 - (b) извлечь адрес возврата *g*(2) и вернуться к *f*;
 - (a) сохранить адрес возврата *g*(3) к *f*;
 - (b) извлечь адрес возврата *g*(3) и вернуться к *f*;
2. извлечь адрес возврата *f*(3) и вернуться к *main*.

Для хранения и извлечения адресов в данном порядке подходит структура данных, которая называется «стек»: кто первый сохранён, тот первый извлечён. Для этого операционная система в автоматическом режиме использует системный стек.

Что ещё хранится в системном стеке помимо адресов? Все локальные переменные и аргументы, которые передаются в функцию. Например, при каждом вызове функции *f* в стек укладываются: адрес возврата к *main* (AB), локальные переменные *n*, *i* и *s*. А при вызове функции *g* сохраняются адрес возврата к *g* (AB) и локальная переменная *x*. После завершения функции память под локальные переменные очищается.

функция	код	стек
main()	int main()	
main()	int ans;	ans
f(3)	int i, s = 0;	ans, AB main, n=3, i, s
g(1)	return x * x;	ans, AB main, n=3, i, s, AB f, x=1
f(3)	s += g(1);	ans, AB main, n=3, i, s
g(2)	return x * x;	ans, AB main, n=3, i, s, AB f, x=2
f(3)	s += g(2);	ans, AB main, n=3, i, s
g(3)	return x * x;	ans, AB main, n=3, i, s, AB f, x=3
f(3)	s += g(3);	ans, AB main, n=3, i, s
main()	ans = f(3);	ans
main()	return 0;	

В стек могут укладываться и некоторые другие параметры. Всё, что сохраняется в стек при вызове функции, называется стековым фреймом. Его точный размер предсказать непросто (нужно учитывать и параметры компилятора, и параметры операционной системы), но на практике обычно достаточно учитывать локальные переменные, аргументы функции и адрес возврата с небольшим запасом.

11.3 Рекурсия

Если в коде функции f имеется вызов этой же функции, то такая функция называется рекурсивной. Рекурсивные вычисления очень напоминают метод математической индукции. Для того чтобы описать рекурсивную функцию, необходимы:

1. база рекурсии (нерекурсивное определение);
2. переход рекурсии (рекурсивное определение).

Рассмотрим простой пример, на котором можно продемонстрировать рекурсию: вычисление факториала. Во-первых, определим базу:

$$1! = 1.$$

Во-вторых, определим рекуррентное соотношение между $n!$ и $(n - 1)!$:

$$n! = n \cdot (n - 1)!.$$

Соответствующий код выглядит следующим образом:

```
int factorial(int n)
{
```



```

    if (n == 1)
        return 1;
    int answer = factorial(n-1) * n;
    return answer;
}

```

В случае с рекурсивными вызовами стоит очень внимательно относиться к нерекурсивной ветви. Например, данный код является некорректным:

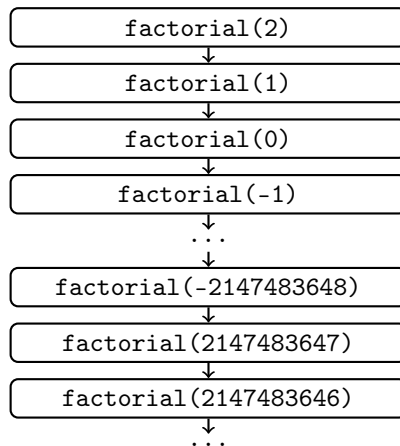
```

int factorial(int n)
{
    int answer = factorial(n-1) * n;
    return answer;
}

int main() {
    f(2);
    return 0;
}

```

Опишем порядок вызовов при вызове `factorial(2)`:



Получаем бесконечные рекурсивные вызовы. Запуск такой программы завершится ошибкой сегментации данных:

```

$ ./prog
Segmentation fault (core dumped)

```

Изначально может показаться странным, почему происходит ошибка доступа к памяти? Аргументы вызовов будут циклически пробегать все значения с 2147483647 до -2147483648 и так далее. При этом каждый вызов занимает 16 байт стека под адрес возврата (8 байт на 64-битной ОС), аргумент (4 байта) и локальные переменные (4 байта). Размер на стек можно проверить командой в терминале:

```
$ ulimit -s
```

Если, например, стек равен 8192 Кб, то максимальное количество рекурсивных вызовов может быть порядка 500 000. После чего произойдёт ошибка сегментации.

11.4 Хвостовая рекурсия

Пользоваться рекурсией можно только в том случае, когда системный стек гарантированно не переполняется. Часто рекурсивный алгоритм можно заменить на итеративный алгоритм (с циклами).

Например, дана рекурсивная функция, в которой для вычисления $f(n)$ необходимо вычислить только $f(n-1)$. Причём этот вызов единственный и находится в самом конце описания $f(n)$. Данный тип рекурсии называется хвостовой:

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    int answer = factorial(n-1) * n;
    return answer;
}
```

Такой рекурсивный код можно заменить на более оптимальный код с простым циклом:

```
int factorial(int n)
{
    int answer = 1, i;
    for (i = 2; i <= n; i++)
        answer = answer * i;
    return answer;
}
```

При этом для вычисления $f(n)$ в системном стеке потребуется память для адреса возврата и трех переменных (n , i , $answer$). Это значительно

меньше, чем расходы в рекурсивном решении, которые пропорциональны n .

11.5 Системный стек как хранилище данных

В системном стеке хранятся локальные переменные вызовов функций. Это можно использовать как самый настоящий стек при рекурсивных вызовах.

Пример 1. Дана последовательность букв, которая завершается переносом строки. Необходимо распечатать её в обратном порядке без использования массивов.

Ввод	star
Вывод	rats

Будем сохранять буквы в локальную переменную рекурсивной функции. Рекурсивная функция состоит из трёх операций:

1. ввод;
2. вывод;
3. рекурсивный вызов.

Соответствующий код

```
void print_array()
{
    char ch = getchar();
    if (ch == '\n')
        return;
    putchar(ch);
    print_array();
}
```

будет выводить буквы в том порядке, в котором они вводятся.

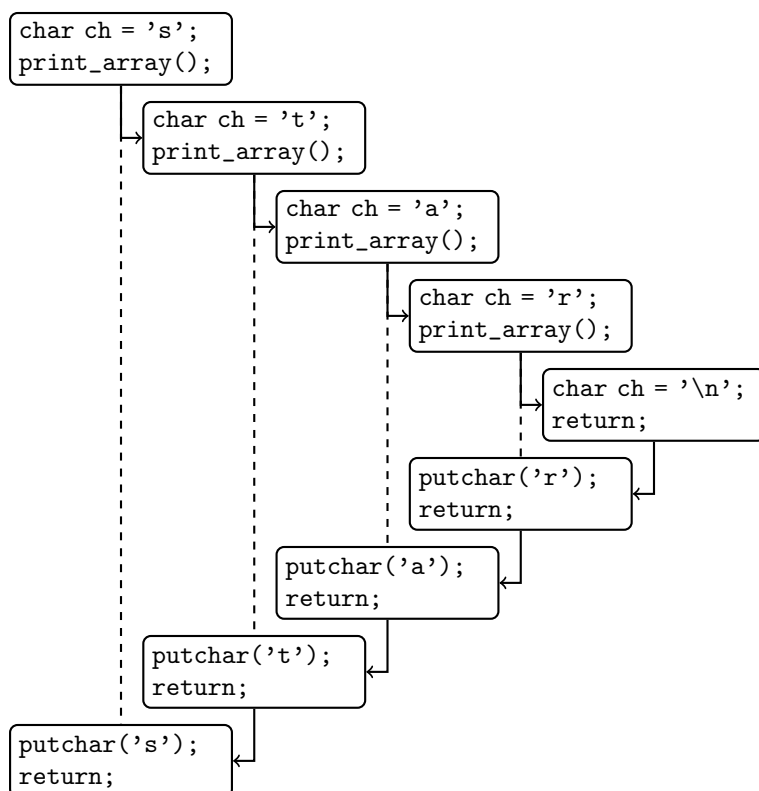
Что будет, если поменять местами вывод и рекурсивный вызов:

1. ввод;
2. рекурсивный вызов;
3. вывод?

Соответствующий код:

```
void print_array()
{
    char ch = getchar();
    if (ch == '\n')
        return;
    print_array();
    putchar(ch);
    return;
}
```

Выводит буквы в обратном порядке.



Давайте разберём пример со словом **star**. Переменная **ch** является локальной для каждого рекурсивного вызова. Значения этих переменных **ch** будут добавляться в системный стек в следующем порядке: **s**, **t**, **a**, **r**. После того как встретится перенос строки, начнётся свёртка системного

стека (прямо как в фильме «Начало», где герои пытались выйти из последнего уровня сна). По понятным причинам выводиться буквы будут в обратном порядке: **r**, **a**, **t**, **s**. И на выводе окажется слово **rats**.

11.6 Избыточные вычисления при рекурсии

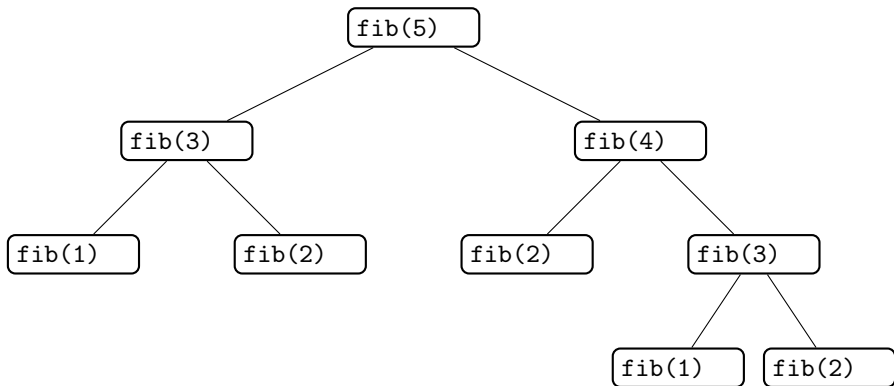
Использование рекурсии не всегда может быть эффективным. Рассмотрим пример с числами Фибоначчи. Дано целое число n от 1 до 90. Вывести F_n , где $F_n = F_{n-1} + F_{n-2}$, $F_1 = F_2 = 1$.

Приведём рекурсивное решение. Определим первые два значения явно, а остальные — рекуррентным соотношением с вызовом рекурсии.

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

Значение F_{90} не помещается в **int**, поэтому используем тип **long long**.

Построим дерево вызовов, которое инициирует вызов **fib(5)**:



Некоторые вызовы производятся многократно. Общее количество вызовов равно 9. Произведём оценку количества вызовов для произвольного n . Количество вызовов функции, порождённых **fib(n)**, равно сумме числа вызовов в **fib(n-1)** и **fib(n-2)**. Если число вызовов обозначить $T(n)$, то

$$\begin{cases} T(n) = T(n-1) + T(n-2) + 1, \\ T(1) = T(2) = 1. \end{cases}$$

Решением данной последовательности является $T(n) = 2F_n - 1$. А для вычисления F_{20} будет произведено более 10000 вызовов функций. Получаем, что рекурсивное решение работает значительно дольше итеративного.

Исключить повторяющиеся вызовы можно с помощью методики «мемоизации» — сохранение уже вычисленных значений в дополнительный массив. Но более правильным будет отказаться от рекурсивного решения в пользу итеративного. Хотя данный вид рекурсии и не является хвостовой, но решение тоже может быть легко записано без рекурсии и дополнительных массивов:

```
long long fib(int n)
{
    if (n <= 2)
        return 1;
    long long answer, f1 = 1, f2 = 1;
    for (i = 2; i <= n; i++)
    {
        answer = f1 + f2;
        f1 = f2;
        f2 = answer;
    }
    return answer;
}
```

Данным примером хочется проиллюстрировать, что рекурсией следует пользоваться аккуратно. С одной стороны, рекурсия часто имеет более простой код. Но с другой стороны, рекурсия может быть неэффективна и использовать дополнительные ресурсы в виде системного стека.

11.7 Примеры

Пример 2. Даны два целых числа a и b от 0 до 10^9 , причём хотя бы одно из них отлично от нуля. Вычислить наибольший общий делитель a и b . Написать рекурсивную функцию.

Ввод	120 0	800 1024	2017 1000
Вывод	120	32	1

Можно найти наибольший общий делитель перебором всех делителей от 1 до $\min(a, b)$. Но эффективнее использовать алгоритм Евклида (при $b \neq 0$):

$$(a; b) = (b; a \bmod b)$$

В частности:

$$\begin{aligned}
 (800; 1024) &= (1024; 800 \bmod 1024) = (1024; 800) = (800; 1024 \bmod 800) = \\
 &= (800; 224) = (224; 800 \bmod 224) = (224; 128) = (128; 224 \bmod 128) = \\
 &= (128; 96) = (96; 128 \bmod 96) = (96; 32) = (32; 96 \bmod 32) = (32; 0) = 32
 \end{aligned}$$

```

1  #include <stdio.h>
2
3  int gcd(int a, int b)
4  {
5      if (b == 0)
6          return a;
7      return gcd(b, a % b);
8  }
9
10 int main()
11 {
12     int a, b, d;
13     scanf("%d %d", &a, &b);
14     d = gcd(a, b);
15     printf("%d\n", d);
16     return 0;
17 }
```

Пример 3. Даны два целых числа: a от 0 до 10^9 и n от 0 до 10^4 , причём хотя бы одно из них отлично от нуля. Написать рекурсивную функцию, которая вычисляет последние 2 цифры числа a^n .

Ввод	2 5	3 11	2018 2019
Вывод	32	47	32

Максимальные входные параметры дают число 1000000000^{10000} , которое не помещается ни в один стандартный тип. Поэтому не получится вычислить a^n с использованием стандартных типов, чтобы в дальнейшем отделить последние 2 цифры.

Вместо вычисления a^n можно сразу вычислять остатки $a^k \bmod 100$ для каждой степени $k \leq n$. То есть:

$$a^k \bmod 100 = (a^{k-1} \bmod 100) * a \bmod 100.$$

Или при обозначении остатков $r_k = a^k \bmod 100$ получаем простое рекуррентное соотношение, которое легко реализуется в виде рекурсивной функции:

$$\begin{cases} r_k = r_{k-1} * a \bmod 100, \\ r_0 = 1. \end{cases}$$

Например, для вычисления последних двух цифр 3^5 достаточно две последних цифры $3^4 = 81$ умножить на 3, и у полученного результата взять две последних цифры 43.

a^k	3^0	3^1	3^2	3^3	3^4	3^5	3^6
$r_{k-1} * a$	-	3	9	27	81	243	129
r_k	1	3	9	27	81	43	29

Осталось заметить, что если основание a будет больше 99, то его можно заменить на остаток при делении на 100 по тем же правилам умножения в арифметике остатков. Например, две последние цифры числа 10203^{11} будут такими же, как и у числа 3^{100} . Таким образом выглядит простая рекурсивная реализация:

```

1  #include <stdio.h>
2
3  int lastDigits(int a, int k)
4  {
5      if (k == 0)
6          return 1;
7      return last_digits(a, k - 1) * a % 100;
8  }
9
10 int main()
11 {
12     int a, n, d;
13     scanf("%d %d", &a, &n);
14     d = lastDigits(a % 100, n);
15     printf("%d\n", d);
16     return 0;
17 }
```

Пример 4. Даны два целых числа a и n от 0 до 10^9 , причём хотя бы одно из них отлично от нуля. Написать рекурсивную функцию, которая вычисляет последние 2 цифры числа a^n . Для эффективного вычисления используйте бинарное возведение в степень.

Ввод	2 5	3 11	2019 123456789
Вывод	32	47	79

Почему бы не воспользоваться предыдущим решением? У него есть один существенный недостаток — последний пример $2019^{123456789}$ приводит к ошибке сегментации памяти. Происходит это из-за слишком глубокой рекурсии, так как каждый вызов функции сопряжён с расходами по памяти в системном стеке. В рамках нашего курса рекомендуется считать 100 000 вызовов критическим значением.

Рассмотрим пример вычисления 3^{11} . В решении из предыдущего примера понадобится 10 содержательных операций:

$$3^{11} = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$$

Можно ли как-то сократить количество вычислений? Да, если использовать бинарное возведение в степень:

$$3^{11} = 3^5 \cdot 3^5 \cdot 3$$

$$3^5 = 3^2 \cdot 3^2 \cdot 3$$

$$3^2 = 3 \cdot 3$$

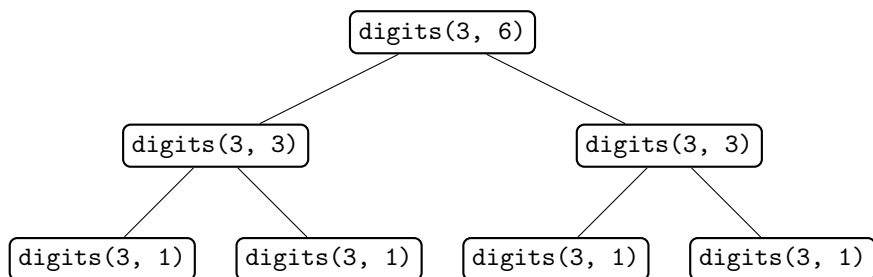
То есть, для вычисления потребовалось 5 операций умножения вместо 10. В общем виде получаем рекуррентное соотношение:

$$\begin{cases} r_k = r_{\lfloor k/2 \rfloor}^2 \cdot a \bmod 100, & \text{если } k \text{ — нечетное,} \\ r_k = r_{\lfloor k/2 \rfloor}^2 \bmod 100, & \text{если } k \text{ — четное,} \\ r_0 = 1. \end{cases}$$

Стоит предостеречь от логической ошибки, которая лишает весь трюк смысла. Если написать так:

```
int digits(int a, int k)
{
    if (k == 0)
        return 1;
    int answer = digits(a, k / 2) * digits(a, k / 2);
    answer = answer % 100;
    if (k % 2 == 1)
        answer = answer * a % 100;
    return answer;
}
```

То произойдет выполнение полного каскада рекурсивных вызовов.



При этом глубина рекурсии будет не такая большая как в предыдущем варианте (всего $m = \lfloor \log_2 n \rfloor$ против n). Теперь системный стек не будет переполняться. Но проблема будет не в памяти, а в количестве действий, которое не изменилось: $1 + 2 + 2^2 + \dots + 2^m = 2^{m+1} \approx 2n$.

Исправим данную оплошность: будем использовать уже вычисленное значение функции $\text{digits}(a, k / 2)$, а не вызывать её дважды.

```

int digits(int a, int k)
{
    if (k == 0)
        return 1;
    int answer = digits(a, k / 2);
    answer = answer * answer % 100;
    if (k % 2 == 1)
        answer = answer * a % 100;
    return answer;
}
  
```

Теперь количество действий будет порядка $\log_2 n$. По условию задачи известно, что $n \leq 10^9 < 2^{30}$. Значит, будет выполнено не более 60 операций умножения. А время работы программы сократится примерно в $n / \log_2 n$ раз.

Пример 5. Дана последовательность из цифр (0–9), круглых скобок и знаков арифметических действий (+ – *), которая задает арифметическую формулу вида:

```

<формула> := (<формула> <знак> <формула>) | <цифра>
<цифра>  := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<знак>   := + | - | *
  
```

Здесь запись $A := B$ означает, что вводится определение термина A , которое может принимать значение B . Вертикальная черта $|$ означает альтернативу. Например, «знак» — это плюс, минус или умножить.

Произвести вычисления формулы с помощью рекурсивной функции.

Ввод	$(1+(2*3))$	$((1+2)*(1-3))$	5
Вывод	6	-6	5

В данном примере постановка сама по себе рекурсивна (формула описывается через формулы), поэтому рекурсивное решение будет на порядок проще нерекурсивного. Данный метод носит название «рекурсивный спуск».

Заметим, что первый считанный символ формулы может быть либо цифрой, либо открывающейся скобкой. В первом случае достаточно вернуть цифру. Во втором случае считываем символы согласно описанию «(формула знак формула)»:

1. формулу слева от знака (вызов `left_value = calculation()`);
2. знак (вызов `ch = getchar()`);
3. формулу справа от знака (вызов `right_value = calculation()`);
4. закрывающуюся круглую скобку (вызов `getchar()`).

Далее достаточно вычислить итоговое значение в скобках, зная аргументы формулы (`left_value`, `right_value`) и знак (`sign`).

```

1  #include <stdio.h>
2
3  int calculation(void)
4  {
5      char ch = getchar();
6      if (ch >= '0' && ch <= '9')
7          return ch - '0';
8
9      int left_value, right_value, sign;
10     left_value = calculation();
11     sign = getchar();
12     right_value = calculation();
13     getchar();
14
15     if (sign == '+')
16         ans = left_value + right_value;
```

```
17     if (sign == '-')
18         ans = left_value - right_value;
19     if (sign == '*')
20         ans = left_value * right_value;
21     return ans;
22 }
23
24 int main(void)
25 {
26     printf("%d\n", calculation());
27     return 0;
28 }
```

11.8 Задания для самостоятельной работы

1. Что хранится в системном стеке во время рекурсивных вызовов?
2. Что будет распечатано на экране, если будет введено 4 числа: 2 0 1 9?

```
#include <stdio.h>

void f() {
    int a;
    scanf("%d", &a);
    if (a % 2 == 0)
        f();
    printf("%d ", a);
}

int main() {
    f();
    return 0;
}
```

3. Нарисуйте дерево вызовов (в узлах отметьте значение локальной переменной n в каждом вызове). Что будет распечатано на экране при вызове $f(6)$?

```
void f(int n) {
    if (n == 0)
        return;
    printf("<%d ", n);
    f(n/2);
    f(n/3);
    printf("%d> ", n);
}
```

4. Опишите данную функцию в виде рекурсивной функции без циклов.

```
int f(int n, int a[]) {
    int ans = 1;
    for ( ; n > 0; n--)
        ans *= a[n - 1];
    return ans;
}
```

5. Опишите рекурсивную функцию без параметров, которая вычисляет сумму всех введенных цифр в строке. Причём каждый вызов функции считает ровно один символ.

11.9 Практикум на ЭВМ

1. Дано целое число n от 1 до 1000. Вычислить приближенно определённый интеграл:

$$\int_0^1 f(x) dx$$

методом левых прямоугольников:

$$h \sum_{i=0}^{n-1} f(i \cdot h),$$

где $h = \frac{1}{n}$. Вывести три ответа (с 4 знаками после запятой): для функции $f(x) = x$, $f(x) = x^2$, $f(x) = \sin(x)$. Описать функцию:

```
double rectangle(int n, double f(double));
```

Ввод	2	100
Вывод	0.2500 0.1250 0.2397	0.4950 0.3284 0.4554

2. Дано целое число n от 1 до 10^9 . Найти произведение ненулевых цифр. Описать рекурсивную функцию, которая вычисляет ответ следующим образом: $\text{digits}(12045) = 5 * \text{digits}(1204)$.

```
int digits(int n);
```

Ввод	12045	999	100200300
Вывод	40	729	6

3. Дана последовательность целых чисел. Каждое число принимает значение в диапазоне от -100 до 100 и отлично от нуля. После последовательности вводится ноль. Найти сумму чисел последовательно. Описать рекурсивную функцию, которая считывает одно число, вызывает рекурсию и возвращает сумму данного числа с результатом рекурсивного вызова:

```
int tail();
```

Ввод	1 2 3 4 -1 -2 0
Вывод	7

4. Дана последовательность целых чисел, каждое из которых от -100 до 100 и не равно нулю. После последовательности вводится нуль. Вывести числа последовательности в обратном порядке. Описать рекурсивную функцию, которая считывает одно число, делает рекурсивный вызов и выводит одно число:

```
void reverse(void);
```

Ввод	1 2 3 4 -1 -2 0
Вывод	-2 -1 4 3 2 1

5. Даны целые числа: n от 1 до 100 и k от 2 до 10. Вывести n в k -ичной системе счисления. Описать рекурсивную функцию (вычисляет последнюю цифру числа и делает рекурсивный вызов):

```
void printDigit(int n, int k);
```

Ввод	11 2	32 9	2016 10
Вывод	1011	35	2016

6. Дана последовательность целых чисел, каждое из которых от 1 до 10^9 . После последовательности вводится нуль. Найти наибольший общий делитель всех чисел последовательности. Описать две рекурсивные функции. Первая вычисляет НОД двух чисел по алгоритму Евклида. Вторая считывает одно число и возвращает НОД этого числа и результата рекурсивного вызова.

```
int gcd(int a, int b);
int multigcd();
```

Ввод	4 8 6 0	21 9 12 0	5 0
Вывод	2	3	5

7. Дано слово (длиной не более 100), состоящее только из строчных латинских букв. Проверить, является ли это слово палиндромом. Выведите YES или NO соответственно. Описать рекурсивную функцию, которая проверяет является ли палиндромом строка $a[i \dots n - i - 1]$ и возвращает 1, если это палиндром, и 0 — в противном случае.


```
int check(int n, char a[], int i);
```

Ввод	label	abba	gag
Вывод	NO	YES	YES

8. Даны целые числа: a от 1 до 10^9 , n от 1 до 10^{18} и p от 2 до 10^9 . Вывести $a^n \bmod p$ с помощью бинарного возведения в степень. Описать рекурсивную функцию с глубиной не более $\log_2 n$ вызовов.

```
long long powerMod(long long a,
                   long long n,
                   long long mod);
```

Ввод	11 2 100	5 4 9	2 64 10
Вывод	21	4	6

9. Даны целые числа s и n от 1 до 100. Распечатать всевозможные представления s в виде суммы n натуральных слагаемых. Описать рекурсивную функцию с аргументами:

- s — сумма, которую нужно разложить на слагаемые;
- n — изначальное количество слагаемых;
- k — количество слагаемых, которое уже использовано;
- $\text{answer}[]$ — слагаемые, которые уже использованы;

Ответ `answer` печатаем, если $n = k$.

```
void addends(int s, int n, int k, int answer[]);
```

Ввод	5 3
Вывод	1 1 3 1 2 2 1 3 1 2 1 2 2 2 1 3 1 1

10. На первой строке дано целое число n от 1 до 100. На второй строке дан массив из n различных целых чисел. Отсортировать массив сортировкой «слиянием». Описать рекурсивную функцию `mergeSort`, в которой выполняются следующие действия:

- (a) сортируется левая и правая половина массивы рекурсивными вызовами (`a[left..middle]` и `a[middle+1..right]`);
- (b) подмассивы `a[left..middle]` и `a[middle+1..right]` объединяется в один отсортированный буфер:
 - i. инициализируются два индекса `l = left`; `r = middle`;
 - ii. если `a[l] < a[r]`, то `a[l]` копируется в конец буфера и `l` сдвигается вправо на один элемент, в противном случае `a[r]` копируется в конец буфера и `r` сдвигается вправо на один элемент;
 - iii. если один из подмассивов закончился (`l > middle` или `r > right`), то остаток другого подмассива копируется в конец буфера;
- (c) буфер копируется в исходный массив.

```
void sort(int n, int a[], int left, int right);
```

Ввод	5 2 1 5 3 4	5 5 4 3 2 1	5 1 2 3 4 5
Вывод	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5

12 Структуры

В теме разобраны примеры использования составных типов данных, которые называются структурами.

12.1 Описание структуры

Рассмотрим задачу, в которой нужно обрабатывать отсчеты времени. Описание момента времени 12:10:05 в программе может выглядеть так:

```
int minute = 12;
int second = 10;
int hour = 5;
```

Для передачи в качестве аргумента функции данного момента времени необходимо использовать три отдельных аргумента. Все операции присваивания и копирования придётся проводить отдельно для секунд, минут и часов.

Для удобства можно скомбинировать все три параметра в одну переменную. Для этого опишем структуру типа `Time`:

```
struct Time {
    int hour;
    int minute;
    int second;
};
```

Описание составного типа начинается со служебного слова `struct`, после чего идёт любое допустимое имя будущей структуры. Далее в фигурные скобки заключается перечисление переменных — поля структуры. В конце описания структуры ставится точка с запятой.

После объявления типа `struct Time` можно создать переменную данного типа следующим образом:

```
struct Time time = {12, 10, 5}
```

Далее мы сможем работать с моментом времени 12:10:05, используя всего одну переменную `time`.

Однотипные поля можно перечислять через запятую. Например, структура для трёхмерных векторов:

```
struct Vector3D {
    double x, y, z;
};
```

Названия для полей следует подбирать таким образом, чтобы они соответствовали описываемому объекту. Например, для комплексных чисел поля лучше назвать `re` (real, вещественная часть) и `im` (image, мнимая часть):

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

Если структура используется в нескольких функциях, то описание следует сделать глобальным для всех функций. Например, сразу после завершения подключения заголовочных файлов с помощью `#include`.

Если структура используется только в рамках одной функции, то описание можно выполнить в теле этой функции.

12.2 Объявление переменных

Объявлять переменные типа структуры можно так же, как и переменные встроенных типов:

```
struct Time {
    int hour;
    int minute;
    int second;
};

int main() {
    struct Time start;
    ...
}
```

Можно описать сразу и структуру, и соответствующие переменные. Например, таким образом задаются три клетки шахматной доски:

```
int main() {
    struct ChessField {
        char column;
        int row;
    } field1, field2, field3;
    ...
}
```

В качестве полей структуры можно использовать массивы. Например, в структуре для описания длинных чисел будет два поля: массив цифр числа и их количество.

```
struct BigInteger {  
    int digit[100], size;  
};
```

Рассмотрим структуру для описания «студента» с заданной фамилией, именем и возрастом:

```
struct Student {  
    char name[10];  
    char surname[10];  
    int age;  
};
```

В качестве поля могут быть и другие структуры. Например, группа студентов (название группы, курс и староста):

```
struct Group {  
    char name[4];  
    int course;  
    struct Student captain;  
};
```

12.3 Инициализация структур

Обращение к полям структуры производится через оператор точка:

```
struct Time {  
    int hour;  
    int minute;  
    int second;  
};  
...  
struct Time start_time;  
start_time.hour = 12;  
start_time.minute = 10;  
start_time.second = 5;
```

Аналогичным образом можно получать значения:

```
if (start_time.minute >= 10)
    start_time.minute -= 10;
```

Инициализировать переменную можно при её объявлении. Например, опишем структуру «вектор» и объявим переменную **e** для единичного вектора, направленного вдоль оси ординат:

```
struct Vector3D {
    double x, y, z;
};
...
struct Vector3D e = {0.0, 1.0, 0.0};
```

Таким образом можно инициализировать клетку шахматного поля:

```
struct ChessField {
    char column;
    int row;
}
...
struct ChessField field = {'E', 2};
```

Для полей массивов верно всё, что верно и для обычных массивов. Поля следует перечислять в порядке объявления:

```
struct BigInteger {
    int size;
    int digit[100];
};
...
struct BigInteger a = {2, {2, 1}};
struct BigInteger b = {3, {9, 8, 7}};
```

В случае с полями-строками всё аналогично:

```
struct Student {
    char name[10];
    char surname[10];
    int age;
};
...
struct Student student = {"Linus", "Torvalds", 21};
```

Стоит отметить, что присваивания строк недопустимы:

```
student.name = "Linus";
```

Код выше вызовет ошибку:

```
error: assignment to expression with array type
```

Если в поле типа «строка» необходимо записать текст, то следует использовать функцию `char *strcpy(char *dest, const char *src)`; из заголовочного файла `string.h`:

```
strcpy(student.name, "Linus");
```

Работать с отдельными элементами такого поля следует обычным образом. Данный код изменит пятую букву имени:

```
student.name[4] = 'x';
puts(student.name);
puts(student.surname);
printf("%d\n", student.age);
```

На выводе получим:

```
Linux
Torvalds
21
```

В качестве поля могут быть использованы и другие структуры:

```
struct Group mm11 =
{
    "mm",
    1,
    {
        "Linus", "Torvalds", 21
    }
};
```

Для работы с такими вложенными полями необходимо использовать тот же оператор точка:

```
if (mm11.captain.age != 21)
    mm11.captain.age = 21;
```

12.4 Расположение в памяти

Размер одного экземпляра структуры равен сумме размеров его полей. Узнать размер структуры можно с помощью функции-оператора `sizeof()`, который можно применить к имени структуры, либо к отдельному экземпляру структуры:

```
struct Student {
    char name[10];
    char surname[10];
    int age;
};
...
struct Student student = {"Linus", "Torvalds", 21};
int size1 = sizeof(struct Student);
int size2 = sizeof(student);
```

Поля в памяти расположены друг за другом (вопросительным знаком обозначены неинициализированные ячейки памяти):

student.name										student.surname										student.age			
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	i	n	u	s	∅	?	?	?	?	T	o	r	v	a	l	d	s	∅	?	21			

Выход за границы массива, который является полем, может испортить другие поля этой же структуры. Например, через поле `name` меняется поле `surname` и наоборот:

```
student.name[11] = 'u';
student.name[12] = 'x';
student.name[13] = 0;
student.surname[-6] = 'x';
puts(student.name);
puts(student.surname);
printf("%d\n", student.age);
```

student.name										student.surname										student.age			
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	i	n	u	x	Ø	?	?	?	?	T	u	x	Ø	a	l	d	s	Ø	?	21			

На выводе получим:

```
Linux
Tux
21
```

12.5 Оператор копирования

И массив, и структура занимают непрерывный блок памяти. Но в отличие от обычных массивов структуры можно копировать оператором присваивания. Например:

```
struct Vector3D {
    double x, y, z;
};
...
struct Vector3D v1 = {2.0, 3.0, 4.0}, v2;
v2 = v1;
```

В процессе присваивания все 24 байта (3 поля типа `double`) копируются из переменной `v1` в переменную `v2`.

Таким же образом можно копировать целые массивы обычным оператором присваивания. Например, структура для длинных чисел:

```
struct BigInteger {
    int size;
    int digit[100];
};
...
struct BigInteger a = {4, {9, 1, 0, 2}}, b;
b = a;
```

Копируются все 404 байта структуры.

12.6 Передача по значению

В отличие от массивов передача аргумента в функцию происходит по значению, как и для простых типов. То есть создается локальная переменная типа структуры, в которую копируется значение структуры-аргумента. Например, рассмотрим функцию, которая вычисляет скалярное произведение трёхмерных векторов, и её вызов:

```
double dot(struct Vector3D v1, struct Vector3D v2)
{
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
...
struct Vector3D a = {1.0, 2.0, 3.0};
struct Vector3D b = {3.0, 1.0, 0.0};
double dot_product = dot(a, b);
...
```

В ходе выполнения данной программы создается целых 4 переменных типа `struct Vector3D`. До вызова функции создаются 2 переменные `a` и `b`. А во время вызова функции `dot` создаются локальные для этой функции переменные `v1` и `v2`, в которые копируются, соответственно, все поля из переменных `a` и `b`. Далее вычисляется скалярное произведение. Результат возвращается, а локальные переменные `v1` и `v2` уничтожаются.

Рассмотрим ещё один пример. Опишем функцию, которая вычисляет сумму двух трёхмерных векторов, и её вызов:

```
struct Vector3D sum(struct Vector3D v1,
                   struct Vector3D v2)
{
    struct Vector3D v3;
    v3.x = v1.x + v2.x;
    v3.y = v1.y + v2.y;
    v3.z = v1.z + v2.z;
    return v3;
}
...
struct Vector3D a = {1.0, 2.0, 3.0};
struct Vector3D b = {3.0, 1.0, 0.0};
struct Vector3D c;
c = sum(a, b);
...
```

Во время выполнения данной программы создаются 6 переменных типа `struct Vector3D`: `a`, `b`, `c` до вызова функции и `v1`, `v2`, `v3` — локальные переменные функции. Копирование производится трижды: из `a` в `v1`, из `b` в `v2` и из `v3` в `c`.

12.7 Передача по указателю

Структуры могут быть достаточно объёмными. Когда аргументы функции нужны лишь для получения значений, от копирования можно избавиться, используя указатели. Напомним, что имеется две основные операции:

1. `&x` — взятие адреса у переменной `x` (узнаем номер первой ячейки памяти, в которой лежит переменная `x`);
2. `*p` — разыменование адреса `p` (получаем доступ к ячейкам памяти, начиная с адреса `p`).

Например:

```
struct Vector3D a = {1.0, 2.0, 3.0};
struct Vector3D *ptr = &a;
```

Теперь в переменной `ptr` будет храниться адрес переменной `a`. Это меньше, чем размер всей структуры: 8 байт против 24 байт. Обращаться к самой структуре можно через операцию разыменования `*`. Например, скопировать значение поля `x`:

```
double value = (*ptr).x;
```

Важно отметить, что оператор точка `.` имеет приоритет выше, чем оператор `*`. Поэтому для корректного доступа необходимы скобки. В противном случае:

```
double value = *ptr.x;
```

компилятор будет пытаться получить доступ к полю `x` у переменной `ptr`:

```
error: request for member 'x' in something
not a structure
```

12.8 Оператор стрелка

Удобным вариантом сокращения пары операторов: разыменования `(*)` и доступа к полю `(.)` — является оператор «стрелка» `->` (символ «минус» и символ «больше»). Данное действие:

```
double value = (*ptr).x;
```

можно заменить на более компактную запись:

```
double value = ptr->x;
```

Снова вернёмся к примеру функции со скалярным произведением.

```
double dot(struct Vector3D *a, struct Vector3D *b)
{
    return a->x * b->x + a->y * b->y + a->z * b->z;
}
...
struct Vector3D a = {1.0, 2.0, 3.0};
struct Vector3D b = {3.0, 1.0, 0.0};
double dot_product = dot(&a, &b);
...
```

Теперь переменных типа `Struct Vector3D` всего лишь две. Посчитаем выигрыш с точки зрения памяти для локальных переменных: было две переменные типа структура ($2 \cdot 24 = 48$ байт), стало две переменные типа указатель ($2 \cdot 8 = 16$ байт).

Указатели также применяются для того, чтобы изменять значение переменных. Например, рассмотрим задачу с вводом двух трёхмерных векторов:

```
struct Vector3D a, b;
scanf("%lf %lf %lf", &a.x, &a.y, &a.z);
scanf("%lf %lf %lf", &b.x, &b.y, &b.z);
```

Явно наблюдается повтор одного и того же кода. Опишем функцию для ввода:

```
struct Vector3D a, b;
scan_vec(&a);
scan_vec(&b);
```

Прототип функции должен быть таким:

```
void scan_vec(struct Vector3D *ptr);
```

Получить доступ к полям этой структуры можно через оператор «стрелку»: `ptr->x`, `ptr->y`, `ptr->z`:

```
void scan_vec(struct Vector3D *ptr) {
    scanf("%lf %lf %lf",
        &(ptr->x),
        &(ptr->y),
```

```

        &(ptr->z));
    }

```

Если вы помните правильный порядок приоритетов операторов, то скобки можно опустить: `&ptr->x`.

Структуры в модифицированном виде еще встретятся в курсе по C++ в виде классов, где передача по указателю — стандартная практика.

12.9 Макроопределение типа

При использовании структур часто возникают очень длинные строки кода из-за записи `struct` Данную запись можно сократить, если воспользоваться знакомым макросом `typedef`.

Например, опишем функцию для вычисления произведения комплексных чисел с коротким обозначением типа `Comp`:

```

struct Complex
{
    double re, im;
};

typedef struct Complex Comp;

Comp multiply(Comp *z1, Comp *z2)
{
    Comp result;
    result.re = z1->re * z2->re - z1->im * z2->im;
    result.im = z1->re * z2->im + z2->re * z1->im;
    return result;
}

...
Comp z = {1.0, 2.0}, u = {3.0, 4.0}, v;
v = multiply(&z, &u);

```

12.10 Примеры

Пример 1. На первой строке дано время в формате `hh:mm:ss`. На следующей строке дано целое число d от 0 до $24 \cdot 60 \cdot 60$. Вывести отсечку времени спустя d секунд (возможен переход на новые сутки).

Ввод	12:10:00 80	23:59:58 3
Вывод	13:30:00	00:00:01

Идея решения заключается в том, чтобы перевести текущее время в абсолютное время в секундах от начала суток. Далее прибавить d секунд. И отсчитать количество часов, минут и секунд снова от начала суток.

Что касается технического оформления, то здесь разумно не заводить новую переменную типа «время» внутри функции, а записать ответ в ту же локальную переменную.

```
1  #include <stdio.h>
2
3  struct Time {
4      int hour, minute, second;
5  };
6
7  struct Time addTime(struct Time time, int delta) {
8      int absolute = time.second +
9                      time.minute * 60 +
10                     time.hour * 60 * 60;
11      absolute += delta;
12
13      time.second = absolute % 60;
14      absolute /= 60;
15      time.minute = absolute % 60;
16      absolute /= 60;
17      time.hour = absolute % 24;
18      return time;
19  }
20
21  int main() {
22      struct Time time, new_time;
23      int delta;
24      scanf(
25          "%d:%d:%d",
```

```

26         &time.hour,
27         &time.minute,
28         &time.second
29     );
30     scanf("%d", &delta);
31     new_time = addTime(time, delta);
32     printf(
33         "%02d:%02d:%02d\n",
34         new_time.hour,
35         new_time.minute,
36         new_time.second
37     );
38     return 0;
39 }

```

Пример 2. На первой строке даны два вещественных числа a и b , которые определяют комплексное число $z = a + bi$. На второй строке дано целое число n от 0 до 100. Вычислить z^n и вывести вещественную и мнимую часть ответа через пробел.

Ввод	2.00 1.00 0	2.00 1.00 4	-0.60 -0.80 12
Вывод	1.00 0.00	-7.00 24.00	0.13 0.99

Разумно описать две функции: для умножения двух комплексных чисел и непосредственно для вычисления степени комплексного числа. Возводить в степень будем самым простым образом, но важно, чтобы возведение в 0 степень работало корректно. Поэтому для вычисления z^n начнём с нейтрального элемента $(1, 0)$ и n раз умножим его на z :

```

1  #include <stdio.h>
2
3  struct Complex {
4      double re, im;
5  };
6
7  typedef struct Complex Comp;
8
9  Comp multiply(Comp a, Comp b) {
10     Comp c;
11     c.re = a.re * b.re - a.im * b.im;
12     c.im = a.re * b.im + a.im * b.re;

```

```

13     return c;
14 }
15
16 Comp power(Comp z, int n) {
17     int i;
18     Comp result = {1.0, 0.0};
19     for (i = 0; i < n; i++)
20         result = multiply(result, z);
21     return result;
22 }
23
24 int main() {
25     Comp z, answer;
26     int n;
27     scanf("%lf %lf\n%d", &z.re, &z.im, &n);
28     answer = power(z, n);
29     printf("%.2lf %.2lf\n", answer.re, answer.im);
30     return 0;
31 }

```

Пример 3. Даны три точки в трёхмерном пространстве (координаты каждой точки на отдельной строчке). Все координаты представляют собой целые числа от -1000 до 1000 . Необходимо вычислить объем тетраэдра, который образован этими тремя точками и началом координат с точностью 2 знака после запятой.

Ввод	1 0 0 0 2 0 0 0 3	0 -1 0 1 0 0 0 0 1	1 2 3 -4 -5 -6 7 -8 9
Вывод	1.00	0.17	16.00

Обозначим соответствующие радиус-вектора v_1 , v_2 и v_3 . Из курса линейной алгебры известно, что объём тетраэдра, натянутого на тройку векторов, в 6 раз меньше объёма соответствующего параллелепипеда. Объём параллелепипеда равен модулю смешанного произведения векторов v_1 , v_2 , v_3 :

$$(v_1, v_2, v_3) = ([v_1, v_2], v_3)$$

Опишем три функции: скалярное произведение (a, b) , векторное произведение $[a, b]$ и ввод вектора. Причём первые две функции можно реализовать передачей по значению или по указателю. Рассмотрим оба варианта.

Решение с передачей аргументов функции по значению:


```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Vector3D {
5     int x, y, z;
6 };
7
8 typedef struct Vector3D Vec3;
9
10 int dot(Vec3 a, Vec3 b) {
11     return a.x * b.x + a.y * b.y + a.z * b.z;
12 }
13
14 Vec3 cross(Vec3 a, Vec3 b) {
15     Vec3 result;
16     result.x = a.y * b.z - a.z * b.y;
17     result.y = a.z * b.x - a.x * b.z;
18     result.z = a.x * b.y - a.y * b.x;
19     return result;
20 }
21
22 void scan_vec(Vec3 *a) {
23     scanf("%d %d %d", &(a->x), &(a->y), &(a->z));
24 }
25
26 int main() {
27     Vec3 a, b, c;
28     int product;
29     scan_vec(&a);
30     scan_vec(&b);
31     scan_vec(&c);
32     product = dot(cross(a, b), c);
33     printf("%.2lf\n", abs(product) / 6.0);
34     return 0;
35 }
```

Решение с передачей аргументов функции через указатели:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```

```
4  struct Vector3D {
5      int x, y, z;
6  };
7
8  typedef struct Vector3D Vec3;
9
10 int dot(Vec3 *a, Vec3 *b) {
11     return a->x * b->x +
12           a->y * b->y +
13           a->z * b->z;
14 }
15
16 Vec3 cross(Vec3 *a, Vec3 *b) {
17     Vec3 result;
18     result.x = a->y * b->z - a->z * b->y;
19     result.y = a->z * b->x - a->x * b->z;
20     result.z = a->x * b->y - a->y * b->x;
21     return result;
22 }
23
24 void scan_vec(Vec3 *a) {
25     scanf("%d %d %d", &(a->x), &(a->y), &(a->z));
26 }
27
28 int main() {
29     Vec3 a, b, c, cross_ab;
30     int product;
31     scan_vec(&a);
32     scan_vec(&b);
33     scan_vec(&c);
34     cross_ab = cross(&a, &b);
35     product = dot(&cross_ab, &c);
36     printf("%.2lf\n", abs(product) / 6.0);
37     return 0;
38 }
```

В данном решении особенно стоит обратить внимание на дополнительную переменную `cross_ab`. Здесь следует предостеречь от очень плохого варианта кода:

```
35     product = dot(&cross(&a, &b), &c);
```

Данное решение может привести к неопределённому поведению программы, включая ошибку сегментации данных. Проблема заключается в том, что функция `cross()` возвращает локальную переменную `result`. Так как при завершении функции уничтожаются все её локальные переменные, единственная возможность в дальнейшем использовать значения из переменной `result` — скопировать эти значения в другую переменную.

Код выше делает следующее:

1. на 33 строке получаем адрес переменной `result` из 20 строки функции `cross()`;
2. при завершении функции `cross()` уничтожаем её локальную переменную `result`;
3. передаем уже неактуальный адрес в функцию `dot`;
4. на 11 строке разыменовываем некорректный адрес `a->x` в функции `dot`.

Последнее действие приводит к ошибке сегментации на строке 11.

Пример 4. На первой строке дано целое число n от 1 до 10. На второй строке дана последовательность из n координат шахматных клеток. Вывести все различные пары полей, для которых верно, что с первого поля из этой пары можно попасть на второе поле из этой пары ходом слона. Слон ходит по диагонали на любое количество клеток.

Ввод	4 A1 D2 B2 C1
Вывод	A1 B2 D2 C1 B2 C1

Сохраним все шахматные поля в массиве структур. Переберём все пары полей двойным циклом: фиксируем i -й элемент и для каждого i перебираем в качестве пары j такие, что $j > i$:

```
for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
        a[i] compare with a[j]
```

При использовании массива структур обращение к полям i -ой структуры из массива будет производиться следующим образом: `a[i].row` и `a[i].col`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct ChessField {
5      char col;
6      int row;
7  };
8
9  typedef struct ChessField Chess;
10
11 int hit(Chess *field1, Chess *field2) {
12     return (abs(field1->col - field2->col) ==
13             abs(field1->row - field2->row));
14 }
15
16 void print(Chess *field) {
17     printf("%c%d ", field->col, field->row);
18 }
19
20 int main() {
21     Chess fields[10];
22     int n, i, j;
23     scanf("%d\n", &n);
24     for (i = 0; i < n; i++)
25         scanf(
26             "%c%d ",
27             &fields[i].col,
28             &fields[i].row
29         );
30     for (i = 0; i < n; i++)
31         for (j = i + 1; j < n; j++)
32             if (hit(&fields[i], &fields[j])) {
33                 print(&fields[i]);
34                 print(&fields[j]);
35                 putchar('\n');
36             }
37     return 0;
38 }
```

Пример 5. Даны два целых числа от 0 до 10^{99} (каждое на отдельной


```
#include <stdio.h>

struct BigInteger{
    int size;
    int digit[100];
};

void scan_long_int(struct BigInteger *number) {
    int i, j, d1, d2;
    char c;
    number->size=0;

    for (i = 0; i < 100; i++)
        number->digit[i] = 0;

    c = getchar();
    while (c >= '0' && c <= '9') {
        number->digit[number->size] = c - '0';
        number->size++;
        c = getchar();
    }
    int m = number->size - 1;
    for (i = 0, j = m; i < j; i++, j--) {
        d1 = number->digit[i];
        d2 = number->digit[j];
        number->digit[i] = d2;
        number->digit[j] = d1;
    }
}

void print_long_int(struct BigInteger *number)
{
    int i;
    for (i = number->size - 1; i >= 0; i--)
        printf("%d", number->digit[i]);
    putchar('\n');
}

struct BigInteger sum(struct BigInteger *num1,
                     struct BigInteger *num2)
```

```
{
    int i, carry = 0, digit;
    int n1 = num1->size;
    int n2 = num2->size;
    struct BigInteger result = {0};
    for (i = 0; i < n1 || i < n2; i++) {
        digit = num1->digit[i] + num2->digit[i];
        digit += carry;
        result.digit[i] = digit % 10;
        carry = digit / 10;
        result.size++;
    }
    if (carry > 0) {
        result.digit[result.size] = carry;
        result.size++;
    }
    return result;
}

int main() {
    struct BigInteger a, b, c;
    scan_long_int(&a);
    scan_long_int(&b);
    c = sum(&a, &b);
    print_long_int(&c);
    return 0;
}
```

12.11 Задания для самостоятельной работы

1. Что будет напечатано?

```
struct Student {
    char name[10];
    char surname[15];
    int mark[3];
};
...
struct Student stud;
printf("%lu", sizeof(stud));
printf(" %lu", sizeof(stud.mark));
```

2. Дана структура:

```
struct MyStructure {char x[2]; char y;} example;
```

Описать 3 различных способа заполнения всех полей переменной `example` структуры `MyStructure` буквами 'z'.

3. Описать три переменные `a`, `b` и `c`, для которых корректны обращения `a[5].x`, `b.x[5]` и `c[5].x[3]` соответственно.
4. Описать структуру «треугольник» с полем в виде массива из 3 точек. Для описания точки использовать структуру «точка» из 2 полей `x` и `y` типа `int`. Создать переменную с именем `egypt`, которая описывает треугольник с вершинами в точках (0; 0), (3; 4) и (0; 4).
5. Дана структура, описывающая героя с запасом маны:

```
struct Hero {
    char name[10];
    int mana;
};
...
struct Hero heroes[100];
...
```

Массив `heroes` уже заполнен. Написать цикл, который выводит имена тех героев, чье имя начинается с буквы 'А' и запас маны составляет не менее 100 единиц.

12.12 Практикум на ЭВМ

1. Даны 2 трёхмерных целочисленных вектора. Найти их скалярное и векторное произведение. Описать соответствующие функции:

```
int dot(struct Vector a, struct Vector b);
struct Vector cross(struct Vector a,
                    struct Vector b);
```

Ввод	1 2 3 2 3 4
Вывод	20 -1 2 -1

2. Даны 2 точки на плоскости. Найти площадь треугольника, образованного этими точками и началом координат. Вывести ответ с одним знаком после запятой. Описать соответствующую функцию:

```
double cross(struct Point p1, struct Point p2);
```

Прочитать про геометрию можно здесь:

<https://habrahabr.ru/post/147691>

<https://habrahabr.ru/post/148325>

Ввод	1 2 3 4	-1 0 0 1
Вывод	1.0	0.5

3. Даны 4 точки на плоскости A, B, C, D . Проверить, лежит ли точка D внутри треугольника ABC . Описать соответствующую функцию, которая возвращает 1, если точка лежит внутри, и 0 — в противном случае:

```
int inside(struct Point A,
           struct Point B,
           struct Point C,
           struct Point D)
```

Ввод	1 2 0 0 2 1 1 1	1 2 0 0 2 1 0 2
Вывод	YES	NO

4. Даны 2 комплексных числа z_1 и z_2 . Каждое комплексное число задано вещественной и мнимой частью. Вычислить $z_1 * z_2 + z_1 + z_2$. Описать соответствующие функции:

```
struct Complex sum(struct Complex z1,
                  struct Complex z2);
struct Complex mult(struct Complex z1,
                   struct Complex z2);
```

Ввод	1.0 2.0 2.0 3.0
Вывод	-1.0 12.0

5. Дано целое положительное число n . Распечатать вершины правильного n -угольника с центром в точке $(0; 0)$, где одна из вершин имеет координату $(1; 0)$. Указание: найдите z^k , где $z = (\cos(\frac{2\pi}{n}); \sin(\frac{2\pi}{n}))$, k принимает все значения от 0 до $n - 1$.

```
struct Complex mult(struct Complex z1,
                  struct Complex z2);
```

Ввод	8
Вывод	1.00 0.00 0.71 0.71 0.00 1.00 -0.71 0.71 -1.00 0.00 -0.71 -0.71 0.00 -1.00 0.71 -0.71

6. Дано время в формате чч:мм:сс и целое положительное число x от 1 до $24 \cdot 60 \cdot 60$. Вывести время через x секунд спустя и x секунд до данного времени. Описать соответствующие функции:

```
struct Time addTime(struct Time t, int s)
struct Time subTime(struct Time t, int s)
```

Ввод	09:10:11 123	00:01:02 3663
Вывод	09:12:14 09:08:08	01:02:05 22:59:59

7. Дано целое положительное число n от 1 до 10 – количество выбранных полей на шахматной доске. Далее n шахматных полей. Напечатать все пары полей таких, что с одного из них можно попасть на другое ходом коня. Пары вывести в лексикографическом порядке (раньше идут те поля, которые раньше по алфавиту). Описать соответствующую функцию, которая возвращает 1, если с одного поля можно попасть ходом коня на другое, и 0 – в противном случае.

```
int knight(struct Chess field1,
           struct Chess field2);
```

Ввод	4 A1 C3 D1 B2
Вывод	B2 D1 C3 D1

8. Дано целое положительное число n от 1 до 10 – количество студентов. Далее n записей, которые состоят из имени студента и возраста. Найти самого старшего и самого младшего. Если таких несколько, то вывести первого. Описать соответствующие функции, которые возвращают порядковый номер самого младшего и порядковый номер самого старшего студента:

```
int argMinAge(int n, struct Student students[]);
int argMaxAge(int n, struct Student students[]);
```

Ввод	3 Dracula 100 Vlad 300 Mavis 21	2 Mavis 21 Andy 21
Вывод	Vlad Mavis	Mavis Mavis

9. Даны две положительные дроби. Найти их сумму и произведение. Результат сократить на наибольший общий делитель. Описать соответствующие функции:

```
int gcd(int a, int b);
struct Rational sum(struct Rational a,
                   struct Rational b);
struct Rational mult(struct Rational a,
                   struct Rational b);
```

Ввод	2/15 1/6
Вывод	3/10 1/45

10. В тексте содержится информация о студентах: количество студентов и информация по ним (фамилия, имя, 3 оценки от 0 до 10). Вывести фамилию студента с минимальным и максимальным средним баллом:

$$\bar{r} = \frac{r_1 + r_2 + r_3}{3}.$$

Вывести фамилии студентов с минимальным и максимальным отклонением от своего среднего балла:

$$d = \sum_{i=1}^3 |r_i - \bar{r}|.$$

Вещественные числа выводить с точностью в два знака после запятой. Для каждого студента использовать структуру вида:

```
struct Student {
    char surname[10];
    char name[10];
    int rating[3];
};
```

Ввод	3 Mikhno Xenia 7 8 7 Korobov Pavel 6 9 6 Pikulina Alice 10 9 5
Вывод	Korobov 7.00 Pikulina 8.00 Xenia 1.33 Alice 6.00

13 Указатели.

Динамические массивы

Динамическое выделение памяти — это возможность управлять памятью вручную. В нашем курсе, это будет показано на примере с массивами, матрицами и структурами.

13.1 Одна переменная

В ранее рассмотренных примерах память выделялась статическим образом. Выделение памяти под переменные и очищение данной памяти происходит автоматически: перед началом выполнения функции выделяется, после завершения вызова функции очищается.

Функции для управления динамической памятью доступны в стандартной библиотеке, которую необходимо подключить:

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

Выделение и удаление ранее выделенной непрерывной области памяти заданного размера n производится с помощью функций:

```
void *malloc(size_t n);  
void free(void *ptr);
```

Название функция от англ. memory allocate — выделить память и free — освободить.

Напомним, что тип `size_t` — это тот же тип, что `unsigned long`, который используют для хранения размеров памяти. Результатом является указатель универсального типа `void *`, который будет преобразован к конкретному указательному типу. Например, таким образом можно выделить память для 1 элемента типа `char`:

```
malloc(1);
```

У динамической памяти нет имён переменных, как у статической. Поэтому единственный способ работать с такой памятью — указатели:

```
char *ptr;  
ptr = malloc(1);  
*ptr = 'w';
```

Теперь в переменной `ptr` хранится адрес динамически выделенной памяти размером в 1 байт. При этом стоит отметить, что сама переменная `ptr` является статической и занимает 8 байт в 64-битной ОС.

Статическая память

переменная	ptr
адрес20
значение74

Динамическая память

адрес74
значение	'w'

В отличие от статической памяти по завершению работы динамическую память нужно очищать вручную:

```
free(ptr);
```

Для целого числа типа `int` потребуется резервировать память вызовом `malloc(4)`. Вместо явного указания в байтах рекомендуется использовать оператор `sizeof`, который определяет размер типа:

```
int *ptr = malloc(sizeof(int));
*ptr = 7;
free(ptr)
```

Оператор разыменования `*ptr`, применённый к указателю на тип `int`, даёт доступ сразу к 4 байтам.

13.2 Массив

Рассмотрим пример, когда необходимо динамически выделить память для массива из 5 целых чисел типа `int`, заполнить его числами от 10 до 14, распечатать массив и очистить память.

Для такого массива потребуется 20 байт:

```
int *ptr = malloc(5 * sizeof(int));
```

Для доступа к произвольному элементу массива используется привычная нотация с квадратными скобками:

```
ptr[3] = 13;
```

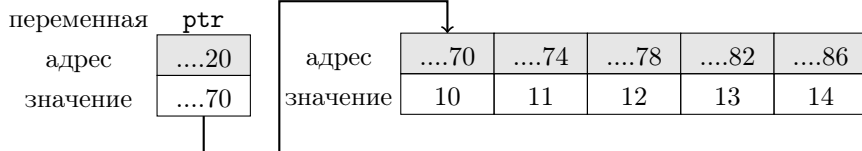
или используется арифметика указателей через оператор `*`:

```
*(ptr + 3) = 13;
```

Для адреса `ptr` равного `..70` указатель `ptr + 3` указывает на адрес `..82` (так как $70 + 3 \cdot 4 = 82$), что соответствует четвертому элементу массива. Подробнее об указателях описано в теме «Указатели и строки».

Статическая память

Динамическая память



Полная версия решения задачи с заполнением выглядит так:

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

int main() {
    int n = 5, i;
    int *ptr = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        ptr[i] = 10 + i;
    for (i = 0; i < n; i++)
        printf("%d ", ptr[i]);
    free(ptr);
    return 0;
}
```

Отличия от работы со статическим массивом незначительны: необходимо выделить память до начала работы и очистить её в конце.

13.3 Функции и динамическая память

Добавим две функции к решению предыдущей задачи: для генерации массива и для вывода массива. Первая функция будет принимать на вход n — размер массива, который нужно сгенерировать, и возвращать указатель на память, в которую записаны сгенерированные числа (`int *`). Вторая функция будет выводить массив на экран.

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

int *generate(int n) {
    int i;
    int *ptr = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
```

```
        ptr[i] = 10 + i;
    return ptr;
}

void print(int n, int *ptr) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", ptr[i]);
}

int main() {
    int *ptr = generate(5);
    print(5, ptr);
    free(ptr);
    return 0;
}
```

Обратите внимание, что память выделяется внутри одной функции, используется внутри другой, а очищает её вообще третья функция `main()`. Такую реализацию не получится сделать со статическим массивом.

13.4 Альтернативная функция для массивов

Наиболее неприятная ошибка при работе с динамической памятью (как и в принципе при программировании на C) — это ошибка сегментации. Возникает она при попытке разыменовать недоступную память. В случае с динамическими массивами это может произойти, если выделить памяти меньше, чем необходимо. Например, типичная ошибка: для массива из 10 чисел типа `int` выделяется 10 байт вместо 40 (забыли домножить на `sizeof(int)`):

```
int *ptr = malloc(10);
ptr[9] = 1;
```

Такую ошибку можно избежать, если выделить память для массива следующей функцией:

```
void *calloc(size_t n, size_t size);
```

Функция `calloc` выделяет память под массив из `n` элементов, каждый из которых имеет размер `size` байт (всего `n*size` байт) и инициализирует его нулями. Функция возвращает указатель на первый байт выделенной

области памяти как и функция `malloc`. Если выделить память не удалось, то функция возвращает нулевой указатель `NULL`.

Например:

```
int *ptr = calloc(10, sizeof(int));
ptr[9] = 1;
```

13.5 Неопределенное поведение программы

Если не выделить память как в примере ниже:

```
int *ptr;
ptr[9] = 0;
```

то в `ptr` будет мусорное значение. Данный код может привести к ошибке сегментации или непреднамеренно изменить другие переменные программы. Эта проблема была подробно разобрана в теме «Указатели и строки». Решением является инициализация указателя значением `NULL`:

```
int *ptr = NULL;
ptr[9] = 0;
```

Если у функции `malloc()` не получается выделить достаточно памяти (размер массива больше размера доступной памяти), то она тоже возвращает нулевой указатель `NULL`.

Функцию `free()` можно применять только для объектов, выделенных динамически (удалить статически выделенные массивы не получится). Причем удалять можно только блоки целиком. Например, вы не можете удалить последний байт строки:

```
char *ptr = malloc(10);
free(ptr + 9);
```

13.6 Расширение или уменьшение массива

Ещё одной интересной возможностью использования динамических массивов является изменение их размера.

Рассмотрим пример: необходимо сгенерировать массив из 5 элементов со значениями от 10 до 14, далее увеличить размер массива до 20.

Первая часть уже была решена ранее:

```
int n = 5;
int *ptr = malloc(n * sizeof(int));
```

```
for (i = 0; i < n; i++)  
    ptr[i] = 10 + i;
```

Решим вторую часть в три шага:

1. выделим другой массив с новым размером (под вспомогательным указателем `new_ptr`);
2. скопируем элементы из исходного массива во второй (из памяти под указателем `ptr` в память под указателем `new_ptr`);
3. очистим исходную память (под указателем `ptr`);

```
int m = 10;  
int *new_ptr = malloc(m * sizeof(int));  
for (i = 0; i < n; i++)  
    new_ptr[i] = ptr[i];  
free(ptr);
```

Функция, которая позволяет автоматизировать действия выше, есть в стандартной библиотеке:

```
void *realloc(void *ptr, size_t size);
```

Название функции от англ. memory reallocation — перевыделение памяти. В качестве аргументов передаются: указатель на уже имеющуюся память и новый размер в байтах. Результатом будет указатель на новую память, куда скопирован массив.

Перепишем код для предыдущего примера:

```
int m = 10;  
int *new_ptr = realloc(ptr, m * sizeof(int));
```

Использование функции для перевыделения памяти имеет свои преимущества:

1. очищение памяти под старым указателем внутри функции;
2. одинаковый интерфейс для уменьшения и для увеличения размера массива;
3. обработка пустого массива (`ptr == NULL`);
4. оптимизация при многократном увеличении массива на единицу.

Оптимизация происходит следующим образом: блоки памяти выделяются с запасом по степеням двойки. То есть, при увеличении массива с 4 до 5 байт выделится массив размера 8 байт. А дальнейшие запросы на увеличения с 5 до 6, с 6 до 7 и с 7 до 8 байт размер массива не изменят.

13.7 Массив для буферизации

Рассмотрим стандартную технику буферизации данных с применением расширяющегося динамического массива.

Пример 1. Дана последовательность целых чисел от 1 до 10^9 . После последовательности вводится нуль. Вывести элементы последовательности в обратном порядке.

Ввод	1 3 2 5 0
Вывод	5 2 3 1

Будем добавлять числа в массив по одному. Для массива заведём указатель (изначально нулевой) и счётчик элементов (изначально нуль):

```
int *array = NULL;
int n = 0;
```

Допустим, в массиве уже есть n элементов. Рассмотрим процесс добавления в конец массива значения из переменной `value` по шагам.

Во-первых, расширим массив до размера $n + 1$:

```
array = realloc(array, (n + 1) * sizeof(int));
```

Во-вторых, запишем в последний элемент массива `array` соответствующее значение (индекс последнего элемента будет как раз n , так как индексация начинается с нуля):

```
array[n] = value;
```

В-третьих, увеличиваем счётчик:

```
n++;
```

Данный порядок шагов верен и для добавления первого элемента, так как вызов:

```
array = realloc(NULL, sizeof(int));
```

эквивалентен вызову:

```
array = malloc(sizeof(int));
```

Осталось обработать вводимые числа в бесконечном цикле, из которого будем выходить при появлении нуля.

```
int *array = NULL;
int n = 0;
int value;
while (1) {
    scanf("%d", &value);
    if (value == 0)
        break;
    array = realloc(array, (n + 1) * sizeof(int));
    array[n] = value;
    n++;
}
```

Дальнейшие действия с массивом ничем не отличаются от действий со статическими массивами. При выходе из цикла число n будет в точности равно количеству элементов последовательности. Возможен альтернативный вариант — без оператора `break`. Эту реализацию приведём в виде полного решения исходной задачи:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int *array = NULL;
6      int n = 0;
7      int value;
8      scanf("%d", &value);
9      while (value != 0) {
10         int newsize = (n + 1) * sizeof(int);
11         array = realloc(array, newsize);
12         array[n] = value;
13         n++;
14         scanf("%d", &value);
15     }
16     int i;
17     for (i = n - 1; i >= 0; i--)
18         printf("%d ", array[i]);
19     putchar('\n');
20     free(array);
21     return 0;
22 }
```

Также не забываем чистить память: `free(array)`.

13.8 Динамические структуры

Очень интересным и наиболее часто применяемым объектом являются динамические структуры данных, такие как списки и деревья. Мы не будем подробно останавливаться на них в текущем курсе. Отметим лишь, что такие структуры не имеют последовательного доступа (как массив), но зато позволяют очень эффективно добавлять, удалять и обрабатывать данные.

13.9 Сравнение статической и динамической памяти

Формализуем отличия статической и динамической памяти.

Во-первых, именованные переменные. Статическая память выделяется благодаря именованному переменным, динамическая память выделяется без переменных.

Во-вторых, время жизни. Память, выделенная статически, автоматически очищается при завершении текущего блока (функции или составного оператора). Память, выделенная динамически, очищается только при вызове функции `free`.

В-третьих, место для памяти. Статическая память – это часть системного стека (`stack`). Динамическая память – это часть кучи (`heap`). И размер кучи, и размер стека в нашем курсе ограничен стандартными настройками компилятора `gcc`. Это означает, что статические переменные выделяются в памяти последовательно друг за другом, а очищаются в обратном порядке. Динамическая память выделяется более сложным образом, и это не всегда последовательные блоки памяти при последовательных запросах.

В-четвертых, объем памяти для массива. При выделении памяти под статический массив вида `int a[10]` можно узнать размер массива `sizeof(a)`. При выделении динамического массива `int *a = malloc(10 * sizeof(int))` узнать его размер не представляется возможным, так как `sizeof(a)` выдаёт размеры переменной `a`. В целом статический массив на n элементов требует только $n * \text{sizeof}(\text{type})$ памяти. А динамический массив, к тому же, дополнительно требует память под переменную, в которой будет храниться указатель на начало массива (в 64-битной ОС это 8 байт), то есть $n * \text{sizeof}(\text{type}) + \text{sizeof}(\text{type}^*)$.

В-пятых, представление матриц. Данный аспект будет подробно раскрыт в следующей теме.

13.10 Передача аргументов функции по указателю

Пример 2. На первой строке дано целое положительное число n . На второй строке массив из n целых чисел. Вывести только чётные числа массива. Описать функции для ввода массива, для фильтрации массива (создать массив, содержащий только чётные элементы входного массива) и для вывода массива.

Ввод	3 14 11 12
Вывод	14 12

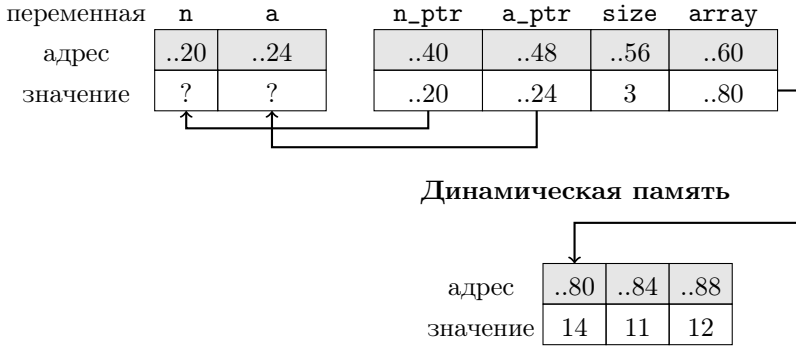
Выделение памяти и ввод чисел без функции выглядит следующим образом:

```
int *array = NULL;
int size, i;
scanf("%d", &size);
array = malloc(size * sizeof(int));
for (i = 0; i < size; i++)
    scanf("%d", &array[i]);
```

В локальной переменной `int size` сохранится размер массива, в локальной переменной `int *array` — адрес начала массива.

Функция может возвращать только одно значение. Поэтому вернуть и размер массива, и указатель на первый элемент обычным оператором `return` не получится. Используем косвенное изменение внешних переменных через указатели `int *n_ptr` и `int **a_ptr` на переменные функции `main` с размером и адресом начала массива. Схематически это будет выглядеть так:

Статическая память main Статическая память scan_array



Чтобы переменной `n` присвоить размер массива, необходимо по адресу `..20` из переменной `n_ptr` положить значение переменной `size`. Аналогичные действия выполняем для адреса начала массива. Итоговый прототип функции:

```
void scan_array(int *size_ptr, int **array_ptr);
```

Перед завершением перенесём полученные значения (адрес начала массива и размер) в соответствующие переменные функции `main` через указатели на них:

```
*n_ptr = size;
*a_ptr = array;
```

При вызове указываем адреса переменных:

```
scan_array(&n, &a);
```

Функция фильтрации работает по аналогии с функцией ввода, только здесь будет 4 параметра: размер и начало входного массива, размер и начало выходного массива. Входные параметры передаются по значению, а выходные параметры – по указателю.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void scan_array(int *size_ptr,
5                 int **array_ptr);
6 void even_filter(int src_size,
```

```
7             int *src_array,
8             int *dst_size_ptr,
9             int **dst_array_ptr);
10 void print_array(int size, int *array);
11
12 int main() {
13     int *a, *b;
14     int n, m;
15     scan_array(&n, &a);
16     even_filter(n, a, &m, &b);
17     free(a);
18
19     print_array(m, b);
20     if (b != NULL)
21         free(b);
22     return 0;
23 }
24
25 void scan_array(int *n_ptr, int **a_ptr) {
26     int *array = NULL;
27     int size, i;
28     scanf("%d", &size);
29     array = malloc(size * sizeof(int));
30     for (i = 0; i < size; i++)
31         scanf("%d", &array[i]);
32
33     *n_ptr = size;
34     *a_ptr = array;
35 }
36
37 void even_filter(int src_size,
38                 int *src_array,
39                 int *dst_size_ptr,
40                 int **dst_array_ptr) {
41     int i, size = 0, newsize;
42     int *array = NULL;
43     for (i = 0; i < src_size; i++)
44         if (src_array[i] % 2 == 0) {
45             newsize = (size + 1) * sizeof(int);
46             array = realloc(array, newsize);
47             array[size] = src_array[i];
```



```

48         size++;
49     }
50     *dst_size_ptr = size;
51     *dst_array_ptr = array;
52 }
53
54 void print_array(int size, int *array) {
55     int i;
56     for (i = 0; i < size; i++)
57         printf("%d ", array[i]);
58     putchar('\n');
59 }

```

13.11 Примеры

Пример 3. Дано целое число n от 1 до 100. В динамический массив записать все простые числа, которые не превосходят n . Вывести количество простых чисел и сами числа по возрастанию.

Ввод	8	2
Вывод	4 2 3 5 7	0

Каждое число будем проверять на делимость (от 2 до n). Если число делится, то оно не является простым. Для удобства обернём это в виде функции. Добавление в динамический массив реализуем с помощью функции `realloc`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int isprime(int d) {
5      for (int i = 2; i < d; i++)
6          if (d % i == 0)
7              return 0;
8      return 1;
9  }
10
11 int main() {
12     int* ans = NULL;
13     int m = 0, i, n;

```

```

14     scanf("%d", &n);
15     for (i = 2; i <= n; i++)
16         if (isprime(i)) {
17             int newsize = (m + 1) * sizeof(int);
18             ans = realloc(ans, newsize);
19             ans[m] = i;
20             m++;
21         }
22     printf("%d:\n", m);
23     for (i = 0; i < m; i++)
24         printf("%d ", ans[i]);
25     printf("\n");
26
27     if (ans != NULL)
28         free(ans);
29     return 0;
30 }

```

Обратите внимание, что перед очисткой памяти `free(ans)`; необходимо убедиться, что она вообще выделялась, иначе при $n = 1$ на 27 строке произойдёт ошибка сегментации данных.

Пример 4. На первой строке дано целое положительное число n . На следующих n строках дана информация о студентах: фамилия (не превышает 10 символов) и возраст (от 1 до 100). Найти самого старшего (или несколько самых старших).

Ввод	5 Bolotnikov 20 Gazizov 19 Kisselev 20 Korobov 19 Pikulina 18
Вывод	Bolotnikov Kisselev

Будем использовать динамический массив структур:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Student {
5      char name[11];

```

```

6      int age;
7  };
8
9  int main() {
10     struct Student *students;
11     int n, i, max;
12     scanf("%d", &n);
13     students = malloc(n * sizeof(struct Student));
14     for (i = 0; i < n; i++)
15         scanf("%s %d",
16             students[i].name,
17             &students[i].age);
18
19     max = s[0].age;
20     for (i = 0; i < n; i++)
21         if (max < students[i].age)
22             max = students[i].age;
23
24     for (i = 0; i < n; i++)
25         if (max == students[i].age)
26             puts(students[i].name);
27     free(students);
28     return 0;
29 }

```

Пример 5. Даны слова, разделённые пробелом (заранее неизвестной длины). После последнего слова стоит точка. Вывести все слова, которые начинаются и заканчиваются на одну и ту же букву.

Ввод	He was a free runner.
Вывод	a runner

Будем собирать текст по буквам в слова с помощью расширения массива. Слово закончилось, если встретился финальный символ (точка) или разделитель (пробел):

```

n = 0;
word = NULL;
ch = getchar();
while (ch != delimiter && ch != final) {
    word = realloc(word, (n + 1) * sizeof(char));

```

```

        word[n] = ch;
        n++;
        ch = getchar();
    }

```

После того как слово собрано, необходимо проверить первую и последнюю букву на совпадение. Если они одинаковы, то вывести слово. Чтобы воспользоваться функцией `puts()` для вывода строки, в конце массива букв добавим нулевой символ:

```

word = realloc(word, (n + 1) * sizeof(char));
word[n] = '\0';
puts(word);

```

Также важно не забывать чистить память после каждого слова:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char* word = NULL;
6      char ch, final = '.', delimiter = ' ';
7      int n, newsize;
8
9      do {
10         n = 0;
11         word = NULL;
12         ch = getchar();
13         while (ch != delimiter && ch != final) {
14             newsize = (n + 1) * sizeof(char);
15             word = realloc(word, newsize );
16             word[n] = ch;
17             n++;
18             ch = getchar();
19         }
20
21         if (n > 0) {
22             if (word[0] == word[n-1]) {
23                 newsize = (n + 1) * sizeof(char);
24                 word = realloc(word, newsize);
25                 word[n] = '\0';
26                 puts(word);
27             }

```

```

28             free(word);
29         }
30     } while (ch != final);
31     return 0;
32 }

```

Пример 6. Дана строка из слов. Слова разделены пробелами, каждое слово состоит из печатных символов, отличных от пробела, табуляции и переноса строки. Считать каждое слово в динамический массив. Отфильтровать в каждом слове только буквы и вывести с помощью `puts()` полученное слово.

```

char *get_word(char *last_char_ptr);
char *alpha_filter(char *word);

```

Ввод	[MSU] {Kaz}akhstan 2019 As-ta-na
Вывод	MSU Kazakhstan Astana

Описание функции `get_word()` было в предыдущем примере выше. Возвращаемым значением будет указатель на начало слова. Чтобы понять, какой символ шёл непосредственно за словом, будем возвращать последний символ (передавая по указателю соответствующую переменную). Если этот символ является переносом строки, то данное слово было последним.

Фильтрацию можно произвести двумя способами:

- за две итерации, то есть сначала найти длину ответа, а потом сразу выделить соответствующую память (с помощью `malloc()`) и скопировать элементы;
- за один проход, то есть расширять итоговый массив поэлементно как только попадаете подходящий символ.

Второй подход уже неоднократно рассматривался, поэтому приведём решение с двумя итерациями и однократным выделением памяти. Для удобства проверки символа на принадлежность к буквам будем использовать `isalpha(char)` из заголовочного файла `ctype.h`.

Алгоритм сжатия строки реализуем с помощью двух указателей: `i` указывает на обрабатываемый символ исходной строки, `filtered_i` указывает на позицию, следующую за последним добавленным символом.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4
5  char *get_word(char *last_char_ptr);
6  char *alpha_filter(char *word);
7
8  int main() {
9      char last_char = '\0';
10     char *word = NULL, *filtered_word;
11     do {
12         word = get_word(&last_char);
13         filtered_word = alpha_filter(word);
14         free(word);
15         puts(filtered_word);
16         free(filtered_word);
17     } while (last_char != '\n');
18     return 0;
19 }
20
21 char *get_word(char *last_char_ptr) {
22     char delimiter = ' ', final = '\n';
23     char *word = NULL, ch;
24     int n = 0, size = 0;
25
26     ch = getchar();
27     while (ch != delimiter && ch != final) {
28         size = (n + 1) * sizeof(char);
29         word = realloc(word, size);
30         word[n] = ch;
31         n++;
32         ch = getchar();
33     }
34     size = (n + 1) * sizeof(char);
35     word = realloc(word, size);
36     word[n] = '\0';
37     *last_char_ptr = ch;
38     return word;
39 }
40
```

```
41 char *alpha_filter(char *word) {
42     char *filtered_word = NULL;
43     int filter_size = 0, i, filtered_i;
44     for (i = 0; word[i]; i++)
45         if (isalpha(word[i]))
46             filter_size++;
47
48     int size = filter_size * sizeof(char);
49     filtered_word = malloc(size);
50
51     filtered_i = 0;
52     for (i = 0; word[i]; i++)
53         if (isalpha(word[i])) {
54             filtered_word[filtered_i] = word[i];
55             filtered_i++;
56         }
57     filtered_word[filtered_i] = '\\0';
58     return filtered_word;
59 }
```

13.12 Задания для самостоятельной работы

1. Написать часть кода, которая динамически выделяет память под ячейку типа `long long` и записывает в неё значение 2019LL.
2. Описать 3 отличия статической и динамической памяти.
3. Написать часть кода, которая динамически выделяет память под строку `math`, заполняет её и выводит на экран функцией `puts()`.
4. Дан динамически выделенный массив из 10 чисел типа `double`. Написать цикл, который будет выводить элементы массива с конца, при этом на каждом шаге удалять выведенный элемент массива.
5. Выбрать корректные функции создания динамического массива из чисел {1, 2, 3} типа `int`. Прокомментируйте некорректные варианты:

а)

```
void scan_array(int *a) {
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
}
...
int *array = NULL;
scan_array(array);
```

б)

```
void scan_array(int *a) {
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
}
...
int *array = malloc(12);
scan_array(array);
```

в)

```
void scan_array(int *a) {
    a = malloc(12);
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
}
```



```
...  
    int *array = NULL;  
    scan_array(array);
```

г)

```
void scan_array(int **a) {  
    int *ptr = malloc(12);  
    ptr[0] = 1;  
    ptr[1] = 2;  
    ptr[2] = 3;  
    *a = ptr;  
}  
...  
    int *array = NULL;  
    scan_array(&array);
```

д)

```
int *scan_array() {  
    int *ptr = malloc(12);  
    ptr[0] = 1;  
    ptr[1] = 2;  
    ptr[2] = 3;  
    return ptr;  
}  
...  
    int *array = NULL;  
    array = scan_array();
```

е)

```
void scan_array(int *a) {  
    a = malloc(12);  
    a[0] = 1;  
    a[1] = 2;  
    a[2] = 3;  
}  
...  
    int *array = malloc(12);  
    scan_array(array);
```

13.13 Практикум на ЭВМ

1. Дано целое число n от 1 до 1000. Сгенерировать динамический массив из квадратов натуральных чисел $1^2, 2^2, 3^2, \dots, n^2$. Вывести данный массив.

Ввод	5
Вывод	1 4 6 9 25

2. На первой строке дано целое положительное число n от 1 до 100. На второй строке дана последовательность из n целых чисел от -1000 до 1000. Добавить чётные числа в динамический расширяющийся массив. Вывести количество чётных чисел и сами числа.

Ввод	6 1 -2 8 -3 0 4
Вывод	4 -2 8 0 4

3. На первой строке дано целое положительное число n от 1 до 100. На второй строке дана последовательность из n целых чисел от -1000 до 1000. Считать данные числа в динамический массив и вывести их. Описать функции ввода (возвращает количество считанных элементов и через указатель возвращает указатель на динамический массив) и вывода:

```
int scan(int **array_ptr);
void print(int size, int *array);
```

Ввод	6 11 12 13 14 15 16
Вывод	11 12 13 14 15 16

4. На первой строке дано целое положительное число n от 1 до 100. На второй строке дана последовательность из n целых чисел от -1000 до 1000. Поменять первый и последний элемент массива местами. Описать соответствующие функции:

```
int scan(int **array_ptr);
void exchange(int size, int *array);
void print(int size, int *array);
```

Ввод	6 11 12 13 14 15 16	3 11 12 13
Вывод	16 12 13 14 15 11	13 12 11

5. На первой строке дано целое положительное число n от 1 до 100. На второй строке дана последовательность из n целых чисел от -1000 до 1000 . Считать данные числа в динамический массив. Скопировать чётные числа в другой массив. Вывести количество чётных чисел и сами числа. Описать соответствующие функции:

```
void scan(
    int *size_ptr,
    int **array_ptr
);
void even(
    int src_size,
    int *src_array,
    int *dst_size_ptr,
    int **dst_array_ptr
);
void print(
    int size,
    int *array
);
```

Ввод	6 1 -2 8 -3 0 4
Вывод	4 -2 8 0 4

6. На первой строке дано целое положительное число n от 1 до 100. На второй строке дана последовательность из n целых чисел от -1000 до 1000 . Считать её в расширяющийся буфер. Описать функцию для вычисления суммы всех чисел:

```
void scan(int *size_ptr, int **array_ptr);
int sum(int size, int *array);
```

Ввод	5 6 7 3 4 8 0
Вывод	33

7. Дано целое положительное число n от 1 до 1000000. Сохранить в динамический массив все делители числа n (по возрастанию) и распечатать его. Описать соответствующие функции:

```
void filter(int n,
           int *dividers_counter_ptr,
           int **dividers_array_ptr);
void print(int size, int *array);
```

Ввод	12
Вывод	1 2 3 4 6 12

8. Дана последовательность слов (заранее неизвестной длины), разделённых пробелом, которая заканчивается переносом строки. Распечатать каждое слово в обратном порядке (каждое слово считывать в динамический массив). Описать соответствующие функции:

```
char *get_word(int *word_size_ptr);
void print_reversed_word(
    int word_size,
    char *word
);
```

Ввод	Are you ready kids Aye Aye Captain
Вывод	erA erA ydaer sdik eyA eyA niatpaC

9. Даны две строки s_1 и s_2 заранее неизвестной длины. Считать их в динамические массивы (выделить память динамически). Сохранить их максимальный общий префикс в новый динамический массив. Описать соответствующие функции:

```
char *scan();
char *prefix(char *s1, char *s2);
```

Ввод	waterfall watermelon
Вывод	water

10. Дано время начало занятий в формате чч:мм и целое положительное число x от 1 до $24 \cdot 60$ — длительность одной пары вместе с переменной в минутах. Необходимо посчитать количество пар и вывести время начала пар, если известно, что последняя пара начинается не позже 17:00. Все отсечки сохранить в динамический массив и вывести. Описать соответствующие функции:

```
void print(struct Time current);

struct Time add_time(struct Time current,
                    int pair_time);

struct Time *generate_time(struct Time start,
                          int pair_time,
                          int *pair_count_ptr);
```

Функция `generate_time` генерирует динамический массив с отсечками, возвращая через указатель `pair_count_ptr` количество отсечек, а через возвращаемое значение — указатель на массив отсечек.

Ввод	09:00 90
Вывод	6 09:00 10:30 12:00 13:30 15:00 16:30

14 Динамические матрицы. Аргументы командной строки

В данной теме речь пойдет о применении динамической памяти для работы с матрицами. Также будут рассмотрены аргументы командной строки.

14.1 Статические матрицы

Рассмотрим пример статической матрицы порядка 2×3 :

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

В памяти такая матрица разложена по строкам:

адрес	..00	..04	..08	..12	..16	..20
значение	1	2	3	4	5	6
строки матрицы	a[0] (int[3])			a[1] (int[3])		
матрица	a (int[2][3])					

С каждой строкой можно работать как с обычным массивом, а вот со столбцами – нельзя. То есть и `a[0]`, и `a[1]` – это массивы типа (`int [3]`), состоящие из 3 элементов. При этом размер памяти для матрицы и отдельных строк будет определяться с учетом того, что каждый элемент занимает 4 байта:

```
sizeof(a) == 24
sizeof(a[0]) == 12
sizeof(a[1]) == 12
sizeof(a[1][2]) == 4
```

14.2 Статические матрицы как аргументы функции

При передаче массива в качестве аргумента функции аргумент будет приведён к типу указателя `int [3] → (int *)`, то есть данные два прототипа эквивалентны:

```
void f(int array[3]);
void f(int *array);
```

При передаче матрицы типа `int [2][3]` в качестве аргумента будет использоваться указатель типа `int (*)[3]`, то есть следующие прототипы одинаковы:

```
void f(int array[2][3]);  
void f(int (*array)[3]);
```

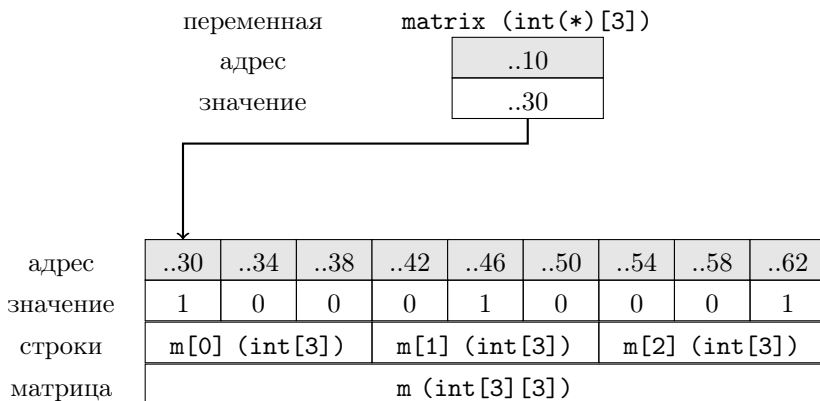
Обратите внимание на круглые скобки. Они явно указывают, что звёздочка относится к `array`, а не к `int`. Если же написать `int *array[3]`, то `array` будет массивом из 3 элементов типа `int *`, что фактически приведёт к типу `int **array`.

В случае с многомерными массивами указатель определяется через все размерности, кроме первой. Например, `int [2][3][4]` превратится в `int (*)[2][3]`.

Опишем функцию, которая инициализирует единичную матрицу размером 3×3 :

```
void eye(int matrix[3][3]) {  
    int i, j;  
    for (i = 0; i < 3; i++)  
        for (j = 0; j < 3; j++)  
            if (i == j)  
                matrix[i][j] = 1;  
            else  
                matrix[i][j] = 0;  
}  
int main() {  
    int m[3][3];  
    eye(m);  
    ...  
}
```

Внутри функции работа с переменной `matrix` производится естественным образом. Но при этом `matrix` – это не сама матрица, а указатель на область расположения реальной матрицы `m`.



14.3 Обнуление строки матрицы

Используя особенности укладки матрицы по строкам, можно инициализировать всю матрицу нулями с помощью функции:

```
void *memset(void *s, int c, size_t n);
```

из библиотеки `string.h`. Функция инициализирует n байт значениями c , начиная с адреса s :

```
#include <string.h>
void eye(int m[3][3]) {
    memset(m, 0, 3 * 3 * sizeof(int));
    int i;
    for (i = 0; i < 3; i++)
        m[i][i] = 1;
}
int main() {
    int matrix[3][3];
    eye(matrix);
    ...
}
```

14.4 Два способа представления динамических матриц

Существует два стандартных способа выделить память под динамическую матрицу: либо уложить всю матрицу построчно в виде одного дина-

мического массива, либо создать дополнительный массив указателей на одномерные динамические массивы. Рассмотрим оба способа на примере создания матрицы размером 2×3 с элементами типа `int`.

В первом способе достаточно выделить память под одномерный массив, который по размеру соответствует всей матрице:

```
int *a = malloc(2 * 3 * sizeof(int));
```

Для такого массива вместо двойной индексации `a[i][j]` необходимо применять обращения вида: `a[3 * i + j]`. Например, изменить правый нижний элемент матрицы можно следующим образом:

```
a[3 * 1 + 2] = 6;
```

После завершению работы необходимо очистить память:

```
free(a);
```

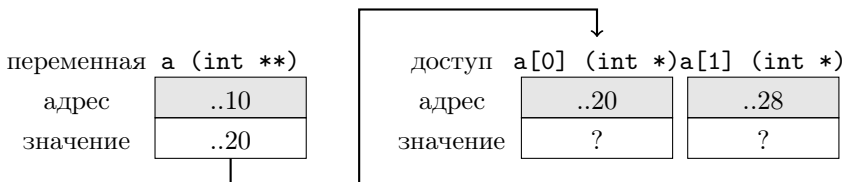
Во втором способе, каждая строка матрицы — это отдельный динамический массив, а указатели на эти массивы хранятся в отдельном массиве указателей. Такой массив состоит из элементов типа `int *`, то есть указатель на такой массив будет иметь тип `int **`:

```
int **a = malloc(2 * sizeof(int *));
```

Каждый элемент равен `sizeof(int *)` байт:

Статическая память

Динамическая память



Значения `a[0]` и `a[1]` заполним указателями на два отдельных блока динамической памяти для первой и второй строки матрицы соответственно:

```
a[0] = malloc(3 * sizeof(int));
a[1] = malloc(3 * sizeof(int));
```

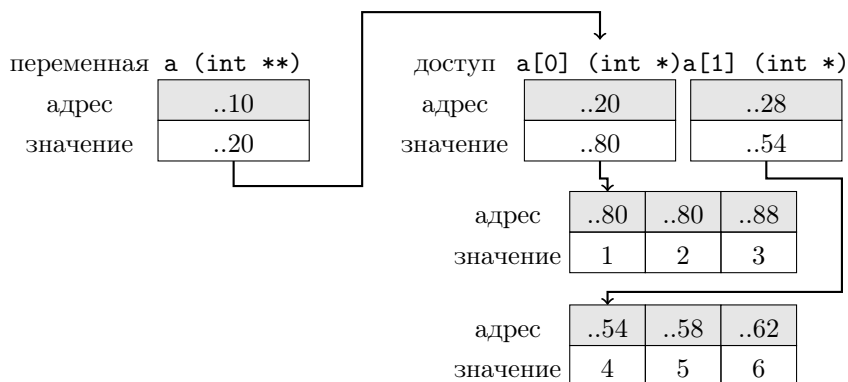
Теперь элементы `a[0]` и `a[1]` — это указатели на динамически выделенную память. Соответственно, к элементам матрицы можно обращаться как в статической матрице:

```
a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 4;
a[1][1] = 5;
a[1][2] = 6;
```

Получается, что каждая строка матрицы хранится независимо от остальных. А массив указателей на эти строки объединяет их всех в матрицу.

Статическая память

Динамическая память



Очищать память необходимо в обратном порядке: сначала динамически выделенные строки матрицы, а потом вспомогательный массив для хранения указателей на них:

```
free(a[0]);
free(a[1]);
free(a);
```

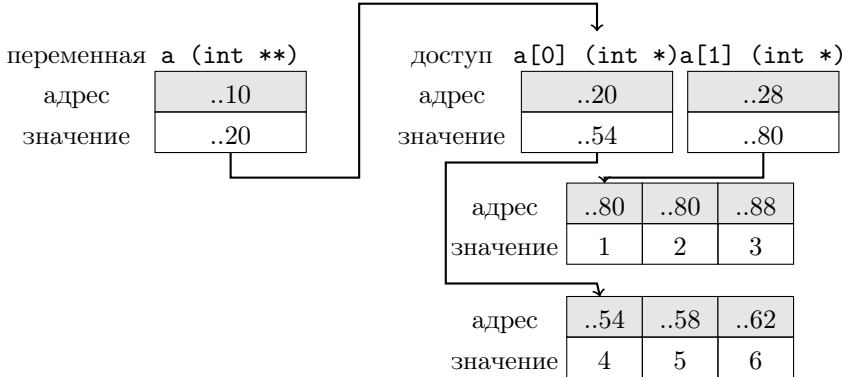
14.5 Быстрая перестановка строк

Представление динамических матриц вторым способом имеет следующее преимущество: строки матрицы можно переставлять без копирования всех элементов строк. Например, чтобы поменять первую и вторую строку матрицы местами, достаточно заменить указатели: в `a[0]` записать адрес `..54`, а в `a[1]` — адрес `..80`:

```
int *tmp = a[0];
a[0] = a[1];
a[1] = tmp;
```

Статическая память

Динамическая память



Далее будут рассматриваться только динамические матрицы, представленные вторым способом.

14.6 Динамические матрицы как аргументы функции

Пример 1. Дано целое число n от 1 до 100. Сгенерировать единичную матрицу соответствующего размера и вывести её. Реализацию описать в функции `main()`.

Ввод	1	3
Вывод	1	1 0 0 0 1 0 0 0 1

Выделить память под каждую из n строк следует в цикле:

```
a = malloc(n * sizeof(int *));
for (i = 0; i < n; i++)
    a[i] = malloc(n * sizeof(int));
```

Заполнение матриц и вывод значений элементов матрицы на экран ничем не отличается от варианта со статической матрицей. Но важно не забыть очистить динамическую память после того, как она уже не нужна.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int n, i, j;
6      int **a;
7
8      scanf("%d", &n);
9      a = malloc(n * sizeof(int *));
10     for (i = 0; i < n; i++)
11         a[i] = malloc(n * sizeof(int));
12
13     for (i = 0; i < n; i++)
14         for (j = 0; j < n; j++)
15             if (i == j)
16                 a[i][i] = 1;
17             else
18                 a[i][j] = 0;
19
20     for (i = 0; i < n; i++) {
21         for (j = 0; j < n; j++)
22             printf("%2d", a[i][j]);
23         putchar('\n');
24     }
25
26     for (i = 0; i < n; i++)
27         free(a[i]);
28     free(a);
29     return 0;
30 }
```

Пример 2. Дано целое число n от 1 до 100. Сгенерировать единичную матрицу соответствующего размера и вывести её. Реализацию описать в трёх функциях (генерация динамической единичной матрицы данного размера, вывод матрицы экран, очистка памяти):

```
int **eye(int n);
void print(int n, int **a);
```

```
void clear(int n, int **a);
```

Ввод	1	3
Вывод	1	1 0 0 0 1 0 0 0 1

Обнулять строки будем с помощью строковой функции

```
1 void *memset(void *s, int c, size_t n);
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int **eye(int n);
6 void print(int n, int **a);
7 void clear(int n, int **a);
8
9 int main() {
10     int n;
11     int **array;
12     scanf("%d", &n);
13     array = eye(n);
14     print(n, array);
15     return 0;
16 }
17
18 int **eye(int n) {
19     int i;
20     int **a = malloc(n * sizeof(int *));
21     for (i = 0; i < n; i++) {
22         a[i] = malloc(n * sizeof(int));
23         memset(a[i], 0, n * sizeof(int));
24     }
25     for (i = 0; i < n; i++)
26         a[i][i] = 1;
27     return a;
28 }
29
30 void print(int n, int **a) {
```

```
31     int i, j;
32     for (i = 0; i < n; i++) {
33         for (j = 0; j < n; j++)
34             printf("%2d", a[i][j]);
35         putchar('\n');
36     }
37 }
38
39 void clear(int n, int **a) {
40     int i;
41     for (i = 0; i < n; i++)
42         free(a[i]);
43     free(a);
44 }
```

14.7 Отличия динамической и статической матрицы

1. Размер требуемой памяти для матрицы размера $n \times m$, где каждый элемент имеет размер s байт, а указатель — p байт. В случае со статической памятью общий размер требуемой памяти: nms . В случае с динамической памятью: $(1 + n)p$ для указателей и nms для самой матрицы.
2. Динамические матрицы хранятся в виде n непрерывных блоков по строкам, а статические — единым цельным блоком. Поэтому такие операции как перестановка строк выполняются очень просто: перестановкой соответствующих указателей.
3. В динамических матрицах размеры двух разных строк одной матрицы могут отличаться.

14.8 Массив строк

При работе с несколькими строками их можно хранить в виде массива. Рассмотрим три способа хранения строк **one**, **two**, **three** в виде массива.

Вариант 1. Статическая матрица из символов. Для строк **one**, **two** достаточно выделить по 4 символа (в конце добавляем нулевой символ — символ конца строки). А вот для строки **three** потребуется уже 6 символов. Чтобы все строки поместились, нужна матрица размером 3×6 . Заполнить её можно следующим образом:

```
char a[3][6];
strcpy(a[0], "one");
strcpy(a[1], "two");
strcpy(a[2], "three");
```

Строки будут записаны в памяти следующим образом:

a[0]						a[1]						a[2]					
..00	..01	..02	..03	..04	..05	..06	..07	..08	..09	..10	..11	..12	..13	..14	..15	..16	..17
o	n	e	∅	?	?	t	w	o	∅	?	?	t	h	r	e	e	∅

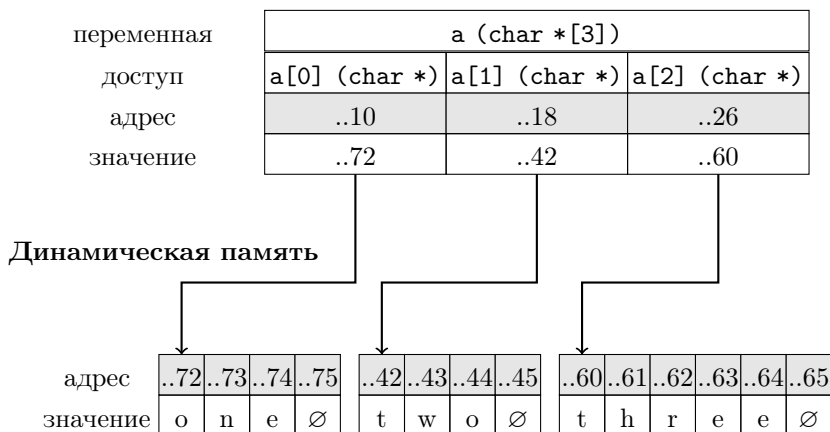
Пожалуй, не лучший способ для хранения строк с различной длиной, так как часть памяти не используется.

Вариант 2. Статический массив из 3 строк, где каждая строка является динамическим массивом. То есть, это будет массив из 3 элементов типа `char *`. Для строк выделим ровно столько памяти, сколько им нужно:

```
char *a[3];
a[0] = malloc(4);
strcpy(a[0], "one");
a[1] = malloc(4);
strcpy(a[1], "two");
a[2] = malloc(6);
strcpy(a[2], "three");
```

Использование памяти в этом случае более эффективно:

Статическая память



Вариант 3. Динамический массив из 3 строк, где каждая строка является динамическим массивом. Всё аналогично предыдущему примеру, только массив указателей будет тоже динамическим:

```
char **a;
a = malloc(3 * sizeof(char *));
a[0] = malloc(4);
strcpy(a[0], "one");
a[1] = malloc(4);
strcpy(a[1], "two");
a[2] = malloc(6);
strcpy(a[2], "three");
```

Во всех трех случаях работать со строкам можно естественным образом:

```
if (strcmp(a[0], a[1]) > 0)
    puts(a[0]);
else
    puts(a[1]);
```

Для динамически выделенных строк нужно всегда помнить о размере реально выделенной памяти. В частности, нужно крайне аккуратно копировать их, чтобы при этом всегда хватало памяти:


```
strcpy(str[0], str[1]);
strcpy(str[0], str[2]);
strcpy(str[2], str[0]);
```

В последней строке при попытке записать 6 байт в массив из 4 байт произойдёт ошибка сегментации.

Пример 3. Дано целое число n . Далее n строк. Необходимо записать все строки в динамический массив строк и вывести строки длиннее 10 символов.

Ввод	6 whoooo ... who lives in a pineapple under the sea Sponge Bob Square Pants
Вывод	in a pineapple under the sea Square Pants

Данный код собирает одну строку в динамически расширяющийся массив, добавляя символ конца строки:

```
char *word = NULL;
int n = 0;
char ch = getchar();
while (ch != '\n') {
    word = realloc(word, (n + 1) * sizeof(char));
    word[n] = ch;
    n++;
    ch = getchar();
}
word = realloc(word, (n + 1) * sizeof(char));
word[n] = '\0';
```

Теперь достаточно n раз вызвать функцию с данным кодом и заполнить динамический массив строк из указателей `word`.

Вводить число n необходимо в форматной строке с символом переноса строки:

```
scanf("%d\n", &n);
```

В противном случае этот перенос будет первым символом первой строки.

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

char *get_string();

int main() {
    char **strings = NULL;
    int n, i;
    scanf("%d\n", &n);
    strings = malloc(n * sizeof(char *));
    for (i = 0; i < n; i++)
        strings[i] = get_string();
    for (i = 0; i < n; i++)
        if (strlen(strings[i]) > 10)
            puts(strings[i]);
    for (i = 0; i < n; i++)
        free(strings[i]);
    free(strings);
    return 0;
}

char *get_string() {
    char *string = NULL;
    int n = 0;
    char ch = getchar();
    int newsize = 0;
    while (ch != '\n') {
        newsize = (n + 1) * sizeof(char);
        string = realloc(string, newsize);
        string[n] = ch;
        n++;
        ch = getchar();
    }
    newsize = (n + 1) * sizeof(char);
    string = realloc(string, newsize);
    string[n] = '\0';
    return string;
}
```

14.9 Аргументы командной строки

При запуске команд терминала можно использовать специальные аргументы. Например, вывод команды без аргументов:

```
$ ls
```

отличается от вывода команды с аргументами:

```
$ ls -l -a
```

Второй вариант выдает более подробную информацию о файлах в директории.

При компиляции `gcc`:

```
$ gcc prog.c -o prog -Wall
```

используется целых 4 аргумента: `prog.c`, `-o`, `prog`, `-Wall`.

Для того чтобы получать аргумент в нашей программе, необходимо воспользоваться следующим прототипом основной функции:

```
int main(int argc, char** argv);
```

При запуске программы параметры `argv` и `argc` будут содержать аргументы командной строки и их количество, включая название вашей программы.

При запуске без дополнительных параметров:

```
$ ./prog
```

значение переменной `argc` будет равно 1, а в переменной `argv` будет находиться указатель на строку `./prog` размером 7 байт (не забываем про нулевой символ в конце строки). Данный код будет выводить имя исполняемого файла:

```
#include <stdio.h>

int main(int argc, char **argv) {
    puts(argv[0]);
    return 0;
}
```

Вывод значимых аргументов, которые идут после самой программы, можно организовать следующим образом:

```
#include <stdio.h>
int main(int argc, char** argv)
{
    int i;
    printf("%d:\n", argc - 1);
    for (i = 1; i < argc; i++)
        puts(argv[i]);
    return 0;
}
```

Например, при запуске:

```
$ ./prog one two three
```

на выводе мы получим

```
3:
one
two
three
```

14.10 Преобразование чисел и строк

Аргумент командной строки является ещё одним средством ввода данных в программу в виде строк. Для того чтобы преобразовать строки в числа, можно использовать некоторые функции из коллекции стандартной библиотеки `stdlib.h`:

```
int atoi(const char *str)
double atof(const char *str)
long long atoll(const char *str)
```

Эти функции переводят строковое представление чисел в числа типа `int`, `double` и `long long` соответственно (от англ. array to int, array to float, array to long long).

Пример 4. Найти сумму двух целых чисел от -1000 до 1000 , которые передаются в качестве аргументов командной строки.

Запуск	<code>./prog 10 14</code>
Вывод	24

С учетом, того, что нумерация начинается с нуля, первый аргумент, который является числом, записан во второй строке `argv[1]`, а второй аргумент — в третьей строке `argv[2]`. Соответственно, решить задачу можно так:

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

int main(int argc, char **argv) {
    int a, b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("%d\n", a + b);
    return 0;
}
```

Если запустить программу без аргументов, то `argv[1]` приведёт к ошибке сегментации данных. Поэтому необходимо проверять количество аргументов. Если их количество отличается от требуемого, то следует досрочно завершить выполнение программы с предупреждением:

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

int main(int argc, char **argv) {
    if (argc != 3) {
        puts("Need 2 arguments");
        return 1;
    }
    int a, b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("%d\n", a + b);
    return 0;
}
```

В случае со стандартным потоком ввода перевод строкового представления чисел в числовое осуществлялся с помощью функции `scanf()`, а обратное действие — с помощью `printf()`. В заголовочном файле `stdio.h` описаны аналоги для записи числовых значений в строку и считывания числовых значений из строки:

```
int sprintf(char *str, const char *format, ...);
```

```
int sscanf(const char *str, const char *format, ...);
```

Например, после такого кода:

```
int a;
sscanf("2018", "%d", &a);
```

в переменной `a` окажется число 2018. А после следующий строк

```
char s[5];
sprintf(s, "%d", 2019);
```

в строке `s` окажется строка «2019».

Альтернативное решение исходной задачи:

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

int main(int argc, char **argv) {
    if (argc != 3) {
        puts("Need 2 arguments");
        return 1;
    }
    int a, b;
    sscanf(argv[1], "%d", &a)
    sscanf(argv[2], "%d", &b)
    printf("%d\n", a + b);
    return 0;
}
```

Пример 5. Вывести самый длинный аргумент командной строки (без учета имени программы). Если таких аргументов несколько, то вывести последний такой аргумент.

Запуск	<code>./prog There were bells on a hill</code>
Вывод	<code>bells</code>

Чтобы найти последний максимальное значение, достаточно использовать стандартный поиск максимума в массиве с нестрогим сравнением.

Длину строки можно легко узнать с помощью функции `size_t strlen(const char *s)`; из библиотеки `string.h`.

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

```
int main(int argc, char** argv)
{
    if (argc == 1) {
        puts("Need more arguments.");
        return 1;
    }
    int i, imax, max, current;
    max = strlen(argv[1]);
    imax = 1;
    for (i = 2; i < argc; i++) {
        current = strlen(argv[i]);
        if (max <= current) {
            max = current;
            imax = i;
        }
    }
    puts(argv[imax]);
    return 0;
}
```

14.11 Задания для самостоятельной работы

1. Какие присваивания допустимы между указанными переменными (среди всех возможных пар)?

```
int a[2][3];  
int (*b)[3];  
int *c[3];  
int **d;
```

2. Описать часть кода, которая выделяет память под нижне-треугольную матрицу размером $n \times n$ из вещественных чисел (память для элементов выше главной диагонали выделять не надо).
3. Дана динамическая матрица размером $n \times n$ из чисел типа `int`. При каком n аналогичная статическая матрица будет на 2020 байт меньше на 32-битной ОС.
4. Написать программу, которая печатает своё имя при запуске из текущей директории (имя запускаемого файла без префикса `./`). Например, при запуске `./prog`, вывести `prog`.
5. Написать программу, которая выводит квадрат вещественного числа, переданного в качестве первого аргумента командной строки. Например, при запуске `./prog 1.2` вывести `1.44`.

14.12 Практикум на ЭВМ

1. Дано целое положительное число n . Далее матрица размером $n \times n$ из целых чисел. Сохранить элементы в динамическую матрицу. Найти сумму элементов каждой строки и произведение элементов каждого столбца.

Ввод	3
	1 2 3
	4 5 6
	7 8 9
Вывод	6 15 24
	28 80 162

2. Дано целое положительное число n . Далее матрица размером $n \times n$ из целых чисел. Сохранить элементы в динамическую матрицу. Переставить первую и последнюю строки местами (без поэлементного копирования элементов, только изменив адреса строк).

Ввод	4
	1 2 3 4
	2 3 4 5
	3 4 5 6
	4 5 6 7
Вывод	4 5 6 7
	2 3 4 5
	3 4 5 6
	1 2 3 4

3. Даны целые положительные числа n и m . Далее матрица размером $n \times n$ из целых чисел. Сохранить элементы в динамическую матрицу. Вывести транспонированную матрицу. Описать функцию, которая создает динамическую матрицу, которая является транспонированной к исходной матрице:

```
int **transpose(int n, int m, int **matrix);
```

Ввод	3 4 1 2 3 4 5 6 7 8 9 10 11 12
Вывод	1 5 9 2 6 10 3 7 11 4 8 12

4. Дано целое положительное n . Далее матрица размером $n \times n$ из целых чисел от -100 до 100 и вектор-столбец размера n . Найти произведение матрицы $n \times n$ и вектор-столбца.

Ввод	2 1 2 4 5 3 6
Вывод	15 42

5. Дано целое положительное число n . Далее массив из n целых чисел. Сохранить числа в динамический массив. Сгенерировать динамическую матрицу типа циркулянт и вывести. Циркулянт — матрица, в которой каждая строка матрицы получается из предыдущей циклическим сдвигом влево.

Ввод	3 1 2 3
Вывод	1 2 3 2 3 1 3 1 2

6. Дано целое положительное число n . Далее дано n строк со словами (первая буква заглавная, остальные строчные). Вывести слова, которые идут лексикографически раньше последнего слова из списка. Описать функцию, которая запишет строку в динамический массив:

```
char *get_word();
```

Ввод	5 Manny Sid Diego Shira Shangri
Вывод	Manny Diego

7. Программа запускается с несколькими аргументами, которые являются вещественными числами. Вывести максимальное значение с 2 знаками после запятой.

Запуск	./prog -1 32e1 153.34
Вывод	320.00

8. Программа запускается с несколькими аргументами. Найти самый длинный аргумент и распечатать его.

Запуск	./prog Bond James Bond
Вывод	James

9. В качестве первого аргумента дан шаблон строки, состоящий из букв, точек и одного символа %. В качестве второго аргумента дано целое число n от 1 до 9. Необходимо сгенерировать динамический массив из n динамических строк, где вместо % подставлены цифры от 1 до n .

Запуск	./prog homework%.c 4
Вывод	homework1.c homework2.c homework3.c homework4.c

10. Программа запускается с несколькими аргументами. Определить порядковые номера аргументов, которые совпадают с названием программы (без точки и слеша).

Запуск	./prog I wrote a prog and run the prog
Вывод	3 8

15 Файлы

В данной теме описана работа с текстовыми файлами, которые позволяют сохранять результаты работы программы в постоянную память и получать данные из неё.

15.1 Открытие файла

Чтобы открыть файл, нужно указать имя файла и режим работы с файлом:

```
FILE *fopen(char *name, char *mode);
```

Параметр `name` включает в себя адрес и имя файла, с которым планируется работа. Адрес указывается относительно исполняемого файла вашей программы (если файл находится в одной директории с программой, то достаточно написать просто его название).

Параметр `mode` задает режим работы с файлом: «**r**» – чтение файла, «**w**» – перезапись файла (стирает содержимое, которое было в файле), «**a**» – добавление в конец файла (оставляет содержимое, которое было в файле).

Функция возвращает указатель на структуру, тип которой определён в макроопределении `FILE`. Память для этой структуры выделяется непосредственно в самой функции. Для дальнейшей работы понадобится только указатель на неё.

Например, открыть файл `input.txt` для чтения можно следующим образом:

```
FILE *input_file;  
input_file = fopen("input.txt", "r");
```

А так открывается файл для записи:

```
FILE *output_file;  
output_file = fopen("output.txt", "w");
```

После работы с файлом важно корректно закрыть его:

```
int fclose(FILE *stream)
```

Если после открытия файла его не закрыть, то можно столкнуться с проблемами. Во-первых, данные, которые вы отправили на запись, могут не записаться в файл (функции записи гарантируют запись в буфер оперативной памяти, а только `fclose` гарантирует «безопасное завершение»

работы с файлом по аналогии с USB-флешкой). Во-вторых, на каждый активный открытый файл операционная система расходует системные ресурсы.

15.2 Чтение и запись

Благодаря функциям, описанным в библиотеке `stdio.h`, ввод и вывод из файлов аналогичен работе со стандартным вводом и выводом.

Ввод-вывод отдельного символа:

```
int fgetc(FILE *stream);
int putc(char ch, FILE *stream);
```

Ввод-вывод обычной строки:

```
char *fgets(char *s, int size, FILE *stream);
int fputs(const char *s, FILE *stream);
```

Ввод-вывод форматной строки:

```
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

Результатом является количество успешно введенных или выведенных символов.

Пример 1. Записать в файл `abc.txt` все буквы английского алфавита.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      FILE *fd;
6      char ch;
7
8      fd = fopen("abc.txt", "w");
9      if (fd == NULL)
10         return 0;
11
12     for (ch = 'a'; ch <= 'z'; ch++)
13         fputc(fd, ch);
14
15     fclose(fd);
16     return 0;
```

```
17 }
```

Компилируем:

```
$ gcc prog.c -o prog -Wall
```

Запускаем:

```
$ ./prog
```

Проверяем содержимое файла:

```
$ cat abc.txt
```

Получаем на выводе алфавит.

15.3 Конец файла

При чтении из файла данные считываются последовательно. Как только файл заканчивается, возникает ситуация «конец файла». Все функции ввода реагируют на конец файла по-разному. Например:

```
int fgetc(FILE *stream);
```

возвращает константу EOF (фактическое значение -1);

```
char *fgets(char *s, int size, FILE *stream);
```

возвращает нулевой указатель NULL (фактическое значение 0);

```
int fscanf(FILE *stream, const char *format, ...);
```

возвращает константу EOF (значение -1).

В терминале ситуацию конца файла можно смоделировать комбинацией `ctrl+d`. Если в строке имеется какой-либо текст, то такая комбинация эмулирует нажатие `enter`. В противном случае комбинация воспроизводит ситуацию конца файла. Таким образом, чтобы завершить ввод концом файла в терминале, достаточно либо нажать `enter` и потом `ctrl+d`, либо дважды нажать `ctrl+d`.

Пример 2. Вывести содержимое файла, имя которого передаётся через аргумент командной строки, то есть реализовать простейший вариант команды `cat`.

f.txt	Hello world!
Запуск	./prog f.txt
Вывод	Hello world!

Считывать символы будем в переменную `char ch`; до тех пор, пока `ch != EOF`. При этом каждый символ будем выводить на экран.

В программе стоит обработать два вида ошибок: недостаточное количество аргументов при запуске и невозможность открытия файла (в частности, если этот файл отсутствует).

```

1  #include <stdio.h>
2
3  int main(int argc, char **argv) {
4      if (argc != 2) {
5          puts("Add filename as argument.");
6          return 1;
7      }
8      FILE *fd = fopen(argv[1], "r");
9      if (fd == NULL) {
10         puts("Can not open the file.");
11         return 2;
12     }
13     char ch = fgetc(fd);
14     while (ch != EOF) {
15         putchar(ch);
16         ch = fgetc(fd);
17     }
18     fclose(fd);
19     return 0;
20 }
```

Пример 3. На стандартный ввод дано имя файла (не более 100 символов). В файле записаны целые числа от -1000 до 1000 через пробел. Количество чисел неизвестно. Найти их сумму.

num.txt	1 2 3 5
Ввод	num.txt
Вывод	11

Если для считывания имени файла использовать функцию `fgets`, то после значащих символов будет символ переноса строки. То есть, вместо 8 байт будет сохранено 9 байт:

'n'	'u'	'm'	'.'	't'	'x'	't'	'\n'	'\0'
-----	-----	-----	-----	-----	-----	-----	------	------

При попытке открытия файла произойдёт ошибка, так как в имени файла нет переноса строки. Поэтому следует убрать этот символ вручную или с

помощью функции `char *strchr(const char *s, int c)`; из библиотеки `string.h`:

```
char file_name[102];
fgets(file_name, 101, stdin);
char *position = strchr(file_name, '\n');
if (position)
    *position = '\0';
```

Если для считывания использовать функцию `scanf`, то ввод значительно упростится:

```
char file_name[101];
scanf("%s", file_name);
```

Ввод чисел из файла будет осуществляться при помощи функции `fscanf` до тех пор, пока не появится конец файла.

```
#include <stdio.h>
int main(int argc, char **argv) {
    char file_name[101];
    scanf("%s", file_name);

    FILE *fd = fopen(file_name, "r");
    if (fd == NULL) {
        puts("Can not opent the file.");
        return 1;
    }
    int sum = 0, value;
    while (fscanf(fd, "%d", &value) != EOF)
        sum += value;
    printf("%d\n", sum);
    fclose(fd);
    return 0;
}
```

Пример 4. Скопировать содержимое из одного файла в другой. Имена передаются через аргументы командной строки, то есть реализовать простейший вариант команды `cp`.

from.txt	Very important data.
Запуск	./prog from.txt to.txt
to.txt	Very important data.


```
#include <stdio.h>
int main(int argc, char **argv) {
    if (argc != 3) {
        puts("Add 2 filenames as argument.");
        return 1;
    }

    FILE *file_src = fopen(argv[1], "r");
    if (file_src == NULL) {
        puts("Can not open source file.");
        return 2;
    }
    FILE *file_dst = fopen(argv[2], "w");
    if (file_src == NULL) {
        fclose(file_src);
        puts("Can not open destination file.");
        return 3;
    }

    char ch = getc(file_src);
    while(ch != EOF) {
        putc(ch, file_dst);
        ch = getc(file_src);
    }
    fclose(file_dst);
    fclose(file_src);
    return 0;
}
```

15.4 Перенаправление потоков ввода-вывода

Допустим, у нас имеется программа, которая работает со стандартным вводом и выводом. Например, данная программа считывает один символ и выводит его на экран.

```
#include <stdio.h>
int main() {
    char ch = getchar();
    putchar(ch);
    return 0;
}
```

Необходимо сделать так, чтобы ввод производился из файла `input.txt`, а вывод в файл `output.txt`. Как можно минимальными усилиями заменить стандартный ввод-вывод на ввод-вывод в файл?

Вариант 1. Перенаправление внутри кода функцией `freopen`. В библиотеке `stdio.h` имеется функция замены одного потока на поток, ассоциированный с файлом:

```
FILE *freopen(const char *path,
              const char *mode,
              FILE *stream);
```

Аргумент `path` — имя файла, аргумент `mode` — режим (чтения или записи), `stream` — указатель на файл. У любой программы имеется 3 стандартных потока: ввод, вывод и вывод ошибок. Для каждого из них определен указатель типа `FILE`: `stdin`, `stdout`, `stderr`. Их можно использовать в качестве потока `stream`.

Заменяем стандартный ввод на ввод из файла `input.txt`, а стандартный вывод на вывод в файл `output.txt`:

```
#include <stdio.h>
int main() {
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    char ch = getchar();
    putchar(ch);
    return 0;
}
```

Теперь первый символ из файла `input.txt` будет скопирован в файл `output.txt`.

Если информацию в выходной файл нужно дописывать, а не удалять, то открываем файл с соответствующим режимом:

```
freopen("output.txt", "a", stdout);
```

После такой замены потока стандартные потоки ввода и вывода закрываются, то есть выполняются `fclose(stdin)` и `fclose(stdout)`. Поэтому вернуть обратно вывод на экран простым действием уже не получится.

Вариант 2. Стандартные потоки можно перенаправить в файлы средствами терминала. Для этого во время запуска можно добавить имя файла после символа больше `>` (перенаправить вывод) или символа меньше `<` (перенаправить ввод). Например, так:

```
$ ./prog < input.txt > output.txt
```

Для дописывания текста в конец файла используется двойной знак больше:

```
$ ./prog >> output.txt
```

Также можно управлять и потоком ошибок, дописав второй поток непосредственно перед знаком больше или двойного больше:

```
$ ./prog 2>> error.log
```

15.5 Пример работы с графическим файлом

Разберём один интересный пример: редактирование графического файла формата bmp (изображение без сжатия).

Файл в формате bmp (bitmap picture) начинается с заголовка с описанием параметров файла, после которого следует описание пикселей изображения в виде матрицы. В простейшем варианте заголовок занимает 54 байта, а каждый пиксель описывается 3 числами, которые соответствуют интенсивности синей, зеленой и красной компоненты цвета. В самом компактном варианте для каждой компоненты цвета каждого пикселя используется 1 байт памяти. Таким образом матрица пикселей будет иметь размер $h \times w$, где каждый элемент занимает ровно 3 байта. Создать такой файл можно в редакторе `gimp`:

```
$ sudo apt install gimp
```

Далее готовим файл `input.bmp`:

- открываем любое изображение в `gimp`;
- выбираем в меню: изображение — размер холста;
- выравниваем и ширину, и высоту на **числа кратные 4** не более 252×252 (например, 100 на 100);
- выбираем в меню: файл — экспортировать как — изображение windows bmp;
- выбираем параметры совместимости: **не сохранять данные о цветовом формате**;
- выбираем дополнительные параметры: **24 разряда (R8 G8 B8)**;

- нажимаем: экспортировать.

Чтобы убедиться, что файл в правильном формате:

```
$ file input.bmp
```

На выходе получим:

```
input.bmp:
PC bitmap, Windows 3.x format, 100 x 100 x 24
```



Проверим размер файла (англ. disk usage):

```
$ du -b input.bmp
30054  input.bmp
```

Он должен быть равен $(3 \cdot w \cdot h + 54)$ байт.

Пример 5. Дано изображение в формате .bmp (24 бита на пиксель). Необходимо выделить границы по заданному уровню L на этом изображении и сохранить полученный результат в другой файл. Граница в пикселе p_{ij} выделяется чёрным, если хотя бы для одного из каналов верно неравенство

$$|p_{ij} - p_{ij-1}| + |p_{ij} - p_{i-1j}| > L$$

input.bmp	
Запуск	<code>./prog input.bmp output.bmp 40</code>
output.bmp	

Разделим всю задачу на три вспомогательные подзадачи:

1. открыть файл (скопируем все пиксели из данного графического файла в динамическую матрицу);
2. изменить файл (из данной динамической матрицы создадим новую, в которой будут выделены границы);
3. сохранить файл (данную динамическую матрицу пикселей сохраним как графический файл).

Хранить изображение будем в динамической матрице из пикселей, где каждый пиксель – это структура с 3 полями типа `unsigned char`, которые отвечают за синий, зеленый и красный канал:

```
1 struct Pixel {
2     unsigned char b, g, r;
3 };
```

Опишем функции создания и удаления динамической матрицы данного размера:

```
1 struct Pixel **create_image(
2     int width,
3     int height
4 ) {
5     int y;
6     struct Pixel **image = NULL;
7     int size = height * sizeof(struct Pixel *);
8     image = malloc(size);
9     for (y = 0; y < height; y++) {
10         int size = width * sizeof(struct Pixel);
11         image[y] = malloc(size);
12     }
13     return image;
14 }
15
16 void free_image(int width,
17                int height,
18                struct Pixel **image) {
19     int y;
20     for (y = 0; y < height; y++)
21         free(image[y]);
22     free(image);
```

Первая подзадача. Опишем функцию, которая открывает данный графический файл и переносит все данные в динамическую матрицу, по шагам:

1. открываем файл;
2. считываем заголовок по байтам (их будет 54);
3. узнаём размеры картинки (ширина и высота находятся в 18 и 22 байте заголовка);
4. создаём динамическую матрицу;
5. заполняем динамическую матрицу;
6. закрываем файл.

Важно отметить, что заголовок нам ещё понадобится, когда мы будем сохранять новый файл. Также нам нужны будут размеры изображения. Поэтому в качестве аргументов передаём: имя файла, размер заголовка, указатель на строку с заголовком (заполним в функции), указатели на переменные для ширины и высоты изображения (заполним в функции). Результатом будет указатель на память с динамической матрицей пикселей:

```
1 struct Pixel **open_bmp_24(char *file_name,
2                             int header_size,
3                             unsigned char *header,
4                             int *width_ptr,
5                             int *height_ptr)
6 {
7     FILE *file = fopen(file_name, "r");
8     if (file == NULL)
9         return NULL;
10
11     int i;
12     for (i = 0; i < header_size; i++)
13         header[i] = fgetc(file);
14
15     int width = header[18];
16     int height = header[22];
17
18     struct Pixel **image;
```

```

19     image = create_image(width, height);
20
21     int x, y;
22     for (y = 0; y < height; y++) {
23         for (x = 0; x < width; x++) {
24             image[y][x].b = fgetc(file);
25             image[y][x].g = fgetc(file);
26             image[y][x].r = fgetc(file);
27         }
28     }
29
30     fclose(file);
31     *width_ptr = width;
32     *height_ptr = height;
33     return image;
34 }

```

Здесь можно отметить, что если числа, описывающие размеры не будут помещаться в 1 байт, то следует считывать сразу 4 байта. Это можно сделать так: вычисляем адрес, далее приводим указатель к указателю на `int` и забираем содержимое:

```

int width = *(int*)(header + 18);
int height = *(int*)(header + 22);;

```

Вторая подзадача. Опишем функцию, которая выделяет границы:

1. создаём динамическую матрицу для результата;
2. для каждого элемента исходной матрицы (кроме крайнего ряда слева и снизу) вычисляем, насколько отличаются цвета

$$|p_{ij} - p_{ij-1}| + |p_{ij} - p_{i-1j}|$$

3. если эта величина больше заданного порога, то записываем черный цвет (0, 0, 0), иначе – белый (255, 255, 255).

```

1 int check(struct Pixel **src,
2           int level,
3           int x,
4           int y)
5 {
6     int db = abs(src[y][x].b - src[y][x-1].b) +

```

```

7         abs(src[y][x].b - src[y-1][x].b);
8     int dg = abs(src[y][x].g - src[y][x-1].g) +
9         abs(src[y][x].g - src[y-1][x].g);
10    int dr = abs(src[y][x].r - src[y][x-1].r) +
11        abs(src[y][x].r - src[y-1][x].r);
12    return db > level || dg > level || dr > level;
13 }
14
15 struct Pixel **border_detect(int width,
16                             int height,
17                             struct Pixel **src,
18                             int level)
19 {
20     struct Pixel **dst;
21     dst = create_image(width, height);
22     int x, y;
23     struct Pixel white = {255, 255, 255};
24     struct Pixel black = {0, 0, 0};
25     for (y = 0; y < height; y++)
26         for (x = 0; x < width; x++) {
27             dst[y][x] = white;
28             if (x > 0 && y > 0) {
29                 if (check(src, level, x, y))
30                     dst[y][x] = black;
31             }
32         }
33     return dst;
34 }

```

Третья подзадача. Опишем функцию, которая сохраняет в графический файл данную динамическую матрицу:

1. открываем файл;
2. записываем заголовок по байтам (их будет 54);
3. записываем динамическую матрицу;
4. закрываем файл.

```

1 int save_bmp_24(char *file_name,
2                 int header_size,
3                 unsigned char *header,

```



```

4             int width,
5             int height,
6             struct Pixel **image)
7 {
8     FILE *file = fopen(file_name, "w");
9     if (file == NULL)
10         return 1;
11
12     int i;
13     for (i = 0; i < header_size; i++)
14         fputc(header[i], file);
15
16     int x, y;
17     for (y = 0; y < height; y++) {
18         for (x = 0; x < width; x++) {
19             fputc(image[y][x].b, file);
20             fputc(image[y][x].g, file);
21             fputc(image[y][x].r, file);
22         }
23     }
24
25     fclose(file);
26     return 0;
27 }

```

Все три вспомогательные задачи решены. Осталось описать основную программу:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv)
5  {
6      if (argc != 4)
7      {
8          puts("Argument format:");
9          puts("./prog input.bmp output.bmp level");
10         return 1;
11     }
12
13     int width, height;
14     const int header_size = 54;

```

```
15     unsigned char header[header_size + 1];
16     struct Pixel **image;
17     image = open_bmp_24(argv[1],
18                           header_size,
19                           header,
20                           &width,
21                           &height);
22     if (image == NULL)
23     {
24         puts("Can not open bmp file.");
25         return 2;
26     }
27
28     struct Pixel **new_image;
29     new_image = border_detect(width,
30                               height,
31                               image,
32                               atoi(argv[3]));
33
34     if (save_bmp_24(argv[2],
35                     header_size,
36                     header,
37                     width,
38                     height,
39                     new_image))
40     {
41         puts("Can not save bmp file.");
42         return 3;
43     }
44     free_image(width, height, image);
45     free_image(width, height, new_image);
46     return 0;
47 }
```

15.6 Задания для самостоятельной работы

1. Почему необходимо закрывать файл с помощью функции `fclose`?
2. Опишите, как можно узнать размер файла в байтах?
3. В качестве аргумента командной строки подаётся целое число n . Написать программу, которая печатает в файл `file.txt` все числа от 1 до n через пробел.
4. В качестве аргументов командной строки подаются имена двух файлов. Скопировать из первого файла во второй только цифры.
5. Что выполняет программа? По каким причинам программа может не завершиться корректно?

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

int main(int argc, char **argv){
    char fname[10], ch;
    FILE *fd;
    strcpy(fname, argv[0] + 2);
    strcpy(fname + strlen(fname), ".c");
    fd = fopen(fname, "r");
    while ((ch = fgetc(fd)) != EOF) {
        fputc(ch, stdout);
    }
    fclose(fd);
    return 0;
}
```

15.7 Практикум на ЭВМ

1. Дано целое число n от 1 до 26. Записать в файл `ascii.txt` первые n букв алфавита английского алфавита.

Ввод	3
<code>ascii.txt</code>	abc

2. Даны целые число n и m от 1 до 10. Записать в файл `table.txt` таблицу умножения $n \times m$.

Ввод	3 5
<code>ascii.txt</code>	1 2 3 4 5 2 4 6 8 10 3 6 9 12 15

3. На стандартный поток ввода подается имя файла. Найти общее количество символов в файле.

<code>input.txt</code>	Bimbo's flight!
Ввод	<code>input.txt</code>
Вывод	15

4. На стандартный поток ввода подается 2 имени файлов. Скопировать содержимое первого файла во второй, оставив только буквы и пробелы.

<code>input.txt</code>	Bimbo's flight!
Ввод	<code>input.txt output.txt</code>
<code>output.txt</code>	Bimbos flight

5. На стандартный поток ввода подается 2 имени файлов. Скопировать первый файл во второй в обратном порядке, если известно, что в первом файле не более 1000 символов.

<code>input.txt</code>	No lemon, no melon
Ввод	<code>input.txt output.txt</code>
<code>output.txt</code>	nolem on ,nomel oN

6. Программа запускается с аргументом, который является именем файла. Распечатать содержимое файла.

input.txt	we all live in a yellow submarine
Запуск	./prog input.txt
Вывод	we all live in a yellow submarine

7. Программа запускается с аргументом, который является именем файла. Вывести количество строк и слов в файле.

input.txt	we all live in a yellow submarine
Запуск	./prog input.txt
Вывод	Lines: 2 Words: 7

8. Программа запускается с аргументами, которые являются именами файлов. Необходимо вывести, какие файлы можно открыть на чтение, а какие — нет.



Запуск	./prog prog.c prog
Вывод	prog.c: readable prog: unreadable

9. Программа запускается с аргументом, который является целыми числом n . Сгенерировать файл с названием `matrix#.txt`, где вместо `#` стоит число n , в который записана единичная матрица $n \times n$.

Запуск	./prog 3
matrix3.txt	1 0 0 0 1 0 0 0 1

10. Дано изображение с шириной w и высотой h . Сделать эффект флага с параметром d . Каждый пиксель нового изображения $b_{i,j}$ заменить на пиксель старого изображения $a_{k,j}$, где

$$k = \left(i + d - d \sin \left(\frac{2\pi \cdot j}{w} \right) \right) \cdot \frac{h}{h + 2d}.$$

input.bmp	
Запуск	<code>./prog input.bmp output.bmp 20</code>
output.bmp	

16 Ответы к проверочным работам

Техническое введение. Терминал и консольные редакторы

1. `mkdir Lion.`
`cd Lion`
2. `nano Timon.txt`
печатаем текст «Awimbawe»
`ctrl + o`
`ctrl + x`
3. `cp Timon.txt Pumba.txt`
4. `nano Pumba.txt`
печатаем текст «Hakuna Matata»
`ctrl + k`
`ctrl + u`
`ctrl + u`
`ctrl + u`
`ctrl + o`
`ctrl + x`
5. `pwd`
`ls`
6. `cat Pumba.txt Timon.txt`
7. Нажимаем `tab` после `ls /media/`
`ls /media/user/1234-5678/`
8. `cp Pumba.txt /media/user/1234-5678/`
`umount /media/user/1234-5678/`
9. `sed 's/tata/th/g' < Pumba.txt`
10. `rm *`
`cd ..`
`rmdir Lion`

Числовые типы. Арифметические операторы

1. а) $0xBOBA > 11 \cdot 16^3 > 11 * 4096 > 30000$;
б) $0xFULL = 15 < 30000$;
в) $0xCAFE > 12 \cdot 16^3 > 12 * 4096 > 30000$.
2. а) unsigned int; б) long long; в) float; г) double.
3. а) неверно (результат 4);
б) неверное (результат 4U);
в) верно;
г) верно;
д) неверно (результат 4.0);
е) верно;
ж) неверно (результат 4LL).
4. а) double (8 байт); б) int (4 байта); в) float (4 байта).
5. а) $((1 + 3) + 4) / 3 * 3 = 6$;
б) $(2 + (((100 / 3) \% 25) * 4)) - 1 = 33$;
в) $(1e1 + 2e2) - 1e-1 = 209.9$.

Переменные. Приведение типов. Первая программа

1. а) корректно (`double` \rightarrow `int`);
б) некорректно (имя переменной не может начинаться с цифры);
в) корректно (`double` \rightarrow `long long`);
г) некорректно (имя переменной не может содержать точку);
д) корректно (`unsigned long long` \rightarrow `short`);
е) корректно (`int` \rightarrow `double`).
2. а) верно;
б) верно;
в) неверно;
г) неверно;
д) неверно;
е) верно.
- 3.

строка	код	x	y
1	int x = 2, y = 3;	2	3
2	x = y - 2;	1	3
3	y = 10 + 5 * x;	1	15
4	x++;	2	15
5	y -= 2;	2	13
6	x *= x;	4	13
7	x -= -x;	8	13

4. ans = n / 10 + n % 10;

5. ans = (a + b) / 2.0 + 2 / (1.0 / a + 1.0 / b);

Условный оператор. Математические функции

1. а) $x \% 2 \neq 0 \ \&\& \ x < 0$ (или $x \% 2 == -1$);

б) $x < 0 \ || \ x \geq 30 \ \&\& \ x \neq 50$;

в) $x \geq 1000 \ \&\& \ x \leq 9999 \ \&\& \ x / 1000 == x \% 10 \ \&\& \ x / 100 \% 10 == x / 10 \% 10$;

2. $\text{abs}((x1 - x2) * (y1 - y2)) == 1 \ ||$
 $\text{abs}(x1 - x2) + \text{abs}(y1 - y2) == 1$

3. $x * x + y * y \leq 1 \ \&\& \ \text{fabs}(x + y) \geq 1$

```
4. #include <stdio.h>
#include <math.h>
int main(){
    int n;
    double S;
    scanf("%d", &n);
    if (n >= 3)
    {
        S = n * 0.5 * sin(2.0 * M_PI / n);
        printf("%.2lf\n", S);
    }
    else
        puts("Do not exist.");
    return 0;
}
```

```
5. #include <stdio.h>                                     //добавить
int main()
{
```

```

int a, b, c, ans;           //убрать int
scanf("%d %d %d", &a, &b, &c); //добавить &
if (a < b && a < c)         //добавить скобки
    ans = b + c ;          //добавить ;
if (b < c && b < a)         //убрать скобки
    ans = a + c;           //заменить == на =
if (c < a && c < b)         //убрать ;
    ans = a + b;           //заменить += на ==
printf("%d\n", ans);       //заменить / на \
return 0;                  //добавить 0
}

```

Цикл while

1. while(1)
 printf("Hello");

Остановить можно комбинацией `ctrl + c`.

2. 50 25 12 6 3 1 0
 7

3. 4 (число делителей числа n)

```

4. #include <stdio.h>
int main(){
    int a, ans = 0;
    scanf("%d", &a);
    while (a != 0){
        if (a < 0)
            ans++;
        scanf("%d", &a);
    }
    printf("%d\n", ans);
    return 0;
}

```

5. 5, т.к. это минимальное n такое, что $\frac{\pi}{2^n} < 0.1 \Leftrightarrow 2^n > 10\pi$, а именно столько раз будет производится деление, то есть тело цикла.

Символьный тип

1. if (ch >= 'A' && ch <= 'Z')
 ch += 'a' - 'A';

2. 9 символов, 4 строки.

```
$  
$  tnt      ntn  
$  
$  \\n
```

3. #include <stdio.h>

```
int main()  
{  
    char ch = 'Z' + 1;  
    while (ch < 'a')  
    {  
        printf("%d:%c\n", ch, ch);  
        ch++;  
    }  
    return 0;  
}
```

```
$ ./prog > ascii.txt
```

4. а) 1;

б) неверное выражение (деление на ноль);

в) 0;

г) $'2' * 3 - 2 * ('2' + 1) - ('2' - 1) = -1$;

д) 0.

5. Неверный результат: aa..

Верный результат: a b..

В общем виде неверно будет обрабатываться строка, в которой есть хотя бы одно слово с четным количеством букв а в конце.

Цикл for

1. \$ 30 10 3

2. \$ 6 5 4

\$ 7 6 5

\$ 8 7 6

\$ 9 8 7

3. подходят б), г), д)

4.

```
#include <stdio.h>
int main() {
    int n, i, a, answer = 0;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &a);
        if (a % 2 == 0)
            answer++;
    }
    printf("%d\n", answer);
    return 0;
}
```
5.

```
#include <stdio.h>
int main() {
    char r, c;
    char black = ('A' + '1') % 2;
    for (r = '8'; r >= '1'; r--) {
        for (c = 'A'; c <= 'H'; c++)
            if ((c + r) % 2 == black)
                putchar('X');
            else
                putchar('0');
        putchar('\n');
    }
    return 0;
}
```

Массивы

1. 2 3 (a[2] и a[5] соответственно).
2.

```
int t = a[49];
a[49] = a[98];
a[98] = t;
```
3.

```
for (i = 0; i < 26; i++)
    a[i] = 'A' + i;
```
4. После первого цикла массив count[] равен: 3 1 2 0 0 1 0 3 (обратите внимание что в массиве a[] имеется 3 нулевых значения). После второго цикла на выводе будет отсортированный массив a[]: 0 0 0 1 2 2 5 7 7 7

```

5.    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (i >= j && i + j <= n - 1)
                ans += a[i][j];

```

Строки и указатели

```

1. #include <stdio.h>
   #include <string.h>
   int main()
   {
       char str[100];
       unsigned long n;
       fgets(str, 100, stdin);
       /* str: 'T' 'o' 'w' 'e' 'r' '\n' '\0' */

       n = strlen(str) - 1;
       /* n = 5 */

       printf("%lu\n", n);
       puts(str + n / 2);
       /* str + 2: 'w' 'e' 'r' '\n' '\0' */

       return 0;
   }

```

2. Данный код вычисляет наибольший общий префикс. Максимум 99 символов (100 символ будет занят переносом строки, 101 символ — нулевым символом).

3. • 20 (печатаем `*(a+1) == a[1]`),
- 21 (изменяем значение `*(a+1))++ == a[1]++`),
- 30 (изменяем указатель `p = a + 2`).

4. а) можно

```

int a = 2;
int *p = &a;
a ** p; // a = a * (*p)

```

б) можно

```
int a = 2;
int *p = &a;
a /=* p; // a = a / (*p)
```

- в) нельзя (начало комментария)
г) можно

```
int a = 2;
int *p = &a;
a = a/ *p; // a = a / (*p)
```

5. \$ maths
\$ mecmath
\$ mathematics
\$ NO

Вызов функции `strstr(a, "a")` возвращает указатель на 5-ю букву строки `a[]`. Вызов функции `strncpy(strstr(a, "a"), b, 4)` изменяет 4 буквы строки `a[]` и возвращает указатель на 5-ю букву `a[]`.

Функции

1. $x = 2$, $y = 3$ и $z = 0$
2. Локальные переменные: $a = 2$ и $b = 3$ — локальных переменных функции `main` и глобальной переменной `a`:

```
int main() {
    int a = 0; //local a = 0, global a = 20
    a = f(10); //local a = 11, global a = 20
    a = f(a); //local a = 12, global a = 20
    a = g(); //local a = 21, global a = 21
    return 0;
}
```

3. Для статического массива: a , v , g , e , и. Для статической матрицы: a , v , g , $ж$.
4. При вызове функций `f2` и `f3` изменятся, а при вызове `f1` — не изменятся.

5.

```
int max2(int a, int b)
{
    if (a > b)
```

```

        return a;
    return b;
}
int max4(int a, int b, int c, int d)
{
    return max2( max2(a, b), max2(c, d) );
}

```

Функции как параметр. Рекурсия

1. Адрес возврата и локальные переменные, в том числе аргументы функции.

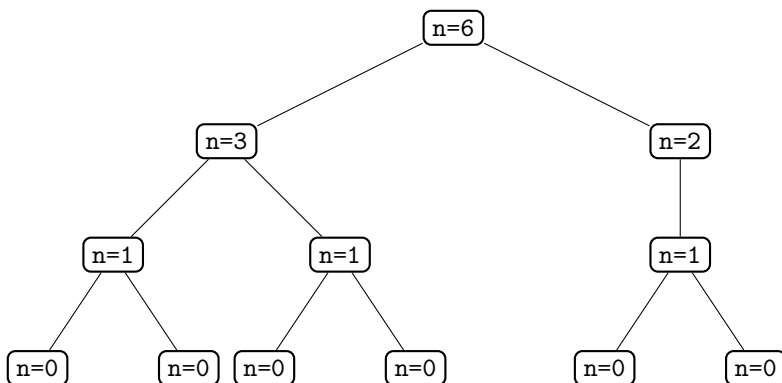
2. \$ 1 0 2

3. \$ [6 [3 [1 1] [1 1] 3] [2 [1 1] 2] 6]

```

4. int f(int n, int a[]) {
    if (n == 0)
        return 1;
    return f(n - 1, a) * a[n - 1];
}

```



```

5. int f() {
    char ch = getchar();
    if (ch == '\n')
        return 0;
    int ans = f();
    if (ch >= '0' && ch <= '9')
        ans += ch - '0';
}

```

```
        return ans;
    }
```

Структуры

1. 37 12

2. Варианты без использования особенности расположения в памяти:

```
example.x[0] = 'z';
example.x[1] = 'z';
example.y = 'z';

strcpy(example.x, "zz");
example.y = 'z';

MyStructure temp = {"zz", 'z'};
example = temp;
```

Варианты с использованием особенностей расположения в памяти:

```
memset(&example, 'z', 3);

example.x[0] = 'z';
example.x[1] = 'z';
example.x[2] = 'z';
```

```
3. struct First{
    int x;
};
struct Second{
    int x[6];
};

struct First a[6];
struct Second b;
struct Second c[6];
```

```
4. struct Point {
    int x, y;
};
struct Triangle {
```

```

    struct Point points[3];
};

struct Triangle egypt = {{0, 0}, {3, 4}, {0, 4}};

5.    int i;
      for (i = 0; i < 100; i++)
          if (heroes[i].name[0] == 'A' &&
              heroes[i].mana > 100)
              puts(heroes[i].name);

```

Динамические массивы и указатели

1.

```
long long *ptr = malloc(sizeof(long long));
ptr[0] = 2019LL; //или *ptr = 2019LL;
```
2. а) место выделения: стек и куча; б) автоматическое выделение/-
чистка памяти и ручное выделение/чистка памяти; в) наличие и
отсутствие именованных переменных, закрепленных за память.
3.

```
char *word = malloc(5);
strcpy(word, "math");
puts(word);
```
4.

```
int i = 9;
while (i > 0) {
    printf("%d\n", array[i]);
    array = realloc(array, i * sizeof(double));
    i--;
}
```
5. а) некорректно, так как память не выделена (произойдет ошибка
сегментации).
 б) корректно;
 в) некорректно, так как указатель на начало динамической памя-
 ти хранится в локальной переменной функции, соответственно,
 по завершению функции доступ к этой памяти будет безвоз-
 вратно потерян; а указатель **array** останется нулевым.
 г) корректно;
 д) корректно;

- е) некорректно, так как указатель на начало динамической памяти хранится в локальной переменной функции, соответственно, по завершению функции доступ к этой памяти будет безвозвратно потерян; а указатель `array` указывает на другую динамическую память.

Динамические матрицы и аргументы командной строки

1. `a` и `c` — статические массивы, значит, они не могут быть слева от присваивания. Остальные проверим по следующему правилу: разыменованные объекты должны быть соответствующего типа. Рассмотрим `b` слева:

```
b = a; //int (*)[3] <- int[2][3]
      //после разыменования: int [3] и int [3]
b = c; //int (*)[3] <- (int *)[3]
      //после разыменования: int [3] и int *
b = d; //int (*)[3] <- int **
      //после разыменования: int [3] и int *
```

Подходит только `b = a`. Рассмотрим `d` слева:

```
d = a; //int ** <- int[2][3]
      //после разыменования: int * и int [3]
d = b; //int ** <- int (*)[3]
      //после разыменования: int * и int [3]
d = c; //int ** <- (int *)[3]
      //после разыменования: int * и int *
```

Подходит только `d = c`.

Ответ: `b = a, d = c`.

2.

```
int i;
double **matrix = malloc(n * sizeof(double *));
for (i = 0; i < n; i++)
    matrix[i] = malloc((i + 1) * sizeof(double));
```
3. Динамическая матрица занимает на $(n + 1) * p$ байт больше, где p — размер указателя. То есть, $4(n + 1) = 2020$. Ответ: 504.
4.

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

```
    puts(argv[0] + 2);
    return 0;
}

5. #include <stdio.h>
   int main(int argc, char **argv) {
       double x = atof(argv[1]);
       printf("%lf\n", x * x);
       return 0;
   }
```

Файлы

1. Чтобы гарантировать запись результата во внешнюю память.

2.


```
FILE *fd = fopen("file.txt", "r");
int ans = 0;
while (fgetc(fd) != EOF)
    ans++;
```

3.


```
#include <stdio.h>
int main(int argc, char **argv) {
    FILE *file = fopen("file.txt", "w");
    int n = atoi(argv[1]);
    int i;
    for (i = 1; i <= n; i++)
        frprintf(file, "%d ", i);
    fclose(file);
    return 0;
}
```

4.


```
#include <stdio.h>
#include <ctype.h>
int main(int argc, char **argv) {
    FILE *file_src = fopen(argv[1], "r");
    FILE *file_dst = fopen(argv[2], "w");
    char ch = fgetc(file_src);
    while (ch != EOF) {
        if (isdigit(ch))
            fputc(ch, file_dst);
        ch = fgetc(file_src);
    }
    fclose(file_src);
```

```
        fclose(file_dst);  
        return 0;  
    }
```

5. Данная программы выводит текст этой программы (если имя исполняемого файла и исходного кода совпадают и лежат в одной директории). То есть, по названию исполняемого файла `./prog` программ пытается открыть файл `.prog.c`. Файл может не открыться, если имя превышает 10 символов или он отсутствует.

17 Задания для подготовки к промежуточному контролю

17.1 Теоретическая часть

1. Запишите команды терминала:
 - (a) Скомпилировать `program.c` в файл `program`.
 - (b) Выполнить программу `program`.
 - (c) Распечатать содержимое файла `program.c` на экран.
 - (d) Скопировать файла `program.c` в файл `home1.c`.
2. Определить максимальное и минимальное значение числа типа `int`. Записать два ответа: точный — в виде степени двойки, и приближенный — в десятичном виде.
3.
 - (a) Перечислите 2 различных целочисленных типа.
 - (b) Укажите размер переменных указанных типов в байтах.
 - (c) Перечислите 2 различных вещественных типа.
 - (d) Укажите размер переменных указанных типов в байтах.
4. Определить значения констант и их тип

Константа	Значение	Тип
<code>2e-1</code>		
<code>0xA8</code>		

5. Вычислить значение результата:
 - а) $2 / 6$;
 - б) $2.0 / 6LL$;
 - в) `'2' / '6'`.
6. Подписать порядок выполнения арифметических действий и вычислить значение:

$9 + 8 - 7 \% 6 * 5 / 4 + 3$

7. Дано положительное четырехзначное число в переменной `n`. Допisać условный оператор, который проверяет, что данное число является палиндромом (то есть при `n = 2018` вывод `no`, при `n = 2002` вывод `yes`).

```
if (                                     ) {
    puts("yes");
} else {
    puts("no");
}
```

8. Чем отличаются функции `abs()` и `fabs()`? Приведите пример неправильного вычисления модуля числа с помощью одной из этих функций.

9. Что выполняет данный участок кода?

```
char ch;
ch = getchar();
if (ch >= 'a' && ch <= 'z') {
    ch += 'A' - 'a';
}
putchar(ch);
```

10. Дано целое число `int n`; . Необходимо проверить, что оно нечетное. Чем плоха такая проверка?

```
if (n % 2 == 1) {
    puts("odd");
} else {
    puts("even");
}
```

11. Чему равно значение переменной `i` после выполнения операций (операции выполняются последовательно друг за другом).

```
int i = 2018;
i++;
i %= 10;
i -= 2;
i *= i;
```

12. Что напечатает данная часть кода?

```
int i;
for (i = -5; i < 5; i++) {
    if (i < 1) {
        printf("%d ", i);
    }
    if (i > -1) {
        printf("%d ", i);
    }
}
```

13. Даны шахматные координаты в виде структуры. Опишите логическое выражение, которое принимает истинное значение, если с поля f можно пойти на поле g ходом слона.

```
struct Chess {
    char row;
    int col;
};
```

14. Дано целое число n . Необходимо проверить, что оно нечетное. Приведите пример некорректной работы данного кода.

```
if (n % 2 == 1) {
    puts("odd");
} else {
    puts("even");
}
```

15. Что напечатает данная часть кода?

```
int i;
for (i = 0; i < 30; i += 2)
    if (i % 3 == 0)
        printf("%d", i);
```

16. Почему работает «оператор спуска»?

```
int x = 10;
while (x --> 0)
    printf("%d", x);
```

17. Что напечатает данная часть кода, если вводится целое число n ?

```
int n, i;
scanf("%d", &n);
for (i = 1; i <= n; i++) {
    while (n % i == 0) {
        n /= i;
        printf("%d\n", i);
    }
}
```

18. Что напечатает данная часть кода?

```
for (int i = 0; i < 5; i++) {
    printf("start d\n", i);
    if (i == 2)
        break;
    printf("finish %d\n", i);
}
```

19. Что будет распечатано?

```
int a[] = {11, 12, 13, 14, 15, 16} , i ;
for (i = 1; i < 6; i+=3) {
    printf("%d ", a[i]);
}
```

20. Поменяйте местами первый и последний элемент массива местами

```
int a[] = {11, 12, 13, 14, 15, 16};
```

21. Заполните массив счетчиков cnt , то есть $cnt[k]$ должно быть равно количеству появлений числа k в массиве a .

```
int a[] = {1, 4, 2, 4, 1, 4};
int cnt[5] = {0};
```

22. Распечатайте матрицу в стандартном виде через двойной цикл

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


23. Чем отличаются следующие объявления переменных?

```
char str[10] = "abc";
```

```
char str[] = "abc";
```

24. Скопируйте в строку `ans` текст `math` из первой строки с помощью функции копирования.

```
char url[] = "mymath.info";  
char ans[10];
```

25. Описать функцию `int mystrlen(char *ptr);`, которая возвращает длину строки по данному указателю (не использовать индексацию через квадратные скобки).

26. Как изменится массив после выполнения кода?

```
int a[] = {11, 12, 13, 14, 15}, i, tmp;  
for (i = 1; i <= 4; i--)  
    if (a[i] > a[i - 1]) {  
        tmp = a[i];  
        a[i] = a[i - 1];  
        a[i - 1] = tmp;  
    }
```

27. Чему равен массив `b` после выполнения кода?

```
int a[7] = {1, 2, 1, 3, 1, 4, 1},  
    b[7] = {1, 2, 3, 1, 2, 3, 4}, i;  
for (i = 0; i < 7; i++)  
    b[a[i]]++;
```

28. Что выполняет данный код?

```
char a[100], b[100], i;  
fgets(a, 99, stdin);  
fgets(b, 99, stdin);  
for (i = 0; a[i] && b[i]; i++) {  
    a[i] = b[i];  
}
```

29. Чему равно значение переменной x?

```
int f(int n, int d) {
    int ans = 0;
    for ( ; n > 0; n /= d)
        ans += n % d;
    return ans;
}
x = f(f(2020, 10), 2);
```

30. Как ускорить решение данной задачи с точки зрения теории чисел?

```
int check(int a, int b) {
    int d, ans = 1;
    for (d = 2; d <= a && d <= b; d++)
        if (a % d == 0 && b % d == 0)
            ans = d;
    return ans;
}
```

31. Что будет распечатано?

```
char a[] = "last work";
char b[10];
strncpy(b, a + 5, 4);
strncpy(b + 5, a, 4);
puts(a);
puts(b);
if (strcmp(a, b) < 0)
    puts(a);
else
    puts(b);
```

32. Изобразить схему расположения элементов в памяти:

```
int a[2][3] = {{11, 12, 13}, {14, 15, 16}};
```

Чему равны значения:

- (a) a[1][2]
- (b) a[0][4]
- (c) (*(a+1)+0)

33. Чем отличается ввод строк через *scanf*, *gets* и *fgets*.
34. Опишите функцию, которая проверяет, является ли данное число простым

```
int isprime(int n);
```

35. Опишите рекурсивную функцию, которая вычисляет наибольший общий делитель чисел *a*, *b*.

```
int gcd(int a, int b);
```

36. Опишите функцию, которая вычисляет максимум в массиве данного размера.
37. Приведите пример передачи в функцию параметра по значению и через указатель. В чем отличие для внешних переменных?

```
void f1(                                ) {  
}  
void f2(                                ) {  
}
```

38. Опишите структуру трехмерный вектор и функцию, которая вычисляет скалярное произведение двух трехмерных вектора.
39. Опишите структуру студент с двумя полями *name* — имя (строка не более 10 символов) и *mark* — оценка (целое число). Создайте 2 переменных, соответствующие студенту *Alice* с оценкой 5 и *Dimitriy* с оценкой 4.
40. Создайте динамический массив из 10 чисел (функция *malloc* для всего массива), заполните его числами от 0 до 9 и удалите массив.
41. Добавьте в пустой динамический массив 10 чисел (функция *realloc* для каждого элемента) числа от 0 до 9 и удалите массив.
42. Создайте динамическую матрицу размера 3×4 и удалите её
43. Что будет напечатано:

```
int a[5] = {10, 20, 30, 40, 50};
int *p = a + 2;
printf("%d", *p);
(*p)++;
printf("%d", *p);
*p++;
printf("%d", *p);
```

44. Опишите пример строки программы, в которой встречаются блок символов подряд:

- (a) o*==*o
- (b) a*==**a
- (c) *x*==*x*

Поясните, что есть что.

45. Что произойдет при вызовах функции *free* и почему:

```
int a[2];
int *b = NULL;
int *c = malloc(sizeof(int));
free(a);
free(b);
free(c);
```

46. Описать содержимое переменных `int argc` и `char **argv`, если программа запущена командой:

```
./prog I'll be bug!}
```

47. Написать программу, в которую через 1-й аргумент командной строки передается целое число. Найти квадрат этого числа.
48. На стандартный ввод подается имя файла. Распечатать только строчные буквы из файла.
49. На стандартный ввод подается целое число n . Написать программу, которая печатает в файл "ascii.txt" первые n символов ascii таблицы.
50. Написать программу, которая печатает свое имя.

17.2 Практическая часть

1. Дано 3 целых числа a , b и c . Если треугольник с данными сторонами существует, то выведите его площадь с 1 знаком после запятой. Иначе выведите сообщение «do not exist».

Ввод	-1 2 3	3 5 4	5 6 7
Вывод	do not exist	6.0	14.7

2. Дано целое положительное число n от 1 до 10000. Далее n целых чисел от -1000 до 1000 . Найти все позиции максимального числа.

Ввод	8 1 5 2 5 3 -1 5 -2
Вывод	2 7

3. Дано целое положительное число. Вывести сумму делителей числа. Описать соответствующую функцию.

Ввод	6	10	13
Вывод	12	18	14

4. Дана последовательность слов (слово — произвольные символы отличные от пробелов, которые разделены ровно одним пробелом) с точкой в конце. Посчитать длину каждого слова.

Ввод	He said Hakuna Matata 3 times.
Вывод	2 4 6 6 1 5

5. Дана строка. Вывести все символы в порядке ascii-таблицы, которые встречаются в строке хотя бы один раз, используя массив для подсчета.

Ввод	HappyNewYear2019
Вывод	0129HNYaerpy

6. Даны две отсечки времени в формате **hh:mm:ss** в течении одних суток. Определить сколько секунд между данными отсечками. Описать подходящую структуру и функцию

Ввод	09:00:00 10:20:00	09:00:00 08:59:59	00:00:00 23:59:59
Вывод	4800	1	86399

7. Даны n . Далее n целых чисел. Сдвинуть массив циклически вправо. Использовать динамический массив.

Ввод	5 1 2 3 4 5
Вывод	5 1 2 3 4

8. Дано целое положительное число N от 2 до 100. Далее $2N$ вещественных чисел – координаты двух n -мерных векторов v_1 и v_2 . Найти косинус угла между v_1 и v_2 . Описать функцию, которая находит скалярное произведение двух N -мерных векторов. Идея: $(\vec{v}_1, \vec{v}_2) = \sqrt{(\vec{v}_1, \vec{v}_1)}\sqrt{(\vec{v}_2, \vec{v}_2)}\cos\alpha$.

Ввод	2 3.0 4.0 4.0 3.0	3 3.0 4.0 0.0 -3.0 4.0 12.0
Вывод	0.96	0.1077

9. Даны целые положительные числа n от 1 до 1000000. Сохранить в динамический массив и распечатать все простые делители числа n с учетом кратности. Для каждого найденного делителя необходимо расширить массив на 1 элемент и добавить его в конец.

Ввод	12
Вывод	2 2 3

10. Дано целое положительное число n от 1 до 20. Вывести матрицу размера $n \times n$, где $a_{ij} = \min(i, j, n + 1 - i, n + 1 - j)$.

Ввод	5
Вывод	1 1 1 1 1 1 2 2 2 1 1 2 3 2 1 1 2 2 2 1 1 1 1 1 1

11. Даны несколько положительных дробей (ввод заканчивается 0), разделенных пробелом. Вычислить сумму дробей (результат сократить на наибольший общий делитель). Числа хранить в динамическом массиве.

Ввод	3 1/2 1/3 1/6 0/0	4 1/2 1/3 2/5 1/6 0/0
Вывод	1/1	7/5

12. Дана последовательность символов (буквы английского алфавита и круглые скобки) с точкой в конце. Необходимо проверить, является ли эта последовательность сбалансированной по скобкам (вывести ОК) или нет (вывести позицию первой скобки слева, которая не имеет пары). Сохранять позиции скобок в расширяющийся динамический массив.

Ввод	Minoins and (Banana).	(Minoins) and) Banana.
Вывод	OK	14

13. Дано целое n от 1 до 100. Далее n целых чисел x_1, x_2, \dots, x_n от (-1000) до 1000 — координаты n -мерного вектора. Необходимо отнормировать вектор, то есть вывести коллинеарный x вектор длины 1. Ответ вывести с точностью в один знак после запятой.

Ввод	2 -3 4	4 3 3 -3 3
Вывод	-0.6 0.8	0.5 0.5 -0.5 0.5

14. Дана строка из английских букв и пробелов. Если строка не превышает 10 символов, то вывести её целиком. Иначе вывести первые и последние 4 буквы, вставив между ними две точки.

Ввод	Everyone they want	Zootopia
Вывод	Ever..want	Zootopia

15. Даны целые положительные числа $a < b$, оба от 1 до 1000000. Описать функцию, которая находит $\sigma(n)$ — сумму всех собственных делителей числа (например, $\sigma(12) = 1 + 2 + 3 + 4 + 6 = 16$). С помощью данной функции вывести все числа из диапазона $[a; b]$, для которых $\sigma(n) > n$. Если таких чисел нет вывести EMPTU.

Ввод	10 20	31 35
Вывод	12 18 20	EMPTY

16. Дано целое положительное n от 1 до 1000000. Далее $(n - 1)$ целых различных чисел от 1 до n . Выяснить, какого числа нет.

Ввод	5 1 5 3 2	2 2
Вывод	4	1

17. Даны N от 1 до 1000. Далее N различных целых чисел от -1000 до 1000. Найти медиану массива. Если N — нечетное, то это центральный элемент отсортированного массива. В противном случае — это полусумма двух центральных элементов отсортированного массива.

Ввод	5 11 14 12 13 20	6 11 14 12 13 15 20
Вывод	13	13.5

18. Через аргумент командной строки передается имя файла. В первой строке содержатся даты 3х занятий в формате дд.мм (день и месяц). Далее содержится информация о студентах: количество студентов и информация по ним (фамилия и 3 оценки от 0 до 10). День считается удачным, если в этот день студент выполнил хотя бы 7 заданий. Вывести у каждого студента удачные дни. Даты хранить в статическом массиве структур дат, а студентов в динамическом массиве.

input.txt	16.11 18.11 23.11 3 Mihno 7 8 7 Korobov 6 9 6 Pikulina 10 9 5
Запуск	./prog input.txt
Вывод	Mihno 16.11 18.11 23.11 Korobov 18.11 Pikulina 16.11 18.11

19. Программа запускается с 2 аргументами (имена входного и выходного файлов). Известно, что текст первого файла состоит только из букв и разделителей (пробелы и переносы строк). При этом никакие 2 разделителя не идут подряд. Необходимо текст первого файла скопировать во второй, при этом заменяя все буквы в четных словах на маленькие, а в нечетных словах на большие.

in.txt	Hello Minoins Do You Want a BANANA
Запуск	./prog in.txt out.txt
out.txt	HELLO minions DO you WANT a BANANA

20. Дана строка из букв, цифр, арифметических знаков и скобок трех видов '(-)', '[-]' и '{-}'. Проверить сбалансировать скобок. Если ошибок нет, вывести ОК, в противном случае — позиции скобок, которые не являются парными.

Подсказка: каждую открывающуюся скобку добавлять в стек (дополнительный расширяющийся массив) вместе с ее позицией.

Ввод	(1+[a*(3+c)+5])	(1+{a*(3+c)+5}))	1+a*[3+c]+5))
Вывод	ОК	14 symbol '))' 4 symbol '{'	12 symbol '))' no pair symbol

18 Основная литература

1. Подбельский В., Фомин С. Курс программирования на языке Си: учебник. ДМК Пресс, 2012. 384 с.
2. Валединский В. Д., Корнев А. А. Методы программирования в примерах и задачах. М. Изд-во ЦПИ при механико-математическом ф-тк МГУ, 2000. 152 с.
3. Столяров А. В. Программирование: введение в профессию. I: Азы программирования. М.: МАКС Пресс, 2016. 464 с. Доступна электронная версия. Режим доступа:
<http://stolyarov.info/books>.
4. Столяров А. В. Программирование: введение в профессию. II: Низкоуровневое программирование. М.: МАКС Пресс, 2016. 496 с. Доступна электронная версия. Режим доступа:
<http://stolyarov.info/books>.
5. Керниган Б., Ритчи Д. Язык программирования С. Москва: Вильямс, 2015. 304 с.

19 Дополнительная литература

1. Купер М. Искусство программирования на языке сценариев командной оболочки. [Электронный ресурс]. Режим доступа:
https://www.opennet.ru/docs/RUS/bash_scripting_guide/
2. Шапошникова С. В. Основы языка С. Курс по программированию. [Электронный ресурс]. Режим доступа:
<https://younglinux.info/c>
3. Чернов А. Стиль форматирования программ. [Электронный ресурс]. Режим доступа:
<https://ejudge.ru/study/3sem/style.shtml>
4. CS106B Style Guide. Stanford University's Computer Science Department. [Электронный ресурс]. Режим доступа:
<http://stanford.edu/class/archive/cs/cs106b/cs106b.1158/>
