Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина
Институт естественных наук и математики

Д. Р. КУВШИНОВ
С. И. ОСИПОВ

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык C++

Практикум

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИМЕНИ ПЕРВОГО ПРЕЗИДЕНТА РОССИИ Б. Н. ЕЛЬЦИНА

Д. Р. Кувшинов, С. И. Осипов

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык C++

Практикум

Рекомендовано методическим советом
Уральского федерального университета в качестве практикума
для студентов, обучающихся по направлениям подготовки
01.03.03 «Механика и математическое моделирование»,
01.03.04 «Прикладная математика»

Екатеринбург
Издательство Уральского университета
2021

УДК 004.43(076.5)
ББК 32.973.26-018.1я73
К88

Под общей редакцией Д. Р. Кувшинова

Рецензенты:
В. С. Тарасян, кандидат физико-математических наук, доцент,
заведующий кафедрой «Мехатроника»
(Уральский государственный университет путей сообщения);
С. В. Булычева, кандидат физико-математических наук, доцент
(Магнитогорский государственный технический университет)

Кувшинов, Д. Р.
К88 Основы программирования : язык C++ : практикум / Д. Р. Кувшинов, С. И. Осипов ; под общ. ред. Д. Р. Кувшинова ; Министерство науки и высшего образования Российской Федерации, Уральский федеральный университет. — Екатеринбург : Изд-во Урал. ун-та, 2021. — 170 с. : ил. — 30 экз. — ISBN 978-5-7996-3256-4. — Текст : непосредственный.

ISBN 978-5-7996-3256-4

Практикум включает в себя набор лабораторных и самостоятельных работ и предназначен для наработки и закрепления навыков элементарной алгоритмизации и написания программ на языке C++. Предлагаются задачи на организацию вычислений, логические операции, работу с потоками ввода-вывода, одномерными и двумерными массивами, си-строками, конечными автоматами, а также задачи на управление памятью.

Для студентов бакалавриата, обучающихся по направлениям «Механика и математическое моделирование» и «Прикладная математика».

УДК 004.43(076.5)
ББК 32.973.26-018.1я73

ISBN 978-5-7996-3256-4

Оглавление

Предисловие

Материал книги рассчитан на длину семестра в 18 недель.

Предполагается использование стобалльной системы. Каждая из работ помечена либо как лабораторная, либо как самостоятельная. Лабораторная работа выполняется в учебном классе и не разделена на варианты. Самостоятельная работа состоит из набора вариантов и может выполняться в классе и назначаться в качестве домашнего задания. Рекомендуется оценивать работу студента на лабораторной работе (кроме первой) в 0–3 балла (отсутствовал, присутствовал, участвовал, выполнил). Итого, лабораторные работы суммарно дают 0–24 балла.

Оставшиеся 0–76 баллов распределяются между самостоятельными работами. Предлагаются следующие максимальные оценки в баллах за самостоятельные работы:

Работа	Оценка	Работа	Оценка
2	4	11	8
4	3	13	9
5	4	14	9
6	5	16	10
7	6	18	9
9	9		

1

Значения и переменные

Цели лабораторной работы

- Освоить сборку консольного приложения в среде разработки на языке программирования C++.
- Изучить базовые средства текстового ввода-вывода, предоставляемые стандартной библиотекой C++.
- Изучить работу операторов C++, выполняющих арифметические операции над целыми числами.
- Изучение цикла чтения.

Оператор	Операция
+	сложение
−	вычитание
*	умножение
/	деление
%	остаток от деления

Арифметические операторы C++

Задание

1. Предполагается, что доступна некая среда разработки (IDE) с установленным компилятором C++. Запустите ее.

2. Создайте новый (пустой) проект консольного приложения.

3. Если создан пустой проект, то он не содержит файлов исходного кода. Добавьте в него новый пустой файл исходного кода на языке C++ (cpp-файл). Далее в данной работе все действия производятся в этом файле.

4. Попробуйте ввести в новый файл текст

   ```
   2+2
   ```

   и запустить сборку проекта. Видны ли ошибки компиляции?

5. Введите следующую программу:

   Пример 1.1. Простейшая программа

   ```
   int main() {}
   ```

   Запустить сборку и исполнение кода. Программа должна собраться без ошибок.

6. Выполните следующую модификацию:

   Пример 1.2. 2+2?

   ```
   int main() { 2+2; }
   ```

   Запустите новую программу. Выводы?

7. Выполните следующую модификацию:

   Пример 1.3. Текстовый вывод

   ```
   #include <iostream>
   int main() {
     std::cout << (2+2);
   }
   ```

Запустите программу.

Здесь **iostream** — заголовочный файл, часть стандартной библиотеки C++. Заголовочные файлы подключаются с помощью *директивы препроцессора* #**include**, после которой идет имя подключаемого файла, указываемое или в «угловых скобках» (знаки «меньше» и «больше»), или в кавычках (знак ").

Стандартные заголовочные файлы C++ не имеют расширения (iostream, но **не** iostream.h и **не** iostream.hpp).

Кавычки принято использовать, когда подключается файл из проекта или находящийся в одной директории с подключающим его файлом исходного кода.

Угловые скобки принято использовать в том случае, когда подключается файл из стандартной библиотеки или иной библиотеки, путь к которой известен компилятору.

8. Выполните следующую модификацию программы.

Пример 1.4. Вывод нескольких значений

```
#include <iostream>
int main() {
  std::cout << "6 by 4 gives " << (6*4) << "\n";
}
```

Что означает знак *? Что будет выведено, если заменить * на / (деление)? Попробуйте сначала дать ответ на этот вопрос, а затем проверьте свой ответ экспериментом.

9. Запись \n задает символ перевода строки. Если при запуске программы вывод осуществляется в окно консоли, которое сразу закрывается, то можно добавить в программу (перед закрывающей }) одну строчку:

```
  std::cin.ignore();
```

10. Выведите значения выражений (4/6), (6/4), (4.0/6), (6.0/4). Проанализируйте полученные результаты.

Когда **оба** операнда являются **целыми числами**, выполняется **целочисленное деление**. В вопросе определения точного смысла операции в C++ основополагающее значение имеет *тип операндов*. При написании программ очень важно об этом помнить.

11. Выведите значения выражений (4%6), (6%4), ((−6)%4), (6%(−4)) и ((−6)%(−4)). Здесь унарный минус — оператор, выполняющий смену знака. Проанализируйте полученные результаты.

12. Теперь попробуйте ввести и запустить следующую программу.

Пример 1.5. Приветствие

```
#include <iostream>
int main() {
  std::cout << "Hello!\n";
  std::cin.ignore();
}
```

Программа должна вывести строку с приветствием, после которой с новой строки ожидать ввод.

13. Часто бывает обременительно в каждом обращении к имени из стандартной библиотеки писать префикс std:: («выбрать имя из *пространства имен* std»). Чтобы этого не делать, можно указать директиву

```
using namespace std;
```

после которой компилятор будет искать имена, не найденные в *глобальном пространстве имен*, в пространстве имен std.

Пример 1.6. Уберем префикс std::

```
#include <iostream>
int main() {
  using namespace std;
```

```
    cout << "Hello!\n";
    cin.ignore();
}
```

14. Для того чтобы получить какие-либо данные от пользователя (из потока ввода cin), требуется «застолбить» в памяти место для их размещения, что в C++ осуществляется путем определения переменной подходящего типа. Предположим, мы считываем два целых числа n и m. Определим две переменные:

    ```
    int n, m;
    ```

    Данная строчка требует от компилятора зарезервировать в памяти место под два значения типа **int** и назначить им имена n и m. Теперь при обращении по имени будет осуществляться доступ к соответствующей области памяти. Основополагающим ее свойством является возможность изменять свое состояние — переменные могут изменять свои значения.

15. Добавьте в программу определение двух целочисленных переменных n и m, как указано в предыдущем пункте. Выведите их значения на экран. Что происходит? Что должно быть выведено?

16. Переменным можно задать начальное значение (*инициализировать* их). Например, так:

    ```
    int n = 1, m = 2;
    ```

    Задайте начальные значения переменным в программе. Убедитесь, что они теперь выводятся на экран.

17. Наконец, благодаря тому, что у нас есть теперь зарезервированное в памяти место под два числа, мы можем запросить их у пользователя и считать из потока cin. Сделать это можно с помощью оператора >> («направление ввода обратно направлению вывода»):

```
cin >> n >> m;  // сначала прочитать n, затем m.
```

Числа вводятся подряд через пробельные символы (например, пробел или перевод строки).

18. Модифицируйте программу таким образом, чтобы она запрашивала у пользователя два целых числа и выводила их произведение и частное. Попробуйте вводить разные числа, в частности, 0 и 0.

19. Экспериментировать с такой программой неудобно, так как ее постоянно приходится запускать заново, чтобы ввести очередную пару чисел. Совсем несложно сделать так, чтобы программа повторяла весь этот набор действий, пока мы ее насильно не прервем. Для этого надо заключить повторяемый код внутрь *вечного цикла*:

```
for (;;) {
  // Повторяющийся код ставить здесь.
}
```

Поместите код своей программы (тот, что заключен внутри { }) в вечный цикл и запустите программу. Что случится, если допустить ошибку ввода (например, ввести букву)?

20. Вечный цикл неудобен тем, что для выхода из него приходится насильно прерывать исполнение программы. При работе в консоли пользователь обычно может это сделать нажатием комбинации Ctrl+C, но для автоматической работы это не годится (например, считывания файла). Мы можем модифицировать цикл так, чтобы он завершался в случае ошибки ввода. Для этого в C++ принято использовать следующую идиоматическую конструкцию на основе **for** (*цикл чтения*):

```
for (определение-переменных; чтение-переменных;) {
  // Работа с очередными прочитанными значениями.
}
```

В нашем случае это превращается в следующий код:

```
for (int n, m; cin >> n >> m;) {
  ...
}
```

Замените в своей программе вечный цикл на цикл чтения. Добейтесь ее работоспособности.

Завершение ввода осуществляется через ошибку ввода или ввод признака конца файла (в отдельной строке Ctrl+Z, Enter в ОС Windows, Ctrl+D, Enter в ОС семейства Unix).

21. Поменяйте тип **int** на тип **float**.

    Проверьте, что теперь получится при вводе пар 1 0 и 0 0.

22. Поменяйте тип **float** на тип **char**. Попробуйте запустить программу. Как можно проинтерпретировать полученные результаты?

23. Пусть есть следующий цикл:

```
for (char c; cin >> c;)
  cout << c << "␣code␣is␣" << int(c) << '\n';
```

    Что он выполняет? Какая кодировка используется вашим консольным приложением?

2

Функции

Цели самостоятельной работы

- Освоить базовый синтаксис определения и применения функций в языке C++.
- Закрепить навыки использования элементарных средств ввода-вывода.

Критерии полноты

1. Программа запрашивает у пользователя значение и выполняет все возможные переводы единиц (всего шесть вариантов для трех единиц).
2. Определены функции перевода единиц, позволяющие переводить значение, заданное в любой из указанных в задании единиц, в значение в любой другой из указанных в задании единиц.
3. Использована композиция функций: для получения шести направлений перевода единиц достаточно определить три или четыре функции.

Замечание. В записи десятичных дробей в данном разделе и вообще в книге в качестве разделителя целой и дробной ча-

сти используется точка, а не запятая. Данное решение следует синтаксису записи чисел в языке C++ (и большинстве других языков программирования).

2.1. Примеры

Задание. Расстояние в дюймах ↔ футах ↔ сантиметрах. Считать, что 1 дюйм равен 2.54 см, один фут равен 30.48 см.

Итак, воспользовавшись данными, указанными в задании, напишем функции, принимающие значение расстояния в сантиметрах и возвращающие его в дюймах (функция cm2in[1]) или футах (функция cm2ft):

```
float cm2in(float cm) {
  return cm / 2.54f;
}
float cm2ft(float cm) {
  return cm / 30.48f;
}
```

Обратное преобразование получается через умножение:

```
float in2cm(float in) {
  return in * 2.54f;
}
float ft2cm(float ft) {
  return ft * 30.48f;
}
```

Оставшиеся два преобразования дюймы ↔ футы можно получить как композицию функций. Например, можно перевести футы в дюймы как cm2in(ft2cm(ft)). Обратите внимание на порядок записи функций: в композиции операция, применяемая первой, идет после операции, применяемой второй. Запишем функцию main, соответствующую заданию:

[1] Цифра 2 в названии функций на английском произносится так же, как предлог *to*, т. е. букв. cm (см) «в» inches (дюймы).

```
#include <iostream>

float cm2in(float cm);
float cm2ft(float cm);
float in2cm(float in);
float ft2cm(float ft);

int main() {
  using namespace std;
  cout << "Enter length: ";
  float len = 0;
  cin >> len;
  cout << "\nin to cm = " << in2cm(len);
  cout << "\nft to cm = " << ft2cm(len);
  cout << "\ncm to in = " << cm2in(len);
  cout << "\ncm to ft = " << cm2ft(len);
  cout << "\nin to ft = " << cm2ft(in2cm(len));
  cout << "\nft to in = " << cm2in(ft2cm(len));
  return 0;
}
```

Данное решение удовлетворяет всем критериям полноты работы.

2.2. Варианты заданий

Вариант 1. Температура в шкалах Цельсия $\leftrightarrow$ Кельвина $\leftrightarrow$ Фаренгейта.

Обозначим температуру в градусах Цельсия через t_C, температуру в градусах Кельвина через t_K, температуру в градусах Фаренгейта через t_F. Тогда:

$$t_F = \frac{9}{5} t_C + 32, \quad t_K = t_C + 273.15.$$

Вариант 2. Расстояние в секундах экватора Земли $\leftrightarrow$ милях $\leftrightarrow$ километрах. Считать, что длина экватора Земли равна 40 075 км, 1 миля равна 1.609 км. Экватор как окружность равен 360°, $1^\circ = 60'$ (минут), $1' = 60''$ (секунд).

Вариант 3. Масса в (английских) фунтах ↔ унциях ↔ килограммах. Считать, что 1 английский фунт равен 16 унциям и 453.59237 г.

Вариант 4. Расстояние в парсеках ↔ световых годах ↔ астрономических единицах. Считать, что 1 парсек равен 3.2616 световых лет и 206 264.8 астрономических единиц.

Вариант 5. Энергия в джоулях ↔ калориях ↔ электронвольтах. Считать, что 1 калория равна 4.1868 Дж, 1 эВ равен $1.602176621 \cdot 10^{-19}$ Дж.

Вариант 6. Расстояние в типографских пунктах ↔ линиях ↔ миллиметрах. Считать, что 1 пункт равен 0.376 мм, 1 линия равна 2.54 мм.

Вариант 7. Объем в литрах ↔ галлонах ↔ баррелях (американских). Считать, что 1 баррель равен 42 галлона, 1 галлон равен 3.785 л.

Вариант 8. Площадь в гектарах ↔ квадратных милях ↔ квадратных километрах. Считать, что 1 км^2 равен 100 га, 1 кв. миля равна 2.588881 км^2.

Вариант 9. Скорость в узлах ↔ км/ч ↔ м/с. Считать, что 1 м/с равен 3.6 км/ч, 1 узел (морских миль в час) равен 1.852 км/ч.

Вариант 10. Плоский угол в градусах ↔ градах ↔ радианах. Полная окружность равна 360°, 400 град, 2π рад.

Вариант 11. Объем при одинаковой массе в нормальных условиях: дистиллированная вода ↔ этанол (95.6%) ↔ концентрированная серная кислота (98%). Считать, что плотность:

- дистиллированной воды 1 г/см^3;
- этанола (95.6%) 0.7693 г/см^3;
- серной кислоты (98%) 1.8365 г/см^3.

Вариант 12. Время в секундах $\leftrightarrow$ звездных сутках $\leftrightarrow$ звездных месяцах. Считать, что звездные сутки равны 86 164 с, звездный месяц равен 2 360 606 с.

Вариант 13. Телесный угол в квадратных секундах $\leftrightarrow$ квадратных градусах $\leftrightarrow$ стерадианах. Считать, что 1 кв. градус равен $3\,600^2$ кв. секунд, 1 стерадиан равен 3 282.80635 кв. градусов.

Вариант 14. Масса при одинаковом объеме в нормальных условиях: свинец $\leftrightarrow$ вольфрам $\leftrightarrow$ уран-238 (обедненный). Считать, что плотность:

- свинца 11.3415 г/см^3;
- вольфрама 19.25 г/см^3;
- урана-238 19.1 г/см^3.

Вариант 15. Скорость в м/с $\leftrightarrow$ км/ч $\leftrightarrow$ фут/с: 1 м/с равен 3.6 км/ч, 1 фут/с равен 0.3048 м/с.

Вариант 16. Ускорение в м/с^2 $\leftrightarrow$ фут/с^2 $\leftrightarrow$ единицах g. Считать, что 1 фут/с^2 равен 0.3048 м/с^2, 1 g равно 9.80665 м/с^2.

3

Командная строка и элементарная алгоритмизация

Цели лабораторной работы

- Изучение базовых конструций языка интерпретатора командной строки.
- Реализация простого линейного алгоритма, обрабатывающего текстовый ввод (файл).

Задание

1. Запустите среду разработки и создайте новый пустой проект консольного приложения. Добавьте в него новый пустой файл исходного кода на языке C++.
2. Напишите работающую программу, выводящую в поток вывода произвольный текст. Соберите консольное приложение. Предположим, что исполняемый файл приложения имеет название lab.exe. Найдите его месторасположение и запустите там терминал командной строки.

3. Введите в терминале команду lab и нажмите Enter. Программа должна вывести свой текст в терминал. В общем-то, скорее всего, набирать lab целиком необязательно: попробуйте набрать l и затем нажать Tab. Терминалы обычно позволяют таким образом перебирать файлы, имена которых начинаются на набранную часть, что очень удобно, если имя длинное.

4. Наберите в терминале lab >o.txt и нажмте Enter. На этот раз в терминале мы текста программы не увидим. Но рядом с lab.exe должен появиться файл o.txt, в который будет записан выведенный программой текст. Убедитесь в этом.

5. Символы < и > («стрелки») в терминале командной строки позволяют *перенаправлять* [*redirect*] стандартные потоки ввода и вывода на файл или устройство, имя которого указывается после стрелки. Запись таким способом в уже существующий файл стирает его старое содержимое без предупреждения, поэтому требуется некоторая осторожность. Убедитесь в этом, поменяв в программе выводимый текст и заново выполнив команду lab >o.txt. Повторить команду можно, не набирая ее вновь: обычно терминалы позволяют прокручивать список уже вводившихся команд с помощью клавиш ↑ и ↓. Скорее всего, будет достаточно нажать ↑ и Enter.

6. Если написать lab >>o.txt, то вывод программы будет дописан к старому содержимому файла. Убедитесь в этом.

7. Определите, что делает следующий код:

```
if (char prev; cin.get(prev)) {
  cout.put(prev);
  for (char ch; cin.get(ch);)
    if (ch != prev)
      cout.put(prev = ch);
}
```

   Проверьте на каком-нибудь текстовом файле.

8. Язык C++ не содержит встроенного строкового типа (запись вроде "some␣text" задает массив символов), однако стандартная библиотека C++ предлагает определение строкового типа std::string, которого достаточно для большинства применений. Оно размещено в заголовочном файле **string**.

   Сравните два следующих примера и определите, в чем заключается разница в их поведении.

Пример 3.1. Чтение строк оператором >>

```
#include <string>
#include <iostream>
using namespace std;
int main() {
  string name; // вначале пустая строка.
  cout << "What␣is␣your␣name?␣";
  cin >> name;
  cout << "Greetings,␣" << name << "\n";
  return 0;
}
```

Пример 3.2. Чтение строк функцией getline

```
#include <string>
#include <iostream>
using namespace std;
int main() {
  string name; // вначале пустая строка.
  cout << "What␣is␣your␣name?␣";
  getline(cin, name);
  cout << "Greetings,␣" << name << "\n";
  return 0;
}
```

9. Функция getline возвращает объект потока ввода, поэтому ее можно использовать в условии цикла чтения.

   Напишите программу, читающую с помощью getline строки из потока ввода и выводящую их в поток вывода без идущих подряд повторений. Проверьте работоспособность

данной программы, используя перенаправление ввода в терминале командной строки (взять файл на вход и получить файл на выходе). Переименуйте полученный исполняемый файл в unique.exe.

10. Запись команд через символ | (*пайп* [*pipe*]) перенаправляет стандартный поток вывода левой команды на стандартный поток ввода правой команды.

    Например, команда sort сортирует переданный ей файл как последовательность строк в словарном порядке и (по умолчанию) выводит результат в стандартный поток вывода. Если мы хотим просматривать результат «экранами», можно передать его команде more:

    sort input.txt | more

    Используйте комбинацию sort и созданного в предыдущем пункте unique, чтобы удалить все повторяющиеся строки из файла. Результат можно вывести в терминал через more или в другой файл с помощью перенаправления потока.

11. Запись команд через символ & приводит к последовательному их исполнению. Интереснее связки, которые позволяют выполнять команды при успехе или неуспехе предыдущей команды. Успех или неуспех определяется числом, которое мы возвращаем из функции main. Если это 0, то вызов считается успешным. В противных случаях — неуспешным.

    Здесь *команда-2* будет выполнена только в случае успеха *команды-1*:

    команда-1 && *команда-2*

    А здесь наоборот, *команда-2* будет выполнена только в случае неуспеха *команды-1*:

    команда-1 || *команда-2*

    Можно объединить оба этих варианта в одну конструкцию (в случае успеха *команды-1* выполнить *команду-2*, в случае неуспеха *команды-1* или *команды-2* выполнить *команду-3*):

команда-1 && *команда-2* || *команда-3*

Таким образом, например, задействовав команду echo можно генерировать сообщение об успехе или неуспехе конкретной команды:

команда && echo Success || echo Fail

12. Напишите программу, читающую строки и возвращающую код ошибки 1 в случае совпадения двух подряд идущих строк. В случае, если совпадение не будет обнаружено, программа должна возвращать код 0 (успех).

    Напишите вызов, задействующий sort и эту программу, а также echo для того, чтобы сообщить пользователю, содержит ли некоторый текстовый файл совпадающие строки или нет.

13. Напишите аналог команды more.

4

Аккумуляция последовательности значений

Цель самостоятельной работы

- Закрепление навыков организации циклов ввода и условного выполнения операций.

Критерии полноты

1. Функция main содержит корректно организованный цикл ввода последовательности чисел. Использовать числовой тип **double**, если в задании не указано иначе.

2. Внутри цикла ввода реализована проверка указанного в задании предиката, в случае истинности которого выполняется заданное действие.

3. Результат выводится в стандартный поток вывода. Если не указан результат в случае отсутствия удовлетворяющих предикату введенных значений, то выводить нуль.

Функции min и max доступны при подключении заголовочного файла **algorithm**.

4.1. Примеры

Задание. Сумма положительных чисел.

Считываем последовательность чисел x. В случае $x > 0$ добавляем x к сумме, начальное значение которой — нуль, который будет выведен, если не удастся считать ни одного числа:

```
#include <cmath>
using namespace std;

int main() {
  double sum = 0;
  // Цикл ввода.
  for (double x; cin >> x;)
    if (x > 0)  // предикат
      sum += x; // аккумуляция
  // Вывод результата.
  cout << sum;
  return 0;
}
```

4.2. Варианты заданий

Вариант 1. Минимум среди чисел, больших единицы. Единица, если таковых не будет введено.

Вариант 2. Максимум среди отрицательных чисел. Нуль, если таковых не будет введено.

Вариант 3. Сумма модулей чисел, не превосходящих по модулю единицу.

Вариант 4. Модуль произведения чисел, не меньших единицы по модулю. Единица, если таковых не будет введено.

Вариант 5. Сумма $\frac{1}{x}$ для $|x| > 10^{-6}$.

Вариант 6. Корень квадратный из суммы квадратов чисел, не превосходящих по модулю единицы.

Вариант 7. Произведение x таких, что $1 < x < 2$. Единица, если таковых не будет введено.

Вариант 8. Максимум модулей четных чисел. Нуль, если таковых не будет введено.

Вариант 9. Сумма синусов чисел, не превосходящих по модулю $\frac{\pi}{2}$.

Вариант 10. Сумма натуральных логарифмов чисел, больших единицы.

Вариант 11. Сумма целых частей $\frac{x}{10}$ для x, делящихся нацело на 10 (читать целые числа).

Вариант 12. Максимум квадратов чисел, косинус которых больше нуля.

Вариант 13. Отношение суммы положительных чисел к максимальному из них. Нуль, если таковых не будет введено.

Вариант 14. Минимум из введенных чисел, для которых определен arcsin.

Вариант 15. Минимум среди чисел, в разряде единиц которых записана 9 (читать целые числа).

Вариант 16. Сумма квадратных корней из модулей чисел, тангенс которых больше единицы.

5

Табуляция заданной функции

Цели самостоятельной работы

- Изучение функций, объявленных в стандартном заголовочном файле cmath.

- Организация элементарных циклов со счетчиком.

Критерии полноты

1. Заданная функция реализована в виде отдельной функции.

2. Значения функции в заданных точках выводятся с помощью цикла со счетчиком.

3. Значения повторяющихся подвыражений должны вычисляться однократно.

5.1. Примеры

Задание. Реализовать функцию

$$f(x) = 2^{\frac{1}{2} - \frac{X}{\pi}} |\sin X|,$$

где $X = x^x$. Вывести значения $f(x)$ в точках $x = \frac{n}{4}$, для $n = 0, 1, \ldots, 16$.

Решение данной задачи можно записать следующим образом:

```
#include <iostream>
#include <cmath>
using namespace std;
// Определение константы "пи".
double const PI = 3.14159265358979324;
// Требуемая функция.
double f(double x) {
  double const X = pow(x, x);
  return exp2(0.5 - X / PI) * fabs(sin(X));
}

int main() {
  cout.precision(12); // выводить 12 цифр
  for (int n = 0; n < 17; ++n) {
    double const x = 0.25 * n;
    cout << n << '\t' << x << '\t' << f(x) << '\n';
  }
  return 0;
}
```

Поскольку в стандарте нет предопределенной константы π, ее приходится задавать в решении.

5.2. Варианты заданий

Вариант 1. Вывести значения функции

$$f(x) = Y - 3^x + e^{-Y}$$

в точках $x = \frac{n}{4} - 1$, где $n = 0, 1, \ldots, 20$, $Y = x^3$.

Вариант 2. Вывести значения функции

$$f(x) = Y \operatorname{arctg} \ln |Y|$$

в точках $x = \frac{n}{4} - 2$, где $n = 0, 1, \ldots, 16$, $Y = x^2 - 1$.

Вариант 3. Вывести значения функции

$$f(x) = (\ln Y + 2x)e^{-Y}$$

в точках $x = \frac{n}{8}$, где $n = -10, -9, \ldots, 20$, $Y = |x| + 1$.

Вариант 4. Вывести значения функции

$$f(x) = \frac{\sin \cos Y}{Y + x}$$

в точках $x = \frac{n}{4} + 1$, где $n = 0, 1, \ldots, 10$, $Y = x^3$.

Вариант 5. Вывести значения функции

$$f(x) = \operatorname{arctg}\left(x + \frac{\operatorname{tg} Y}{Y}\right)$$

в точках $x = \frac{n}{8} - 2$, где $n = 0, 1, \ldots, 24$, $Y = x^2 + 1$.

Вариант 6. Вывести значения функции

$$f(x) = e^Y - 1000^{-Y}$$

в точках $x = \frac{n}{16} - 1$, где $n = 0, 1, \ldots, 32$, $Y = \lfloor x \rfloor - x$.

Вариант 7. Вывести значения функции

$$f(x) = \sqrt[3]{x(Y-1)(Y-2)}$$

в точках $x = \frac{n}{16} - 1$, где $n = 12, 13, \ldots, 36$, $Y = x \sin x$. Значение кубического корня должно вычисляться верно.

Вариант 8. Вывести значения функции

$$f(x) = e^{Y(1-Y)} - (x+1)^{-2}$$

в точках $x = \frac{n}{8}$, где $n = 0, 1, \ldots, 24$, $Y = \sqrt{x}$.

Вариант 9. Вывести значения функции

$$f(x) = \log_3 (3Y + e^{-Y})$$

в точках $x = \frac{n}{8} - 1$, где $n = 0, 1, \ldots, 32$, $Y = \sqrt{|x - 1|}$.

Вариант 10. Вывести значения функции

$$f(x) = \sin(Y\sqrt{2}) + \cos(Y\sqrt{3})$$

в точках $x = \frac{n}{8}$, где $n = 1, \ldots, 25$, $Y = x \ln x$. Константы (корень из двух, корень из трех) определить отдельно.

Вариант 11. Вывести значения функции

$$f(x) = H + \operatorname{arctg}(H - \sqrt{2})$$

в точках $x = \frac{n}{4} - 1$, где $n = 0, 1, \ldots, 14$, $H = x \ln(|x| + 1)$. Константу «корень из двух» определить отдельно.

Вариант 12. Вывести значения функции

$$f(x) = 2^Y + 2^{-1-Y}$$

в точках $x = \frac{n}{8}$, где $n = 0, 1, \ldots, 24$, $Y = 1 - \frac{1}{2}x^2$.

Вариант 13. Вывести значения функции

$$f(x) = \frac{4^{\sin Y}}{\operatorname{tg} Y}$$

в точках $x = \frac{n}{16}$, где $n = 5, 6, \ldots, 21$, $Y = x(1 + x(1 - x^2))$.

Вариант 14. Вывести значения функции

$$f(x) = \ln(2 + Y) - \frac{Y}{|x|}$$

в точках $x = \frac{n}{8}$, где $n = 0, 1, \ldots, 35$, $Y = \sin x$.

Вариант 15. Вывести значения функции

$$f(x) = \frac{\operatorname{tg}(1+Y)}{\operatorname{tg}(1-Y)}$$

в точках $x = \frac{n}{8}$, где $n = -1, 0, 1, \ldots, 20$, $Y = x^{\frac{2}{3}}$.

Вариант 16. Вывести значения функции

$$f(x) = \sqrt[7]{\cos^7 Y + \sin^7 Y}$$

в точках $x = \frac{n}{8} - 1$, где $n = 0, 1, \ldots, 30$, $Y = \ln(1+x)$.

6

Логические вычисления

Цели самостоятельной работы

- Изучить взаимное соответствие булевских и теоретико-множественных операций на примере вычисления индикаторов множеств на плоскости.

- Закрепить навыки использования булевских (логических) операций.

Критерии полноты

1. Индикатор множества, изображенного на иллюстрации в назначенном варианте, должен быть реализован в виде функции, принимающей координаты точки и возвращающей булевское значение.

2. Программа должна запрашивать у пользователя координаты точки и выводить признак попадания указанной точки в заданную фигуру.

3. Программа должна позволять проверить на попадание произвольное количество точек без перезапуска.

6.1. Примеры

Задание. Построить индикатор фигуры, изображенной на рис. 6.1.

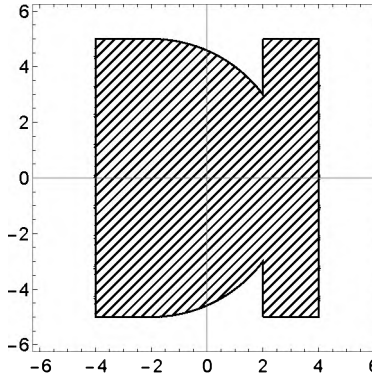

Рис. 6.1. Пример

Фигура на изображении может быть получена пересечением полуплоскости ($x \geqslant -4$) и объединения следующих фигур:

- прямоугольник с углами $(-4, -5)$ и $(-2, 5)$;
- прямоугольник с углами $(2, -5)$ и $(4, 5)$;
- круг радиуса 5 с центром в точке $(-2, 0)$.

Удобно определить отдельные функции-индикаторы для множеств «круг с заданными радиусом r и координатами центра cx, cy» и «прямоугольник со сторонами, параллельными осям координат, заданный абсциссами левой и правой сторон и ординатами нижней и верхней сторон»:

```
// Проверить попадание в круг.
bool in_circle(float x, float y,
    float cx, float cy, float r) {
  float const dx = x - cx, dy = y - cy;
```

```
  return dx*dx + dy*dy <= r*r;
}
// Проверить попадание в прямоугольник.
bool in_rectangle(float x, float y,
  float xmin, float xmax, float ymin, float ymax) {
  return xmin <= x && x <= xmax
      && ymin <= y && y <= ymax;
}
```

Теперь несложно написать индикатор заданной фигуры:

```
// Проверить попадание в заданную фигуру.
bool in_figure(float x, float y) {
  return (in_rectangle(x, y,  2,  4, -5, 5)
       || in_rectangle(x, y, -4, -2, -5, 5)
       || in_circle(x, y, -2, 0, 5))
      && x >= -4;
}
```

Функция main итоговой программы, удовлетворяющей критериям полноты, выглядит следующим образом:

```
int main() {
  using namespace std;
  cout << "Enter a sequence of coordinates x, y: ";
  // Цикл ввода пар значений x, y до первой ошибки.
  for (float x = 0, y = 0; cin >> x >> y;)
    cout << "(x, y) is inside == "
         << in_figure(x, y) << endl;
  return 0;
}
```

Для выполнения данного задания можно взять исходный код примера и заменить в нем функцию-индикатор фигуры-примера на свою функцию-индикатор.

6.2. Варианты заданий

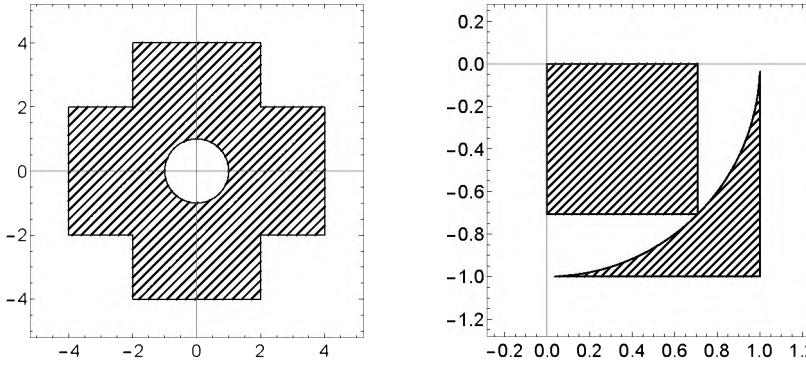

Рис. 6.2. Варианты 1 и 2

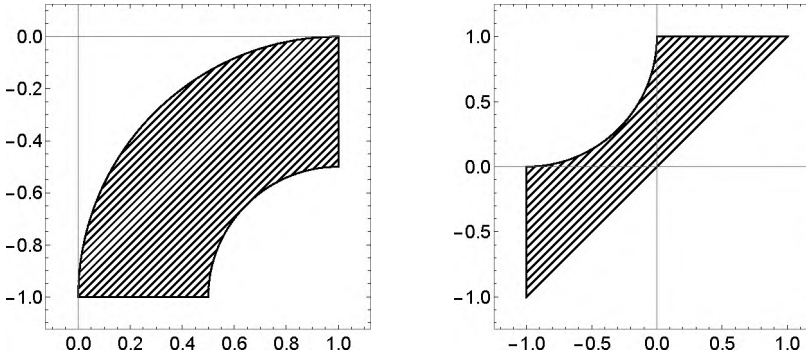

Рис. 6.3. Варианты 3 и 4

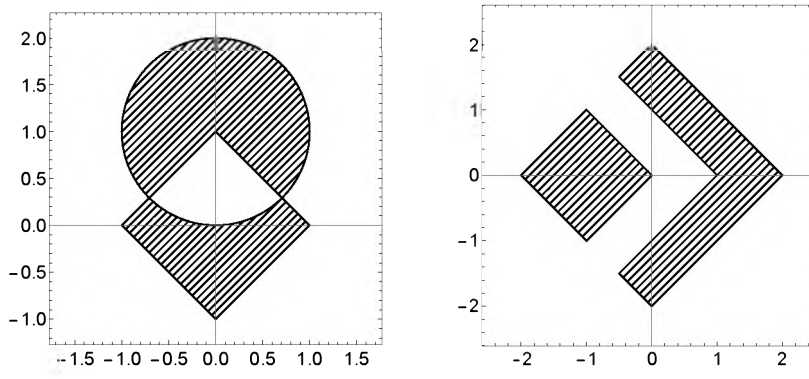

Рис. 6.4. Варианты 5 и 6

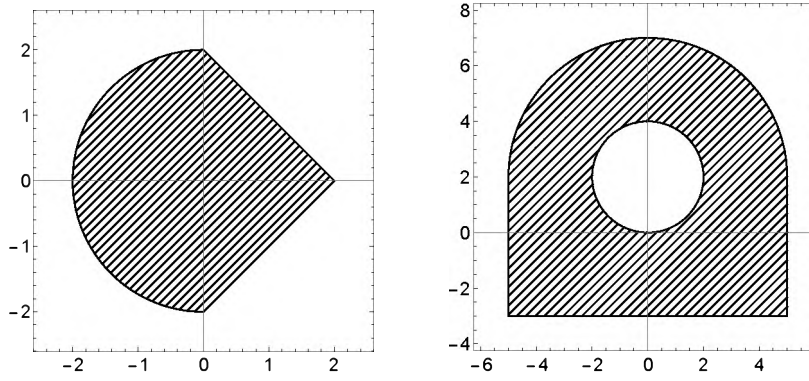

Рис. 6.5. Варианты 7 и 8

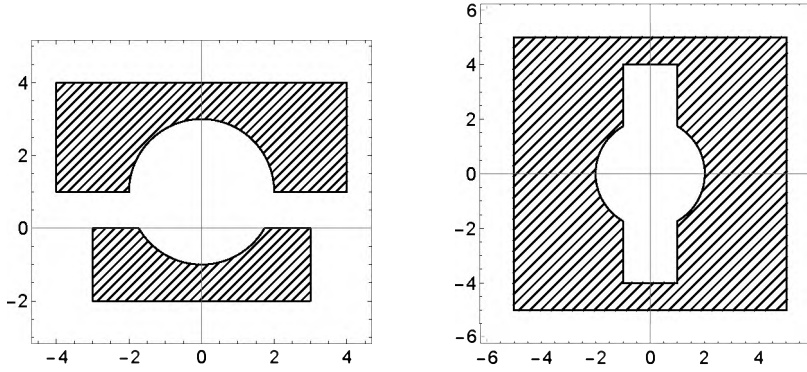

Рис. 6.6. Варианты 9 и 10

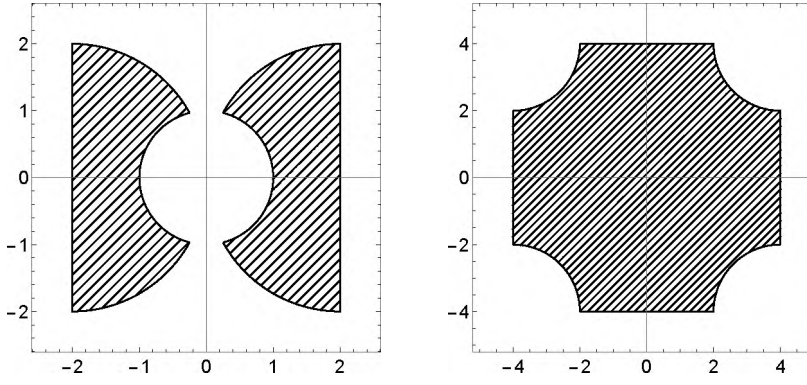

Рис. 6.7. Варианты 11 и 12

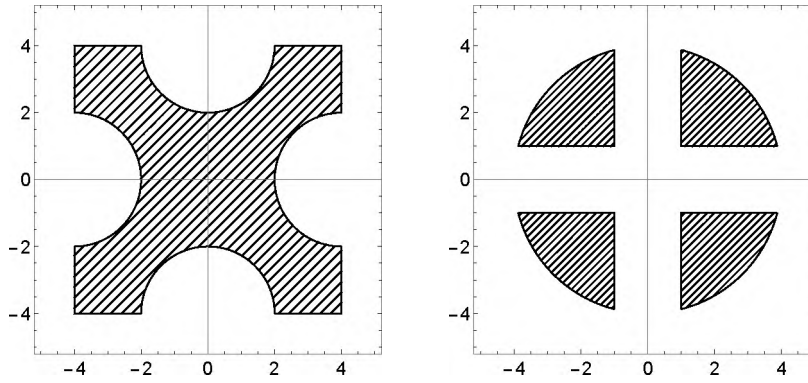

Рис. 6.8. Варианты 13 и 14

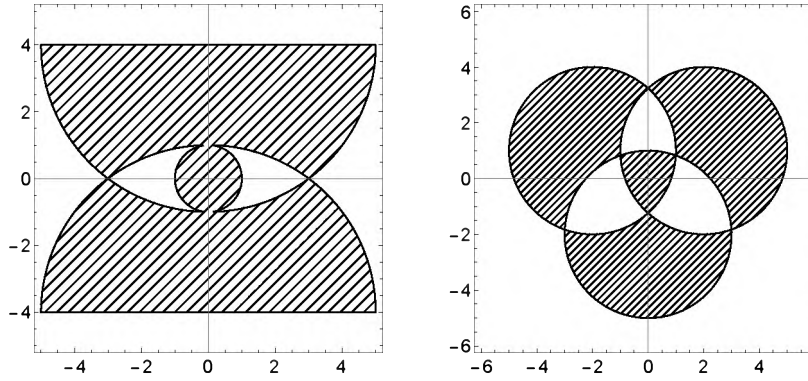

Рис. 6.9. Варианты 15 и 16

7

Простая обработка массива

Цели самостоятельной работы

- Изучение стандартного способа передачи массива в функцию.

- Закрепление навыков элементарной алгоритмизации.

Критерии полноты

1. Реализована отдельная функция, принимающая массив чисел и выполняющая над ними различные действия, в зависимости от заданных условий.

2. Функция main содержит пример массива, состоящего из не менее чем семи элементов.

3. Результат выводится в стандартный поток вывода в функции main.

7.1. Примеры

Задание. Вывести все элементы, превосходящие 1000 по модулю, и их индексы. Вернуть число элементов, не превосходящих 1000 по модулю.

Решение

```
#include <cmath>      // fabs
#include <cstddef>    // size_t
#include <iterator>   // size()
#include <iostream>
using namespace std;

// Функция принимает массив num размера n.
size_t process(double const num[], size_t n) {
  size_t count = 0; /* число элементов,
    не превосходящих 1000 по модулю. */
  for (size_t i = 0; i < n; ++i) {
    if (fabs(num[i]) > 1000.)
      // Вывести i, num[i]:
      cout << i << "\t" << num[i] << '\n';
    else
      // Увеличить счетчик на единицу:
      ++count;
  }
  return count; // вернуть значение счетчика
}

int main()
{
  // Проверка.
  cout.precision(10); // выводить до 10 цифр
  double const check[] {
    -1111, 222, 2, 0, 1e-10, -1e+6,
    222, 1000, 1000.001 };
  cout << process(check, size(check));
  return 0;
}
```

7.2. Варианты заданий

Вариант 1. Вывести максимум четных чисел и вернуть сумму нечетных чисел (целые числа).

Вариант 2. Вывести минимум положительных чисел (или нуль) и вернуть произведение отрицательных чисел.

Вариант 3. Вывести каждое третье число и вернуть сумму прочих чисел.

Вариант 4. Вывести все элементы, превосходящие единицу по модулю, и их индексы. Вернуть арифметическое среднее прочих чисел.

Вариант 5. Вывести сумму модулей чисел, являющихся целыми (массив типа **double**), вернуть количество нецелых чисел.

Вариант 6. Завести две суммы (вначале содержат нули). Проходя элементы массива по порядку, добавлять их к первой сумме, если это не сделает ее отрицательной, иначе добавлять ко второй сумме.

Вывести вторую сумму, вернуть первую.

Вариант 7. Завести две суммы (вначале содержат нули). Пройти массив по порядку, добавляя к первой сумме те элементы, что больше ее по модулю, остальные добавлять ко второй сумме.

Вывести первую сумму, вернуть вторую.

Вариант 8. Добавлять модуль числа к сумме, если оно четное, вычитать, если нечетное. Вывести количество нечетных чисел. Вернуть накопленную сумму.

Вариант 9. Выводить каждое пятое число и вернуть максимум из прочих.

Вариант 10. Вернуть количество элементов, делящихся на десять нацело. Вывести с их индексами прочие элементы.

Вариант 11. Вывести все элементы x такие, что $\frac{\sin x}{x} > 0.99$. Вернуть минимум из модулей прочих элементов.

Вариант 12. Вывести все положительные элементы и их индексы. Вернуть корень квадратный из суммы квадратов отрицательных элементов.

Вариант 13. Вывести элементы, не попадающие в полуинтервал $(1, 2]$, и их индексы. Вернуть среднее геометрическое элементов, попадающих в $(1, 2]$.

Вариант 14. Вывести минимум модулей элементов, не превосходящих единицу по модулю. Вернуть сумму обратных значений прочих элементов (под обратным к a здесь понимается $\frac{1}{a}$).

Вариант 15. Вывести все элементы, не попавшие в отрезок $[-100, 500]$, и их индексы. Вернуть арифметическое среднее прочих чисел.

Вариант 16. Вывести максимум среди элементов, являющихся отрицательными или нечетными (в целых числах), вернуть минимум прочих (неотрицательных и четных).

8

Процедурное программирование

Цели лабораторной работы

- Изучить принцип разработки сверху-вниз.
- Изучить набор элементарных математических функций, предоставляемых стандартной библиотекой C++.

Математическая модель

Рассмотрим задачу моделирования вертикального полета летательного аппарата с реактивным двигателем.

Предположения модели:

- Максимальная высота подъема значительно меньше радиуса планеты — пренебрегаем зависимостью ускорения свободного падения от высоты.
- Атмосфера отсутствует или имеет малую плотность — пренебрегаем аэродинамическим сопротивлением и воздействием ветра.

- Двигатель включается и выключается мгновенно, во время работы двигателя скорость расхода рабочего тела постоянна.

- Удельный импульс двигателя (эффективная скорость истечения рабочего тела) постоянен.

Итак, имеем следующие постоянные:

- g — ускорение свободного падения, $\frac{\text{м}}{\text{с}^2}$;

- I — удельный импульс реактивного двигателя, $\frac{\text{м}}{\text{с}}$.

Введем следующие переменные:

- t — время полета, с;

- m — масса аппарата, кг;

- x — высота аппарата, отсчитываемая от поверхности планеты (0) вверх, м;

- v — скорость (положительная в случае подъема, отрицательная в случае спуска), $\frac{\text{м}}{\text{с}}$;

- w — ускорение, $\frac{\text{м}}{\text{с}^2}$.

Индексом 0 обозначим момент включения двигателя, индексом 1 — момент выключения двигателя. Тогда $\Delta t = t_1 - t_0$ есть время работы двигателя. Аналогичный смысл дадим обозначению Δ и при применении к прочим переменным. Через Δv_r обозначим изменение скорости, приобретаемое в результате действия реактивной тяги, вычисляемое по формуле Циолковского. Благодаря предположениям модели изменения скорости под действием реактивной силы и гравитации можно

просто сложить:

$$\begin{aligned}\Delta v &= \Delta v_r - g\Delta t,\\ \Delta v_r &= I \ln \frac{m_0}{m_1},\\ m_1 - m_0 &= \Delta m.\end{aligned}$$

Подставив вместо Δt переменную $0 \leqslant \tau \leqslant \Delta t$ $(t = t_0 + \tau)$, можно выразить функции изменения массы и скорости аппарата:

$$\begin{aligned}m(\tau) &= m_0 + r\tau, \quad r = \frac{\Delta m}{\Delta t} \leqslant 0,\\ v(\tau) &= v_0 + I \ln \frac{m_0}{m(\tau)} - g\tau.\end{aligned}$$

Ускорение $w(\tau)$ получим как производную скорости:

$$w(\tau) = -\frac{Ir}{m(\tau)} - g.$$

Проинтегрировав скорость, получим высоту $x(\tau)$:

$$x(\tau) = x_0 + v_0\tau - \frac{g}{2}\tau^2 + I\left(\tau + \frac{m(\tau)}{r} \ln \frac{m_0}{m(\tau)}\right).$$

Итак, пусть заданы начальные значения t_0, m_0, v_0, x_0 и целевые Δt и Δm. Тогда не составит труда вычислить конечные значения переменных (при $\tau = \Delta t$):

$$\begin{aligned}t_1 &= t_0 + \Delta t, \quad m_1 = m_0 + \Delta m,\\ w_1 &= -\frac{Ir}{m_1} - g, \quad v_1 = v_0 - g\Delta t + I \ln \frac{m_0}{m_1},\\ x_1 &= x_0 + v_0\Delta t - \frac{g}{2}\Delta t^2 + I\left(\Delta t + \frac{m_1}{r} \ln \frac{m_0}{m_1}\right).\end{aligned}$$

Программная реализация

Для реализации вышеприведенных формул на языке C++ нам пригодится стандартный заголовочный файл **cmath**. В частности, объявленная там функция log вычисляет натуральный логарифм.

Для выполнения моделирования можно определить следующие глобальные переменные:

```
// Параметры модели:
double g;
double I;
double empty_m, fuel_m;

// Переменные модели:
double t0, t1;
double m0, m1;
double x0, x1;
double v0, v1;
double w0, w1;
```

Параметр empty_m задает массу аппарата без топлива, а fuel_m — начальную массу топлива.

Впрочем, следует отметить, что хотя формально все это — числа, они имеют различный физический смысл. Чтобы подчеркнуть этот факт, можно воспользоваться средствами системы типов языка программирования и определить для каждой физической величины свой тип. В данной программе мы ограничимся симуляцией разных типов через объявление синонимов типов.

```
// Типы:
using Scalar = double;
using Acceleration = Scalar; // м/с2
using Velocity = Scalar;     // м/с
using Position = Scalar;     // м
using Time = Scalar;         // с
using Mass = Scalar;         // кг
```

```
// Параметры модели:
Acceleration g;
Velocity I;
Mass empty_m, fuel_m;

// Переменные модели:
Time t0, t1;
Mass m0, m1;
Position x0, x1;
Velocity v0, v1;
Acceleration w0, w1;
```

Процесс моделирования требует задания значений параметров модели. Главная процедура сводится к циклу, повторяющему ввод параметров модели и симуляцию (две подзадачи).

Пример 8.1. Главная процедура

```
int main() {
  // Объявления процедур:
  void select_model();
  void simulate();
  // Вечный цикл итераций.
  for (;;) {
    // Установить параметры модели.
    select_model();
    // Моделирование.
    simulate();
    // Сброс и очистка потока ввода.
    cin.clear();
    cin.sync();
    cin.ignore(cin.rdbuf()->in_avail());
  }
  return 0;
}

// Заглушки:
void select_model() {
  cout << "select_model\n";
}
```

```
void simulate() {
  cout << "simulate\n";
}
```

Выбор параметров модели сводится к определению значений g, I и empty_m. Вынесем его в отдельные процедуры, чтобы дать возможность выбрать из заранее заготовленного набора значений.

Пример 8.2. select_model

```
void select_model() {
  // Объявления процедур:
  void select_g();
  void select_I();
  void select_craft();
  // Выбрать каждый параметр.
  cout << "Model setup\n";
  select_g();
  select_I();
  select_craft();
}

// Заглушки:
void select_g() {
  cout << "select_g\n";
}

void select_I() {
  cout << "select_I\n";
}

void select_craft() {
  cout << "select_craft\n";
}
```

В принципе, на данный момент конструкция столь проста, что не нуждается в отладке. Хотя всегда полезно на каждом этапе собрать (убедиться в отсутствии ошибок компиляции) и запустить текущую версию программы — это облегчает локализацию ошибок.

Определим функции select_g и select_I. Например:

Пример 8.3. select_g

```
void select_g() {
  Acceleration static const
    g_mercury = 3.7,    // Меркурий
    g_moon    = 1.62,   // Луна
    g_europa  = 1.315,  // Европа
    g_pluto   = 0.617,  // Плутон
    g_ceris   = 0.27;   // Церера
  cout << "\nSelect location:\n"
    "0. No gravity\n"
    "1. Mercury\n"
    "2. Moon\n"
    "3. Europa\n"
    "4. Pluto\n"
    "5. Ceris\n";
  switch (cin.get()) {
  case '0': g = 0.0;         break;
  case '1': g = g_mercury;   break;
  case '2': g = g_moon;      break;
  case '3': g = g_europa;    break;
  case '4': g = g_pluto;     break;
  case '5': g = g_ceris;     break;
  default:
    cout << "enter value: g = ";
    cin >> g;
  }
  cin.ignore();
}
```

Теперь перейдем к подзадаче собственно моделирования.

Вначале у нас есть только значения параметров модели. Требуется задать начальные значения переменных. После чего организовать цикл ввода команды «назначить режим работы двигателя». Режим работы двигателя определяется целевыми значениями расхода топлива Δm и времени Δt, за которое этот расход осуществляется.

После выполнения каждой такой команды требуется вывести на экран состояние системы и переиспользовать полученные конечные значения переменных как начальные значения для следующего шага. В случае пересечения нулевой отметки высоты — закончить (посадка или падение).

Пример 8.4. Функция simulate

```
void simulate() {
  // Объявления процедур:
  void init_model();
  void report_model();
  void next_step();
  bool stop_simulation();
  void simulate_step(Mass, Time);
  // Начальные приготовления.
  init_model();
  cout << "Simulation loop\n"
    "Enter objective delta-m and delta-t\n";
  // Цикл моделирования.
  Mass dm = 0;
  Time dt = 0;
  while (cin >> dm >> dt) {
    simulate_step(dm, dt);
    report_model();
    if (stop_simulation())
      break;
    next_step();
  }
}

// Заглушки:
bool stop_simulation() {
  cout << "stop_simulation?\n";
  return false;
}

void simulate_step(Mass dm, Time dt) {
  cout << "simulate_step("
       << dm << ", " << dt << ')'
```

```
        << ", r = " << dm/dt << '\n';
}
```

Две из подзадач можно записать сразу — они очень просты:

```
// Инициализация модели до начала моделирования.
void init_model() {
  t0 = 0;
  m0 = empty_m + fuel_m;
  x0 = 0;
  v0 = 0;
  w0 = 0;
}

// Подготовка следующего шага моделирования.
void next_step() {
  t0 = t1;
  m0 = m1;
  x0 = x1;
  v0 = v1;
  w0 = w1;
}
```

Здесь может возникнуть вопрос: обязательно ли стартовать с нулевой отметки с нулевой скоростью? Ответ, конечно же, отрицательный. Можно добавить ввод x0 и v0, но для упрощения программы мы не будем этого делать.

Еще две подзадачи тоже просты:

```
// Вывод параметров на экран.
void report_model() {
  cout << "t = " << t1;
  cout << " m = " << m1 << "(fuel: "
       << (m1 - empty_m) << ")\n";
  cout << "x = " << x1 << ", v = " << v1;
  cout << ", w = " << w0 << ".." << w1 << "\n\n";
}

// Проверка условий завершения моделирования.
bool stop_simulation() {
  return x1 < 0;
}
```

Полученная в результате применения метода разработки «сверху-вниз» программа имеет следующую структуру:

```
main -> int
  select_model
    select_g
    select_I
    select_craft
  simulate
    init_model
    report_model
    next_step
    stop_simulation -> bool
    simulate_step(Mass dm, Time dt)
```

Задание

Задача лабораторной работы заключается в продолжении разработки (доработке) данной программы. Осталось решить основную подзадачу — моделирование шага работы двигателя. Для этого следует воспользоваться итоговыми формулами из раздела «Математическая модель».

Попробуйте также ответить на следующие вопросы:

1. Что произойдет, если пользователь задаст $\Delta t = 0$?
2. Что произойдет, если пользователь задаст $\Delta t < 0$?
3. Правильно ли моделируется ситуация, когда пользователь задал $\Delta m = 0$ (свободное падение)?
4. Правильно ли моделируется ситуация, когда пользователь задал $\Delta m <$ (масса-пустого$-m_0$), т. е. когда топлива не хватает?
5. Как добавить проверку успешности посадки?
6. Как добавить ввод начальных позиции и скорости?

9

Элементарные вычисления

Цели самостоятельной работы

- Закрепление навыков выполнения элементарных вычислений на языке C++.
- Изучение способов возвращения более одного значения из функции.

В данном задании акцентируется важность детального изучения различных возможных (пусть и маловероятных) случаев, комбинаций значений параметров задачи.

Критерии полноты

1. Реализована функция, вычисляющая $f(x, a, b, c)$. Упрощение выражения не допускается.
2. Реализована функция вычисления (одного) корня для заданных значений параметров (a, b, c).
3. Функция вычисления корня проверяет все возможные случаи несуществования корней и существования несчетного множества корней (например, когда подходит любое

действительное число). Допустимы некоторые послабления — см. пример 9.5.

4. Реализована программа, позволяющая пользователю ввести произвольное количество наборов значений параметров (a, b, c) и для каждого набора выводящая либо некоторый корень, либо сообщение о несуществовании корня, либо сообщение о существовании несчетного множества корней.

5. При обнаружении корня программа должна выполнять подстановку его и исходных параметров в $f(x, a, b, c)$ и выводить полученное значение. Его близость нулю свидетельствует в пользу корректности решения.

9.1. Примеры

Задание 1. Дано $f(x) = ax + b$.

Начнем с упрощенного примера, заключающегося в решении скалярного линейного уравнения.

Пример 9.1. Решатель линейного уравнения v1

```
float solve_linear(float a, float b) {
  // Неужели так просто?
  return -b / a;
}
```

Можно поэкспериментировать с этим кодом, вводя разные a и b. Что произойдет, если ввести $a = 0$? А если и $b = 0$?

Очевидно, что решений у конкретного уравнения с конкретными значениями параметров может и не быть (случай $a = 0$ и $b \neq 0$), а может быть бесконечно много (случай $a = 0$ и $b = 0$). Простейшим подходом будет заявить «наша функция возвращает решение уравнения, если оно существует и единственно», но это означает перекладывание задачи по определению возможности решения уравнения на пользователя программы.

Более удобной для пользователя представляется функция, которая проверяет существование корня и возвращает некое описание ситуации, например: «корень получен», «корней не существует», «(почти) любое число подходит в качестве корня». Этого можно добиться, возвращая числовой код ситуации либо характеристику «количество корней». Отрицательное число будет означать ситуацию «почти любое число подходит в качестве корня».

Однако, если функция возвращает количество корней, то нужно решить, каким образом она будет возвращать одновременно и сам корень, если он существует. C++ располагает рядом способов решения данной проблемы. Предпочтительный способ зависит от конкретной ситуации. Мы выберем простейший с точки зрения требуемого объема знаний способ — передачу в функцию по ссылке переменной, в которую записывается второе возвращаемое значение. Так можно вернуть из функции сразу несколько значений за один вызов, но под них придется заранее завести переменные-приемники в вызывающем коде. Если эти значения однотипны, то в качестве альтернативы может выступать запись их из функции в массив.

Функцию, решающую линейное уравнение, перепишем так, чтобы в качестве основного результата она возвращала количество корней (тогда его удобно будет сразу проверять в условии инструкций **if** или **switch**), а вычисленный корень записывала в переменную, переданную по ссылке в качестве параметра.

Пример 9.2. Решатель линейного уравнения v2

```
// Особое значение «бесконечное количество корней».
int const INFINITE_ROOTS = -1;
// Функция возвращает «количество корней».
// Корень записывает по ссылке root.
int solve_linear(float a, float b, float & root) {
  if (a == 0) return b == 0? INFINITE_ROOTS: 0;
  root = -b / a;
  return 1;
}
```

Задание 2. Дано $f(x) = ax^2 + bx + c$.

Квадратное уравнение представляет собой более интересный объект. Все коэффициенты могут принимать любые значения, соответственно, у нас может быть 0, 1, 2 корня или все действительные числа могут быть корнями. Случай $a = 0$ мы уже умеем решать с помощью функции solve_linear. И как раз удобно, что она возвращает количество корней, поэтому можно просто записать вызов solve_linear в инструкции **return**.

Итак, функция solve_quadratic возвращает до двух корней, записывая их в переменные, переданные по ссылкам (запись в root2 осуществляется только в случае двух различных корней).

Пример 9.3. Решатель квадратного уравнения

```
int solve_quadratic(
    float a, float b, float c,
    float & root1, float & root2) {
  if (a == 0) // сводится к линейному
    return solve_linear(b, c, root1);
  // ниже a != 0

  float const d = b*b - 4*a*c;
  if (d < 0) // нет корней
    return 0;
  if (d == 0) { // один корень
    root1 = -b / (2*a);
    return 1;
  }

  // Два корня.
  float const ds = sqrt(d);
  root1 = (-b - ds) / (2*a);
  root2 = (-b + ds) / (2*a);
  return 2;
}
```

Задание 3. Дано $f(x) = 1 + \sin(a^x + |\log_b c|)$.

Положим $f(x) = 0$ и выразим формально x через значения параметров a, b и c. Получим

$$x = \log_a(-|\log_b c| - \pi/2 + 2\pi n), \quad n \in \mathbb{Z}.$$

Итак, здесь может быть счетное множество корней, но по заданию нам достаточно найти один. Сначала, впрочем, проанализируем эту формулу:

- При $a \leqslant 0$, или $a = 1$, или $b \leqslant 0$, или $b = 1$, или $c \leqslant 0$ значение формулы не определено. Исходя из этого можно подумать, что тогда и корней нет. На самом деле следует рассмотреть указанные случаи применительно к исходной формуле $f(x)$: случаи $b \leqslant 0$, $b = 1$ и $c \leqslant 0$ однозначно дают отсутствие решений. А вот ситуация с a особая (см. следующие пункты).

- Случай $a = 1$: корней нет, если $1 + \sin(1 + |\log_b c|) \neq 0$, в противном случае подходит любое x.

- Случай $a = 0$: корней нет, если $1 + \sin(|\log_b c|) \neq 0$, в противном случае подходит любое $x > 0$.

- Случай $a < 0$: множество выбора x следует ограничить целыми числами. В данном задании для простоты будем считать, что корней в этой (и подобных) ситуации нет.

- В оставшихся случаях имеем счетное множество корней, среди которых следует выбрать один. Для того чтобы формула корня имела смысл, выражение под логарифмом должно быть положительным, поэтому достаточно, например, положить (скобки $\lceil\ \rceil$ означают округление до ближайшего сверху целого):

$$n = 1 + \left\lceil \frac{|\log_b c|}{2\pi} + \frac{1}{4} \right\rceil.$$

Приведем код функции, решающей рассмотренное уравнение с учетом указанных выше упрощений.

Пример 9.4. Решение трансцендентного уравнения

```
// Решение уравнения f(x) = 0.
int solve_f(float a, float b, float c,
    float & root) { // возвращаем один корень
  if (a < 0 || b <= 0 || b == 1 || c <= 0)
    return 0; // нет корней
  // Потенциально почти все возможные x — корни.
  if (a == 0 || a == 1)
    return is_almost_zero(f(a, b, c, 1))?
              INFINITE_ROOTS: 0;
  // Счетное число корней, получим один из них.
  float const
    // Часть выражения:
    expr_part = fabs(log(b, c)) + Half_Pi,
    // Номер корня:
    n = 1 + ceil(expr_part / Double_Pi),
    // Аргумент логарифма в формуле корня:
    log_arg = Double_Pi*n - expr_part;
  assert(log_arg > 0);
  // Вычислим собственно корень.
  root = log(a, log_arg);
  return 1;
}
```

Здесь INFINITE_ROOTS уже является признаком не просто бесконечного множества корней, но несчетного множества корней (например, все действительные числа, кроме нуля).

Функция is_almost_zero проверяет значение на близость нулю. Она заменяет проверку на равенство нулю, которая может плохо работать на практике из-за погрешности машинных вычислений. Параметр tolerance задает допустимую величину отклонения проверяемого значения от нуля:

```
// Проверка значения на близость нулю.
float const TOLERANCE = 0.0001f;
bool is_almost_zero(float x,
    float tolerance = TOLERANCE) {
  return fabs(x) <= tolerance;
}
```

Теперь приведем весь код решения данного примера. Этот код можно использовать как образец при решении своего варианта задания.

Пример 9.5. Пример решения

```
#include <iostream>
#include <cmath>
#include <cassert> // assert
using namespace std;

// Особое значение «бесконечное количество корней».
int const INFINITE_ROOTS = -1;
// Используемый числовой тип (Number Type).
using NT = float;

NT const // Вспомогательные числовые константы:
  HALF_PI   = 1.5707963267949,  // половина пи
  PI        = 3.14159265358979, // число пи
  DOUBLE_PI = 6.2831853071796,  // два пи
    // граница между "нулем"и "ненулем"
  TOLERANCE = 1e-10;

// Логарифм по произвольному основанию.
NT log(NT base, NT arg) {
  return log(arg) / log(base);
}

// Проверка значения на близость нулю.
bool is_almost_zero(
    NT x, NT tolerance = TOLERANCE) {
  return fabs(x) <= tolerance;
}
```

```
// Левая часть уравнения.
NT f(NT a, NT b, NT c, NT x) {
  return 1 + sin(pow(a, x) + fabs(log(b, c)));
}

// Решаем уравнение f(a, b, c, root) = 0.
// Функция возвращает «количество корней»,
// один корень записывает по ссылке.
int solve_f(NT a, NT b, NT c, NT & root) {
  if (a < 0 || b <= 0 || b == 1 || c <= 0)
    return 0; // нет корней.
  // Потенциально почти все возможные x — корни.
  if (a == 0 || a == 1)
    return is_almost_zero(f(a, b, c, 1))?
       INFINITE_ROOTS: 0;

  // Счетное число корней, получим один из них.
  NT const
    // Часть выражения:
    expr_part = fabs(log(b, c)) + HALF_PI,
    // Номер корня:
    n = 1 + ceil(expr_part / DOUBLE_PI),
    // Аргумент логарифма в формуле корня:
    log_arg = DOUBLE_PI*n - expr_part;

  assert(log_arg > 0);
  root = log(a, log_arg);
  return 1;
}

int main() {
  cout << "Solving f(a, b, c, x) = 0,"
       << " enter a, b, c:\n";
  cout.precision(16);
  for (NT a, b, c, x; cin >> a >> b >> c;) {
    switch (solve_f(a, b, c, x)) {
    case 0:
      cout << "no roots\n";
      break;
    case INFINITE_ROOTS:
```

```
        cout << "any number is a root\n";
        break;
      case 1: // один корень, записан в x
        cout << "x == " << x
             << ", error is " << f(a, b, c, x)
             << endl;
        break;
      default:
        assert(!"impossible roots quantity");
        cout << "unknown error\n";
        return 1; // ошибка
      }
    }

  return 0;
}
```

Благодаря строчке

```
using NT = float;
```

компилятор понимает NT как **float**. При желании мы можем заменить его в одном месте, например, на **double**:

```
using NT = double;
```

и тогда NT везде станет **double**.

Это удобно, поскольку можно заменить тип в одном месте вместо того, чтобы заменять его во многих местах с перспективой где-то забыть заменить или заменить что-то не то.

Полезно поэкспериментировать с разными типами и посмотреть качество вычисления решения при использовании типа **float** и при использовании типа **double**.

9.2. Варианты заданий

Вариант 1. $f(x) = a^c x^{-2} + \dfrac{x^7 \sin b}{a^2 - c^2}$.

Вариант 2. $f(x) = \exp(a \exp(a + bx)) - c^{10}$.

Вариант 3. $f(x) = 2\cos(a \ln(bx^2 - c^2) + c^2) + \sqrt{2}$.

Вариант 4. $f(x) = \operatorname{arctg}\left(a \exp \dfrac{b \sin^3 x}{c} + 1\right) + \dfrac{\pi}{4}$.

Вариант 5. $f(x) = \operatorname{ctg}\left(12 + \dfrac{b^a}{x^2} - 2\pi ab\right) + 4c^2$.

Вариант 6. $f(x) = \sqrt[7]{\lg\left(1 + \dfrac{ax^3}{100(b^2 + c^2)}\right)} - \dfrac{b}{100(a^2 + c^2)}$.

Вариант 7. $f(x) = \sin(a\pi \cos(b^2 \operatorname{tg} c^3 x^4))$.

Вариант 8. $f(x) = \sin^2 \dfrac{a^2 x^{3b}}{\operatorname{tg} 2^c} + c$.

Вариант 9. $f(x) = |ab| - \cos\left(\dfrac{a+1}{b-1} - 2x^c\right)$.

Вариант 10. $f(x) = \exp\left(a \operatorname{tg}\left(b^2 x^2 - a^4\right)\right) - c^3$.

Вариант 11. $f(x) = b + |\sin \pi c| - \dfrac{c2^a}{\arcsin x^a}$.

Вариант 12. $f(x) = -c + \dfrac{\cos 2a^{2x+b}}{\ln c}$.

Вариант 13. $f(x) = 1 + 8\log_a b - \left(\dfrac{ab}{c^3}\right)^{ax^2}$.

Вариант 14. $f(x) = \sqrt{a^2 - \sqrt{b^2 + \dfrac{1}{\sin cx^3}}} - c^2$.

Вариант 15. $f(x) = 1 - 3bc - c^{-c}\operatorname{arctg}\left(3ax^9 + b\right)$.

Вариант 16. $f(x) = b^2 - \left|c\ln\cos\dfrac{2a}{c - x^2}\right|$.

10

Массивы и двоичные файлы

Цели лабораторной работы

- Изучить конструкцию «массив».
- Изучить работу с двоичными файлами средствами стандартной библиотеки C++.

Простое гаммирование

Рассмотрим следующую задачу. Есть две последовательности целых чисел. Одну из них назовем *ключ*, а другую — *текст*. Используя ключ, требуется получить новый текст — *шифротекст* (зашифровать текст), из которого можно восстановить исходный текст, используя тот же ключ (расшифровать текст).

Шифры, использующие один и тот же ключ для зашифровки и расшифровки, называют *симметричными*. Если для зашифровки и расшифровки используются разные ключи, да еще и такие, что вычисление по одному ключу другого является вычислительно очень сложной задачей, то такие шифры называют *асимметричными*. Асимметричные шифры решают

проблему безопасной передачи ключа. В случае симметричного шифра ключ требуется передавать отдельно по надежному каналу связи[2]. Доступ третьих лиц к ключу компрометирует шифр.

Один из простейших симметричных шифров для последовательностей целых основан на поэлементном сложении (вычитании) текста и ключа, называемом *гаммированием*. Если значение должно находиться в определенных пределах (например, байт — от 0 до 255), то сложение и вычитание можно делать по модулю (с оборачиванием). Однако с технической точки зрения при выполнении данной операции проще заменить и сложение, и вычитание одной операцией — поразрядным исключающим *или*[3]. Тогда применение ключа к тексту дает шифротекст, а применение ключа к шифротексту дает исходный текст.

Если длина ключа меньше длины текста, то ключ можно повторять столько раз, сколько нужно. Например, если дан текст "Hello,␣world" и ключ "+ZY−", то имеем:

текст	H	e	l	l	o	,		w	o	r	l	d
ASCII	48	65	6C	6C	6F	2C	20	77	6F	72	6C	64
ключ	+	Z	Y	−	+	Z	Y	−	+	Z	Y	−
ASCII	2B	5A	59	2D	2B	5A	59	2D	2B	5A	59	2D
ш/т	c	?	5	A	D	v	y	Z	D	(	5	I
ASCII	63	3F	35	41	44	76	79	5A	44	28	35	49

Задание 1

Пусть пользователь вводит две строки: текст и ключ. Примените ключ к тексту и выведите в поток ввода шифротекст. Примените ключ к шифротексту и проверьте совпадение результата с исходным текстом.

В данной задаче удобно использовать std::string. Размер строки s можно получить с помощью вызова s.size() (возвращает число типа size_t).

---

[2] Например, созданному с помощью асимметричного шифра.

[3] Данный метод был запатентован в 1919 г. Г. Вернамом для шифрования телеграфных передач.

Шифр Вернама

Повторение ключа делает шифр уязвимым. Однако, если ключ имеет ту же длину, что и текст, используется лишь один раз (для каждой передачи создается свой уникальный ключ) и является истинно случайным, то рассмотренный выше метод (также часто называемый *шифр Вернама* [*Vernam cipher, one-time pad*]) является устойчивым к криптоанализу[4].

Реализуем консольное приложение vernam, применяющее шифр Вернама. Текст или шифротекст задан в виде (двоичного) файла. Ключ задан в виде другого (двоичного) файла. Результат (шифротекст или текст) сохраним в третий файл.

Имена файлов будем передавать параметрами командной строки:

- –I*input* — имя файла с (шифро)текстом;
- –K*key* — имя файла с ключом;
- –O*output* — имя файла результата.

Например, команда

```
vernam –Imessage.txt –Kkey.dat –Obits.bin
```

должна взять файл message.txt, применить к нему ключ из файла key.dat и сохранить результат в bits.bin.

Будем считать ошибкой пользователя ситуации:

- задан параметр, не являющийся именем одного из трех файлов;
- не задано имя какого-либо из трех файлов;
- имя какого-либо из трех файлов задано более одного раза;
- невозможно открыть какой-либо из файлов;

[4] См.: *Shannon C.* Communication Theory of Secrecy Systems // Bell System Tech. J. 1949. № 28(4). P. 656–715.

- файл ключа меньше файла текста (данный факт допустимо определять уже во время работы).

В каждой из перечисленных ситуаций должно выводиться соответствующее сообщение в поток вывода ошибок и приложение должно возвращать ненулевой код ошибки.

В случае ошибки во время исполнения приложение должно возвращать ненулевой код ошибки.

Коды ошибок должны быть разными для всех ситуаций.

Применим метод разработки сверху-вниз.

Основная процедура — функция main. Есть стандартный способ получения параметров командной строки через параметры функции main:

```
int main(int argc, char const * argv[]) { ... }
```

Здесь argc — число параметров командной строки, argv — массив указателей на нуль-терминированные строки (си-строки), содержащие параметры командной строки.

Главная процедура сводится к решению двух подзадач: разбору набора параметров и выполнению гаммирования. К решению второй подзадачи приступаем только при условии успешности решения первой подзадачи.

```
int main(int argc, char const * argv[]) {
  // Объявления процедур. Возвращают код ошибки.
  int parse_params(int argc, char const * argv[]);
  int vernam_encode();
  // Разбор параметров.
  int error = parse_params(argc, argv);
  if (error != 0)
    return error;
  // Применение шифра.
  error = vernam_encode();
  return error;
}
```

Заметим, впрочем, что заранее заготовленный набор кодов ошибок в виде перечислимого типа удобнее, чем «чистый»

int. Учитывая перечисленные выше требования, список ошибок можно сделать таким:

```
enum Error {
  No_error = 0,          // нет ошибок.
  Unknown_param,         // неизвестный параметр.
  Not_enough_params,     // недостаточно параметров.
  Excessive_params,      // лишние параметры.
  Bad_file_name,         // не удается открыть файл.
  File_error,            // ошибка ввода-вывода файла.
  Too_short_key };       // слишком короткий ключ.
```

Добавим также процедуру уведомления пользователя об ошибке report_error. Итак, имеем:

```
int main(int argc, char const * argv[]) {
  // Объявления процедур. Возвращают код ошибки.
  Error parse_params(int argc, char const * argv[]);
  Error vernam_encode();
  void report_error(Error);
  // Разбор параметров.
  Error error = parse_params(argc, argv);
  // Применение шифра.
  if (error == No_error)
    error = vernam_encode();
  // Вывод сообщения об ошибке.
  report_error(error);
  return error;
}
```

Для работы с файлами будем использовать стандартные файловые потоки C++:

```
#include <fstream>
std::fstream input, key, output;
```

По принципу использования данные объекты аналогичны cin и cout, благодаря чему не составит труда написать функцию vernam_encode в виде цикла чтения, обрабатывающего по одному символу за итерацию:

```
Error vernam_encode() {
  for (char a, b; input.get(a) && key.get(b)
```

```
                    && output.put(a ^ b);)
    continue;
  if (input.bad() || key.bad() || !output.good())
    return File_error;
  if (input && !key)
    return Too_short_key;
  return No_error;
}
```

Вообще, чтение и запись по одному байту могут быть медленными. Быстрее обрабатывать данные блоками размера от 16 байт и больше. К сожалению, длина файла может не делиться нацело на выбранный размер блока, это усложняет алгоритм.

Создадим два глобальных статических массива символов, например, в 64 кб.

```
size_t const BUF_SZ = 64*1024;
using Buffer = char[BUF_SZ];
Buffer io_buf, key_buf;
```

Переделаем vernam_encode для обработки блоков. Для этого вынесем собственно гаммирование в отдельную функцию vernam_encode_buffer. Для чтения блока используется функция readsome, для записи — write. Данные функции получают указатель на блок и его размер в байтах. Функция readsome возвращает количество действительно прочитанных байт.

```
Error vernam_encode() {
  // Собственно процедура гаммирования.
  void vernam_encode_buffer(size_t n);
  // Блоками по размеру буфера.
  while (auto const in_bytes_read =
         input.readsome(io_buf, BUF_SZ)) {
    auto const key_bytes_read =
         key.readsome(key_buf, BUF_SZ),
               bytes_to_write =
                 in_bytes_read < key_bytes_read?
                 in_bytes_read : key_bytes_read;
    vernam_encode_buffer(bytes_to_write);
    if (!output.write(io_buf, bytes_to_write))
```

```
      return File_error;
    if (key_bytes_read < in_bytes_read)
      return Too_short_key;
  }
  return input.bad()? File_error: No_error;
}
```

Функция гаммирования очень проста:

```
void vernam_encode_buffer(size_t n) {
  for (size_t i = 0; i < n; ++i)
    io_buf[i] ^= key_buf[i];
}
```

Современный компилятор при включении оптимизации способен использовать здесь регистр наибольшего доступного размера, чтобы обрабатывать данные блоками 4–64 байта за итерацию.

Новый вариант может оказаться быстрее старого на порядок. Впрочем, нужно учитывать влияние кэширования записи, выполняемого современными ОС: реальная запись может продолжаться еще долго после того, как наша программа уже закончила работу. Поэтому, например, если запись осуществляется на съемный носитель, его нельзя отключать сразу по завершении программы, а требуется выполнить процедуру размонтирования средствами ОС.

Задание 2

Сравните варианты обработки по одному байту и блоками на реальных файлах достаточно большого размера. Проверьте идентичность результатов.

Использование хэширования

Теперь вернемся к ситуации короткого ключа. Например, ключа, заданного «паролем». Если в качестве (повторяющегося) ключа использовать пароль непосредственно, то стойкость такого шифра оказывается весьма низкой.

Хэш-функция [*hash function*] — отображение произвольной строки в конечное множество. **Хэш** — значение хэш-функции.

Если строки совпадают, то и их хэши совпадают. Но из того, что хэши двух строк совпадают, в общем случае не следует, что эти строки совпадают, поскольку множество строк счетно (бесконечно), а множество хэшей — конечно.

Ситуация совпадения хэшей разных строк называется *коллизией* [*collision*]. Если хэш-функция вводится на конечном множестве строк, то существует возможность сделать ее инъективной, т. е. такой, что хэши всех строк из этого конечного множества будут различны. В этом случае такую хэш-функцию называют *совершенной* [*perfect*].

Хорошая хэш-функция должна обеспечивать «хорошее перемешивание[5]» исходной строки: результат должен выглядеть как случайное число.

На практике хэш-функции разделяют на два класса: криптографические и некриптографические.

Некриптографические хэш-функции обычно вычисляют хэш как 32-битное или 64-битное целое. Они должны обеспечивать быстрое вычисление хэша при как можно более низкой вероятности случайной коллизии.

Криптографические хэш-функции вычисляют хэш как битстроку заданной длины (обычно 128 бит или более). Главное их качество — очень высокая вычислительная сложность обращения хэша [*hash forging*], т. е. вычисления по заданному хэшу любой строки, хэш которой равен заданному.

Криптографические хэш-функции используются в криптографических протоколах (шифрование, электронная подпись), проверке и подтверждении целостности данных (например, вычисленный хэш полученного файла должен совпасть с переданным).

Особое место занимают криптографические хэш-функции, используемые для организации учетных записей пользовате-

[5] Собственно, англ. *to hash* (от фр. *hacher*) означает «мелко порубить, нашинковать».

лей, защищенных паролем: вместо паролей в базе данных хранятся их хэши[6].

Итак, пусть есть (криптографическая) функция $H(s)$, отображающая произвольные бит-строки в бит-строки длины b. Если мы ее применим к паролю, то независимо от его длины мы получим бит-строку длины b, напоминающую случайную последовательность бит. Однако, если она будет повторяться, толку от этого будет мало. Довольно простым выходом из ситуации может быть использование хэш-функции на каждом шаге: вместо того, чтобы повторять ключ, брать хэш ключа:

$$\begin{aligned} k_0 &= H(\text{пароль}), \\ c[(i-1)b \ldots ib] &= t[(i-1)b \ldots ib] \operatorname{xor} k_{i-1}, \\ k_i &= H(k_{i-1}), \quad i = 1, 2, \ldots \end{aligned}$$

Здесь c обозначает шифротекст, t — исходный текст.

Впрочем, вся последовательность хэшей будет определяться исходным паролем и только им. Знание любой части переданных данных позволит восстановить или подделать весь «хвост» передачи. Лучше сделать так, чтобы ключ определялся не только паролем, но и текстом, тогда каждая передача будет уникальна, и любой участок будет также зависеть от предыдущих данных.

Предлагаемая схема известна как *режим обратной связи по шифротексту.* (Знаком | обозначена конкатенация строк.)

$$\begin{aligned} k_{-1} = H(\text{пароль}), \quad k_0 &= H(k_{-1}|Iv), \\ c[(i-1)b \ldots ib] &= t[(i-1)b \ldots ib] \operatorname{xor} k_{i-1}, \\ k_i = H(k_{-1}|k_{i-1}|c[(i-1)b \ldots ib]), \quad i &= 1, 2, \ldots \end{aligned}$$

Значение Iv называется *вектором инициализации.* Это значение является (желательно уникальным и истинно случайным) идентификатором передачи. Оно не обязано быть сек-

[6] Обычно это хэши конкатенации пароля и случайно сгенерированной строки фиксированной ширины, называемой *солью.*

ретным, его смысл — в затруднении использования статистики накопленных передач для взлома.

Значение k_{-1} является секретным и для его восстановления из k_i требуется уметь обращать хэш-функцию $H(s)$, что предполагается практически невозможным, поэтому его подмешивание на каждом шаге вычисления k_i делает затруднительным вычисление очередных k_{i+j}, даже если удалось узнать k_i, $i > -1$.

К сожалению, теперь обратное и прямое преобразования не совпадают:

$$k_{-1} = H(\text{пароль}), \quad k_0 = H(k_{-1}|Iv),$$
$$t[(i-1)b \dots ib] = c[(i-1)b \dots ib] \operatorname{xor} k_{i-1},$$
$$k_i = H(k_{-1}|k_{i-1}|c[(i-1)b \dots ib]), \quad i = 1, 2, \dots$$

Необходимо отметить, что на настоящее время криптография является весьма развитой областью знаний с большим объемом накопленных теоретических результатов и практики их использования. Поэтому рекомендуется использовать хорошо изученные популярные методы, а не изобретать свои. Множество связанных с этим вопросов заведомо выходит за пределы данной книги, рассмотренный в данном разделе алгоритм шифрования носит иллюстративный характер.

В качестве хэш-функции предлагается алгоритм BLAKE2b[7] с шириной хэша 512 бит ($b = 512$).

Можно написать следующую обертку вызова:

```
#include "blake2b.h"
char key_buf[64];   // 512-бит, сюда пишем хэш.
void hash(char const in[], size_t bytes) {
  blake2b(key_buf, sizeof(key_buf),
```

[7] См.: *Saarinen M.-J., Aumasson J.-P.* The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC) // IETF : [site]. URL: `https://tools.ietf.org/html/rfc7693` (accessed: 29.07.2020). Реализация алгоритма на языке C, выполненная М.-Ю. Саариненом, доступна по URL: `https://github.com/mjosaarinen/blake2_mjosref`.

```
    nullptr, 0, in, bytes);
}
```

Итак, требуется указать: входной файл (параметр –I), выходной файл (параметр –O), режим (зашифровка –e или расшифровка –d), файл с прообразом вектора инициализации (параметр –V), файл с паролем (параметр –P). Если какой-либо из последних двух файлов не указан, то прочитать данные из потока ввода (две полных строки, сначала строку, хэш которой станет вектором инициализации, затем — пароль).

Задание 3

Реализуйте описанный алгоритм шифрования. Проверьте работоспособность программы. Проверьте зависимость шифротекста от вектора инициализации и пароля.

11

Линейные алгоритмы I

Цели самостоятельной работы

- Приобретение навыков реализации простейших линейных операций над последовательностями, простейшего тестирования, использования статических массивов.

- В каждом варианте указано, **что** требуется получить как результат обработки последовательности. При этом предполагается, что последовательность (в массиве или потоке ввода) состоит из чисел с плавающей точкой и может быть любой длины.

Критерии полноты

1. Написать функцию, вычисляющую требуемый результат, обрабатывая как последовательность произвольный массив (функция должна принимать адрес и размер массива либо диапазон, заданный парой указателей).

2. Написать функцию, возвращающую код ошибки или булевское значение (успех или неуспех проверки), выполняющую тестирование функции из п. 1 на не менее чем двух массивах, заданных непосредственно в ней (т. е. предпо-

лагаемый результат работы можно посчитать «вручную» заранее).

3. Написать функцию, вычисляющую требуемый результат, читая числа последовательности произвольной длины (и не храня их все в памяти) со стандартного потока ввода до первой ошибки или конца ввода.

4. Написать программу, выводящую результат теста из п. 2 и позволяющую пользователю ввести последовательность и получить результат работы функции из п. 3.

11.1. Примеры

Отношение числа положительных элементов к числу отрицательных

Задание. Дана последовательность, требуется вычислить отношение числа положительных элементов в последовательности к числу отрицательных элементов. Поведение для пустой последовательности не определено.

Пример: $\{4, 1, 100, 0, 20, 0, -1, -2, 3, -6\} \to 5/3 = 1.(6)$.

При прохождении последовательности достаточно отслеживать количество встреченных положительных чисел и количество встреченных отрицательных чисел. Для этого следует завести две целочисленных переменных. Когда последовательность кончится, вернуть их отношение.

Обратите внимание на явное преобразование типа в возвращаемом выражении (инструкция **return**) — целочисленное деление здесь не годится.

Пример 11.1. Решение п. 1

```
// Массив передаем как адрес + размер.
double pn_ratio(
    double const numbers[], size_t n) {
  // Счетчики положительных и отрицательных чисел.
```

```
  size_t positives = 0, negatives = 0;
  for (size_t i = 0; i < n; ++i) {
    negatives += numbers[i] < 0;
    positives += numbers[i] > 0;
  }
  return double(positives) / negatives;
}
```

Запись

```
negatives += numbers[i] < 0;
positives += numbers[i] > 0;
```

может показаться странной, однако она полностью корректна, и компиляторы для производительных процессоров будут порождать эффективный машинный код для такой конструкции с большей вероятностью, чем для ветвления.

То же действие, записанное через ветвление:

```
double const n = numbers[i];
if (n < 0)
  ++negatives;
else if (n > 0)
  ++positives;
```

Чтобы написать тест, проверяющий функцию pn_ratio, создадим локальный статический массив и передадим его в функцию. Сравним ее ответ с ответом, заранее вычисленным нами. В тесте ниже это выполнено дважды для разных массивов.

Пример 11.2. Решение п. 2

```
bool test_pn_ratio() {
  double const test1[]
    { 4, 1, 100, 0, 20, 0, -1, -2, 3, -6 };
  if (pn_ratio(test1, size(test1)) != 5./3)
    return false;

  double const test2[]
    { -40, -2, -111, 42, 0, 0, 2, -1000, -4 };
  if (pn_ratio(test2, size(test2)) != 2./5)
    return false;
```

```
  // Все проверки прошли успешно.
  return true;
}
```

Алгоритм, обрабатывающий последовательность, считываемую по одному элементу из потока ввода, по существу не отличается от алгоритма, обрабатывающего массив. Вместо текущего индекса есть текущий (последний считанный) элемент и состояние потока.

Тип istream — стандартный тип, к которому также принадлежит и объект стандартного потока ввода cin. Использование ссылки in позволяет вызвать функцию для любого потока ввода, что может быть удобно, в частности, в целях тестирования.

Пример 11.3. Решение п. 3

```
// Считывание чисел с потока ввода.
double pn_ratio(istream & in) {
  size_t positives = 0, negatives = 0;
  for (double x; in >> x;) {
    negatives += x < 0;
    positives += x > 0;
  }
  return double(positives) / negatives;
}
```

Структура программы в целом:

Пример 11.4. Решение, общий вид

```
#include <iostream>
#include <iterator> // std::size
using namespace std;

// П. 1.
double pn_ratio(double const numbers[], size_t n);
// П. 2.
bool test_pn_ratio();
// П. 3.
double pn_ratio(istream & in);
```

```
int main() {
  // Протестировать вариант для массива.
  cout << test_pn_ratio() << endl;
  // Запустить вариант для потока ввода.
  double const result = pn_ratio(cin);
  cout << "\nResult: " << result << endl;
  return EXIT_SUCCESS;
}
```

Евклидова норма вектора

Задание. Вычислить евклидову норму последовательности как многомерного вектора. Евклидова норма вектора определяется как корень квадратный из суммы квадратов компонент вектора.

Пример: $\{1, -5, 2, 20, -13\} \to \sqrt{1 + 25 + 4 + 400 + 169} \approx$ ≈ 24.4744765.

Предполагается простой алгоритм: накопить сумму квадратов в отдельной переменной и вернуть ее квадратный корень.

Пример 11.5. Решение п. 1

```
// Массив передаем как адрес + размер.
double euclid_norm(double const a[], size_t n) {
  double s = 0.0;
  for (size_t i = 0; i < n; ++i)
    s += a[i] * a[i];
  return sqrt(s);
}
```

Вариант для работы с потоком ввода полностью аналогичен варианту для массива.

Пример 11.6. Решение п. 3

```
double euclid_norm(istream & in) {
  double s = 0.0;
  for (double x; in >> x;)
```

```
    s += x * x;
  return sqrt(s);
}
```

Тестирование сравнивает результат функции с приближенно вычисленным результатом, поэтому введем вспомогательные конструкции: точность сравнения (допустимая погрешность) — константа TOLERANCE и функция сравнения на равенство с заданной точностью almost_equal.

Пример 11.7. Решение п. 2

```
// Допустимая погрешность.
double const TOLERANCE = 1e-6;

// Сравнение на равенство двух значений
// с относительной погрешностью tolerance.
// В данном примере используется при тестировании.
bool almost_equal(
  double x1, double x2,
  double tolerance = TOLERANCE) {
  return abs(x1 - x2) <=
    tolerance * fmax(abs(x1), abs(x2));
}

// П. 2: тестирование функции из п. 1.
bool test_euclid_norm() {
  double const test1[]
    { 4, 1, 0, 3, 0, -1, -2, -6 };
  if (!almost_equal(
        euclid_norm(test1, size(test1)),
        8.18535277))
    return false;
  double const test2[] { -40, -2, -1000, -4 };
  if (!almost_equal(
        euclid_norm(test2, size(test2)),
        1000.8096722))
    return false;
  // Все тесты прошли успешно.
  return true;
}
```

Общая структура программы повторяет таковую из предыдущего примера.

Расстояние между последним и первым нулями

Задание. Определить расстояние (в элементах) между первым и последним нулями.

Пример: $\{1, 2, 0, 2, 3, 0, 4, 5, 0, 8\} \to 6$.

Требуется найти позиции первого и последнего нулей и вернуть разность этих позиций. Поиск первого нуля заключается в пропуске ненулевых элементов от начала массива до, собственно, первого нуля. Поиск последнего нуля заключается в прохождении массива до конца и запоминании позиции последнего встреченного нуля. Естественным образом, последнее запомненное значение будет позицией последнего нуля.

Итак, реализуем эти идеи в виде кода.

Пример 11.8. Решение п. 1

```
size_t zero_dist(int const a[], size_t n) {
  size_t i = 0; // позиция первого нуля
  // Пропустить все ненулевые элементы.
  while (i < n && a[i] != 0)
    ++i;
  // i — позиция первого нуля
  // (или i == n, если нулей нет).
  size_t j = i; // позиция последнего нуля
  // Запоминать позицию каждого следующего нуля.
  for (size_t k = i+1; k < n; ++k)
    if (a[k] == 0)
      j = k;
  // Результат — разность позиций.
  return j - i;
}
```

Данный алгоритм прозрачно переносится на поток ввода.

Пример 11.9. Решение п. 3

```
size_t zero_dist(std::istream & in) {
```

```
  size_t i = 0;
  // Пропустить все ненулевые элементы.
  for (int x; in >> x && x != 0;)
    ++i;
  // i — позиция первого нуля (или поток кончился).
  size_t j = i, k = i + 1;
  // Запоминать позицию каждого следующего нуля.
  for (int x; in >> x; ++k)
    if (x == 0)
      j = k;
  // Результат — разность позиций.
  return j - i;
}
```

У Упростите код функции из примера 11.9 (использовать на одну переменную меньше).

Общая структура программы повторяет таковую из предыдущих примеров. Осталось реализовать п. 2 — тестирование варианта для массива.

Пример 11.10. Решение п. 2

```
bool test_zero_dist() {
  int const test1[]
    { 1, 2, 0, 2, 3, 0, 4, 5, 0, 8 };
  if (zero_dist(test1, size(test1)) != 6)
    return false;

  int const test2[]
    { 1, 2, 3, 4, 5 };
  if (zero_dist(test2, size(test2)) != 0)
    return false;

  int const test3[]
    { 0, 1, 1, 1, 1, 0 };
  if (zero_dist(test3, size(test3) - 1) != 0)
    return false;
  if (zero_dist(test3, size(test3)) != 5)
    return false;
  return true;
}
```

Обратите внимание на состав тестов. Желательно сделать побольше разных тестов для охвата разных, в том числе, маловероятных ситуаций. Пусть искомые элементы будут в середине, в самом начале, в самом конце или вообще будут отсутствовать.

Длина последнего участка из неотрицательных чисел

Задание. Определить длину последнего участка, состоящего из неотрицательных чисел.

Пример: $\{1, 2, 3, -1, 0, 0, 1, 4, -5, -5\} \to 4$.

При выполнении п. 1 имеем очевидное решение — искать последний участок с конца массива. Для этого следует найти первое неотрицательное число с конца — оно будет последним неотрицательным числом в массиве и, следовательно, замыкающим последний участок, состоящий из неотрицательных чисел. Далее остается посчитать неотрицательные числа после найденного (в направлении от конца к началу).

Таким образом, решение состоит из двух циклов, стоящих друг за другом.

Пример 11.11. Решение п. 1(а)

```
// Движение по массиву назад.
size_t last_nonnegative_span_backward
    (int const a[], size_t n) {
  // Пропустить все отрицательные с конца.
  while (n-- != 0 && a[n] < 0) {}
  if (n + 1 == 0)
    return 0;
  // Посчитать неотрицательные.
  size_t len = 1;
  while (n-- != 0 && a[n] >= 0)
    ++len;
  return len;
}
```

Данный алгоритм прост, но его нельзя перенести на поток ввода, поскольку поток ввода мы читаем только в одном направлении — от начала к концу. Поэтому напишем еще один вариант (в аналогичных заданиях делать это необязательно — достаточно какой-то одной работающей реализации п. 1), просматривающий массив от начала к концу.

Будем запоминать длину каждого пройденного участка, состоящего из неотрицательных чисел. Естественным образом, эта длина в конце будет длиной последнего такого участка. Для того чтобы сделать это, будем считать неотрицательные числа счетчиком cur, а когда встретится отрицательное число, запомним длину пройденного участка в переменную len и обнулим cur.

Пример 11.12. Решение п. 1(б)

```
// Движение по массиву вперед (менее эффективно,
// но алгоритм совпадает с алгоритмом для п. 3).
size_t last_nonnegative_span_forward
    (int const a[], size_t n) {
  size_t len = 0, cur = 0;
  for (size_t i = 0; i < n; ++i) {
    if (a[i] < 0) {
      // Если здесь закончился очередной участок:
      if (cur > 0) {
        len = cur;
        cur = 0;
      }
    }
    else {
      ++cur;
    }
  }
  // Если в конце неотрицательный участок:
  if (cur != 0) len = cur;
  return len;
}
```

Этот вариант уже легко «портировать» на поток ввода.

Пример 11.13. Решение п. 3

```
size_t last_nonnegative_span
    (std::istream & in) {
  size_t len = 0, cur = 0;
  for (int x; in >> x;) {
    if (x < 0) {
      if (cur > 0) {
        len = cur;
        cur = 0;
      }
    }
    else {
      ++cur;
    }
  }
  if (cur != 0)
    len = cur;
  return len;
}
```

Общая структура программы повторяет таковую из предыдущих примеров. Осталось реализовать п. 2 — тестирование варианта для массива. Используя **for** для диапазона, можно протестировать оба варианта реализации (в своей работе так делать необязательно).

Пример 11.14. Решение п. 2

```
bool test_last_nonnegative_span() {
  for (auto tested : // проверим оба варианта
    { &last_nonnegative_span_forward,
      &last_nonnegative_span_backward }) {
    int const test1[]
      { 1, 2, 3, -1, 0, 0, 1, 4, -5, -5 };
    if (tested(test1, size(test1)) != 4)
      return false;
    int const test2[]
      { -1, -1, -1 };
    if (tested(test2, size(test2)) != 0)
      return false;
    int const test3[]
```

```
      { 1, 1, 1, 1, 1 };
    if (tested(test3, size(test3)) != 5)
      return false;
  }
  return true;
}
```

11.2. Варианты заданий

Вариант 1. Арифметическое среднее элементов последовательности (сумму всех элементов поделить на их количество).

Пример: $\{1, 5, 10, -4, 8, 8, 0\} \to (1+5+10-4+8+8+0)/7 = 4$.

Вариант 2. Размах последовательности (разность между максимумом и минимумом). Размах пустой последовательности равен нулю.

Пример: $\{1, 10, -12, 4, 12, 14, 0\} \to$ размах $= 26$ (поскольку максимум $= 14$, минимум $= -12$).

Вариант 3. Разность между суммой всех четных и суммой всех нечетных чисел в последовательности.

Пример: $\{10, 4, 5, 0, -15, 2, 7, -5, 0\} \to 24 = (10+4+0+2+ + 0) - (5 - 15 + 7 - 5)$.

Вариант 4. Геометрическое среднее элементов последовательности (произведение всех элементов, из которого извлекли корень степени, равной количеству элементов).

Пример: $\{9, 9, 10, 24, 25, 65\} \to (9 \cdot 9 \cdot 10 \cdot 24 \cdot 25 \cdot 65)^{1/6} \approx$ ≈ 17.779721.

Вариант 5. Дана последовательность дробных чисел. Определить, сколько из них являются целыми числами.

Пример: $\{0.0, -1.5, 18.0, 22.001, -1.0\} \to 0.0$, 18.0 и -1.0 — целые $\to 3$.

Вариант 6. Разность между частным суммы модулей элементов последовательности и максимума из модулей элементов последовательности и единицей.

Пример: $\{4, 10, -12, 4, 12, 14, 0\} \to (4+10+12+4+12+14+ +0)/14 - 1 = 3$.

Вариант 7. Геометрическое среднее элементов последовательности через арифметическое среднее логарифмов как e в степени арифметическое среднее логарифмов элементов.

Пример: $\{9, 9, 10, 24, 25, 65\} \to (9 \cdot 9 \cdot 10 \cdot 24 \cdot 25 \cdot 65)^{1/6} = = \exp((\ln 9 + \ln 9 + \ln 10 + \ln 24 + \ln 25 + \ln 65)/6) \approx \approx \exp(2.8780585) \approx 17.779721$.

Вариант 8. Определить, сколько раз в последовательности встречаются пары соседних элементов, равных нулю.

Пример: $\{1, 0, 0, 0, 10, 0, 12, 0, 0\} \to 3$.

Вариант 9. Найти максимальную по модулю разность соседних элементов последовательности. Для пустой последовательности возвращать 0.

Пример: $\{1, 10, 8, 1, 0, 10, 4\} \to \max\{9, 2, 7, 1, 10, 6\} = 10$.

Вариант 10. Определить, сколько раз в последовательности целых чисел встречаются пары соседних элементов, четность которых совпадает (т. е. четное-четное или нечетное-нечетное).

Пример: $\{1, -4, 2, 3, 4, 4, 3, 1, 1, 5, -5\} \to -4$ и 2, 4 и 4, 3 и 1, 1 и 1, 1 и 5, 5 и $-5 \to 6$.

Вариант 11. В последовательности найти разность между наименьшим положительным числом и наибольшим отрицательным числом.

Пример: $\{8, -10, -4, -14, 5, 15, 2, -6, 3, -3\} \to (2) - (-3) \to \to 5$.

Вариант 12. В последовательности найти сумму наибольшего положительного числа и наименьшего отрицательного числа. В случае отсутствия положительных или отрицательных чисел брать сумму с нулем.

Пример: $\{8, -10, -4, -14, 15, 2, -6, -3\} \to 15 + (-14) \to 1$.

Вариант 13. Предполагая, что последовательность содержит единственные минимум и максимум, найти расстояние в позициях между минимумом и максимумом. Для пустой последовательности возвращать нуль.

Пример: $\{4, 1, 0, 5, 4, 2, 1\} \to$ минимум $= 0$, позиция 2; максимум $= 5$, позиция $3 \to$ расстояние между ними $= 1$.

Вариант 14. Определить длину последнего участка последовательности, состоящего из идущих подряд нулей.

Пример: $\{0, 0, 0, 1, 2, 3, 4, 0, 0, 5, 6, 7\} \to 2$.

Вариант 15. Определить длину последнего участка последовательности (не короче двух элементов), содержащего строго возрастающую подпоследовательность.

Пример: $\{1, 2, 3, 4, 4, 1, 0, -1, 2, 3, -2, -2\} \to 3$.

Вариант 16. Определить длину последнего участка последовательности (не короче двух элементов), содержащего пилообразную подпоследовательность.

Пример: $\{1, 2, 3, 1, 4, 1, 5, -1, 2, 3, -2, 1\} \to 4$ (соответствует участку $2, 3, -2, 1$).

12

Случайный лабиринт

Цели лабораторной работы

- Изучить конструкцию «двумерный массив».
- Реализовать рекурсивный алгоритм.

Задание

Рассмотрим задачу генерации прямоугольного лабиринта на сетке квадратных ячеек. Ячейка может находиться в двух состояниях: проходимом и непроходимом. Сам лабиринт в таком случае может быть представлен в виде двумерного булевского массива.

```
// Размеры лабиринта.
int const MAZE_WIDTH = 79;
int const MAZE_HEIGHT = 23;

// Сам лабиринт в виде булевского массива.
bool maze[MAZE_HEIGHT][MAZE_WIDTH];
```

Обратите внимание на адресацию — она аналогична той, что принята в матрицах: первая координата — номер строки (смещение вниз по вертикали), вторая — номер столбца (смещение вправо по горизонтали).

Цикл работы программы следующий:

- Сгенерировать лабиринт (заполнить maze соответствующим образом).
- Вывести результат на экран.
- Если пользователь нажимает Enter — повторить сначала. Если пользователь ввел что-то другое — выйти.

Это может быть записано на C++ следующим образом:

```
int main() {
  void show_maze();
  void generate_maze();

  // Повторять, пока пользователь нажимает Enter.
  do {
    generate_maze();
    show_maze();
  } while (cin.get() == '\n');
  return 0;
}
```

Рассмотрим реализацию.

Функцию вывода на экран можно построить на основе стандартного текстового вывода в консоль:

```
void show_maze() {
  // Представление ячейки.
  static char const cell_repr[] { '␣', '#' };
  // Вывести...
}
```

Вывод можно представить в виде двойного цикла (по строкам и затем по столбцам), выводящего тот или иной символ из cell_repr. Данные циклы можно построить как простые циклы со счетчиком или как циклы по диапазонам:

```
void show_maze() {
  // Представление ячейки.
  static char const cell_repr[] { '␣', '#' };
```

```
  // Вывести.
  for (auto & row: maze) {
    for (auto cell: row)
      cout.put(cell_repr[cell]);
    cout << '\n';
  }
}
```

В целях тестирования можно написать простую функцию генерации лабиринта. Например, такую:

```
void generate_maze() {
  for (int y = 0; y < MAZE_HEIGHT; ++y)
    for (int x = 0; x < MAZE_WIDTH; ++x)
      maze[y][x] = ((x^y)/8) & 1;
}
```

Что даст «шахматную доску» с клетками размера 8×8 ячеек.

Для создания случайного (псевдослучайного) лабиринта нужен генератор (псевдо)случайных чисел. Ниже дан код на основе стандартной библиотеки C++ (а именно, заголовочного файла **random**):

```
// Следующее (псевдо)случайное число.
size_t random() {
  // Генератор типа «вихрь Мерсенна».
  thread_local mt19937_64 rng {
    // Попробовать сгенерировать случайное зерно.
    random_device{}()
  };
  // Получить следующий элемент последовательности.
  return rng();
}
```

Если теперь в функции generate_maze заменить предпоследнюю строчку на

```
      maze[y][x] = random() & 1;
```

то можно убедиться в работоспособности функции random.

Теперь можно приступить к выполнению главной задачи. Ниже предлагается конкретный метод ее решения, результатом

которого должен быть лабиринт, в котором существует единственный путь, соединяющий любые две проходимые ячейки.

Вначале сделаем лабиринт полностью непроходимым:

```
for (auto & row: maze)
  for (auto & cell: row)
    cell = true;
```

Начиная с произвольной клетки с нечетными координатами, мы можем «пробивать путь» (двигаясь **только** по непроходимым ячейкам и превращая их в проходимые), случайно выбирая любое из четырех направлений (вправо, влево, вверх, вниз). Естественно, мы не можем пересекать границы лабиринта. Если идти некуда — завершаем работу.

Шагать требуется сразу на две ячейки: по одной ячейке на стенку и на «коридор».

Итак, алгоритм («сходить с ячейки X, Y») можно условно записать так:

- Начать с ячейки X, Y.
- Сделать ячейку X, Y проходимой.
- Перебирать в случайном порядке направления: вправо, влево, вверх, вниз:
  - Пусть сдвиг по направлению задается приращениями координат dx, dy.
  - Если допустимо двигаться в ячейку с координатами X+2*dx, Y+2*dy:
    - Сделать ячейку X+dx, Y+dy проходимой.
    - Сходить с ячейки X+2*dx, Y+2*dy.

Итак, требуется реализовать вышеописанный алгоритм и убедиться в его работоспособности.

Пример работы программы представлен на рис. 12.1.

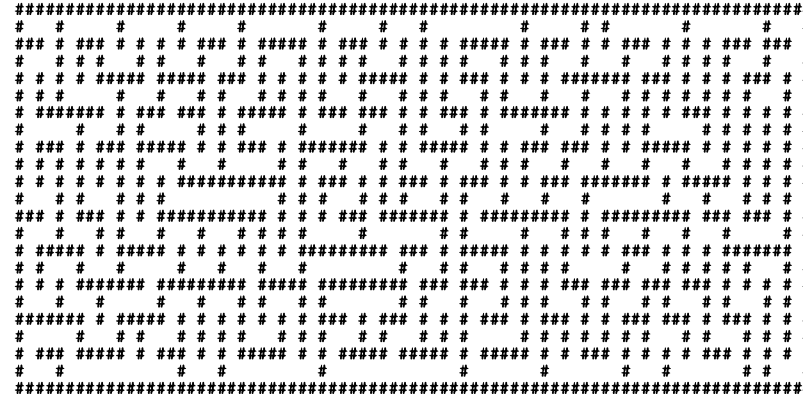

Рис. 12.1. Пример результата работы готовой программы

13

Матрицы

Цель самостоятельной работы

- Закрепление навыков элементарной алгоритмизации и работы с двумерными массивами.

Критерии полноты

1. Должна быть реализована отдельная функция, выполняющая действие, указанное в задании.

2. Данная функция должна принимать произвольную (необязательно квадратную) матрицу, не копируя ее. Элементы матрицы, как правило, являются числами с плавающей запятой.

3. В виде отдельной функции должны быть реализованы по два автоматических теста на «положительный» и «отрицательный» ожидаемые ответы функции. Матрицы в тестах должны быть разных размеров.

13.1. Примеры

Класс матрицы

Вначале дадим исходный код класса Matrix, которым можно пользоваться при решении заданий этой работы.

Пример 13.1. Matrix

```
#include <utility>
#include <memory>
#include <ostream>
#include <initializer_list>
// Объект «матрица».
template <typename ET = double>
class Matrix {
public:
  // Тип элементов.
  using value_type = ET;
  // Конструктор по умолчанию (пустая матрица).
  Matrix(): _rows(0), _cols(0) {}
  // Конструктор матрицы заданных размеров.
  Matrix(int rows, int cols)
      : _rows(rows), _cols(cols),
        _data(std::make_unique<ET[]>(
                std::size_t(rows) * cols)) {}
  // Конструктор из списка-инициализатора.
  template <typename Scalar>
  Matrix(std::initializer_list<
          std::initializer_list<Scalar>> il)
      : Matrix() {
    auto const rows = int(il.size());
    int cols = 0;
    for (auto & row: il) {
      auto const row_cols = int(row.size());
      cols = cols < row_cols? row_cols: cols;
    }

    if (rows > 0 && cols > 0) {
      Matrix m(rows, cols);
      m.fill(ET{});
```

```
    auto il_p = il.begin();
    auto row = m.data();
    for (int r = 0; r < rows;
         ++r, ++il_p, row += cols) {
      int i = 0;
      for (auto el: *il_p)
        row[i++] = el;
    }
    swap(m);
  }
}

// Перемещающий конструктор.
Matrix(Matrix && other)
    : _rows(other.rows()), _cols(other.cols()),
      _data(std::move(other._data)) {
  other._rows = other._cols = 0;
}
// Перемещающий оператор присваивания.
Matrix & operator=(Matrix && other) {
  if (this != &other) {
    // Через перемещающий конструктор.
    Matrix temp(std::move(other));
    swap(temp);
  }
  return *this;
}

// Копирующий конструктор.
Matrix(Matrix const & other)
    : Matrix(other.rows(), other.cols()) {
  // Скопировать элементы.
  auto const s = other.data();
  auto const d = data();
  for (std::size_t i = 0; i < size(); ++i)
    d[i] = s[i];
}
// Копирующий оператор присваивания.
Matrix & operator=(Matrix const & other) {
  Matrix(other).swap(*this);
```

```
    return *this;
  }

  // Заполнить матрицу константой.
  void fill(ET val) {
    // Так как все элементы одинаковые и идут подряд, то
    // можно организовать обход как линейного массива.
    auto const sz = size();
    auto const p = data();
    for (size_t i = 0; i < sz; ++i)
      p[i] = val;
  }

  // Проверка на пустоту.
  bool empty() const { return _data == nullptr; }
  // Сколько строк?
  int rows() const { return _rows; }
  // Сколько столбцов?
  int cols() const { return _cols; }

  // Обратиться к строке матрицы.
  ET * operator[](int row)
  { return data() + row*_cols; }
  ET const * operator[](int row) const
  { return data() + row*_cols; }

  // Сколько всего элементов в массиве?
  std::size_t size() const
  { return std::size_t(rows()) * cols(); }

  // Прямой доступ к массиву.
  ET * data() { return _data.get(); }
  ET const * data() const { return _data.get(); }

  // Обменять содержимое двух матриц.
  void swap(Matrix & other) {
    std::swap(_rows, other._rows);
    std::swap(_cols, other._cols);
    _data.swap(other._data);
  }
```

```
private:
  // Данные объекта.
  int _rows, _cols;
  std::unique_ptr<ET[]> _data;
};
// Сравнение матриц на равенство.
template <typename T1, typename T2>
bool operator==
    (Matrix<T1> const & a, Matrix<T2> const & b) {
  auto const rows = a.rows(), cols = a.cols();
  if (rows != b.rows() || cols != b.cols())
    return false;
  auto const sz = a.size();
  auto const p = a.data();
  auto const q = b.data();
  for (size_t i = 0; i < sz; ++i)
    if (p[i] != q[i])
      return false;
  return true;
}

// Сравнение матриц на неравенство.
template <typename T1, typename T2>
inline bool operator!=
    (Matrix<T1> const & a, Matrix<T2> const & b)
{ return !(a == b); }

// Вывести матрицу в поток вывода.
template <typename ET>
std::ostream & operator<<
    (std::ostream & os, Matrix<ET> const & mtx) {
  auto const rows = mtx.rows(), cols = mtx.cols();
  auto m = mtx.data();
  for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < cols; ++col)
      os << ' ' << m[col];
    os << '\n';
    m += cols; // к следующей строке
  }
```

```
  return os;
}

// Матрица целых.
using Matrixi = Matrix<int>;
// Матрица float.
using Matrixf = Matrix<float>;
// Матрица double.
using Matrixd = Matrix<double>;
// Матрица bool.
using Matrixb = Matrix<bool>;
```

Объекты данного класса можно инициализировать аналогично двумерному статическому массиву из списка строк, каждая из которых задана списком чисел. Реализация данной возможности использует стандартный компонент под названием initializer_list, рассматривать который в данный момент мы не будем.

В принципе, уже функции заполнения, копирования, сравнения матриц на равенство, создания единичной матрицы можно считать примерами. Ниже даны еще примеры.

Проверка матрицы на единичность

Задание. Убедиться, что матрица с заданной точностью является единичной. Единичная матрица — это квадратная матрица, на главной диагонали которой стоят нули, а прочие элементы — единицы.

Предположим, что элементы матрицы — числа с плавающей запятой. Тогда с «заданной точностью» будет означать, что числа $a \approx b$, если $|a - b| \leqslant \varepsilon_{abs}$, где ε_{abs} и есть та самая заданная точность (допустимая абсолютная погрешность).

Решение. Введем вспомогательную функцию для проверки на близость (примерное равенство) чисел a и b:

```
bool are_almost_equal(float a, float b, float eps) {
  return fabs(a - b) <= eps;
}
```

Воспользуемся определенным выше типом Matrixf. Функция «проверки на единичность» может быть записана так:

```
bool is_identity(Matrixf const & m, float eps = 0) {
  int const rows = m.rows(), cols = m.cols();
  // Проверить размеры.
  if (rows != cols)
    return false;
  // Проверить элементы.
  for (int i = 0; i < rows; ++i)
    for (int j = 0; j < cols; ++j)
      if (!are_almost_equal(m[i][j], i==j, eps))
        return false;
  return true;
}
```

В первую очередь проверим, что матрица действительно квадратная (иначе сразу вернем ложь). Затем мы проходим матрицу целиком, перебирая все строки (i) и столбцы (j), и убеждаемся, что элемент-ij приближенно равен (i==j), что есть единица для равных i и j и нуль в противном случае. В конце мы возвращаем истину, поскольку тот факт, что исполнение дошло до строчки

```
  return true;
```

означает, что все проверки прошли успешно, и наша матрица является единичной.

Теперь напишем тест. Тест возвращает номер ошибки или 0 в случае успешного прохождения:

```
int test_is_identity() {
  Matrixf t1 {
    { 0, 0, 0 },
    { 0, 1, 0 },
    { 0, 0, 1 },
  };

  // Это не единичная матрица.
  if (is_identity(t1))
    return 1;
```

```
  t1[0][0] = 1;
  // А теперь — единичная.
  if (!is_identity(t1))
    return 2;
  t1[1][1] += 0.01f;
  // Не единичная, если погрешность == 0.
  if (is_identity(t1))
    return 3;
  // Единичная в пределах погрешности 0.02.
  if (!is_identity(t1, 0.02f))
    return 4;
  Matrixf t2 {
    { 1, 0, 0 },
    { 0, 1, 0 },
  };
  // Не квадратная, значит — не единичная.
  if (is_identity(t2))
    return 5;
  // Все тесты прошли успешно.
  return 0;
}
```

Нулевая часть матрицы

Задание. «Квадратной частью матрицы» будем называть квадратную подматрицу, верхний левый угол которой совпадает с верхним левым углом матрицы, а размер равен минимуму из числа строк и числа столбцов матрицы.

Если матрица квадратна, то ее квадратная часть совпадает с ней самой. Если в матрице столбцов больше, чем строк, то ее можно представить как ее квадратную часть и набор дополнительных столбцов. Если же в матрице строк больше, чем столбцов, то ее можно представить как ее квадратную часть и набор дополнительных строк.

Задача состоит в определении того, являются ли все элементы матрицы вне ее квадратной части нулевыми. Для квадратной матрицы положим это истиной.

Решение. Итак, есть три случая:

1. Матрица квадратная. В этом случае возвращаем истину.

2. Матрица содержит больше строк, чем столбцов. Тогда нужно проверить, что она имеет следующий вид:

$$\begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix}$$

3. Матрица содержит больше столбцов, чем строк. Тогда нужно проверить, что она имеет следующий вид:

$$\begin{pmatrix} a_{11} & \cdots & a_{1m} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} & 0 & \cdots & 0 \end{pmatrix}$$

Выполним функциональную декомпозицию, выделив проверки двух последних случаев в отдельные функции. Например, так:

```
bool are_rows_zero
  (Matrix1 const & m, int from, int to);
bool are_cols_zero
  (Matrix1 const & m, int from, int to);
bool are_outer_zeroes(Matrix1 const & m) {
  auto const rows = m.rows(), cols = m.cols();
  if (rows == cols)
    return true;
  if (rows < cols)
    return are_cols_zero(m, rows, cols);
  return are_rows_zero(m, cols, rows);
}
```

Для реализации функций, проверяющих «лишние» строки и столбцы, нам пригодится еще одна вспомогательная функция, проверяющая, что массив чисел состоит из нулей:

```
bool only_zeroes(int const a[], int n) {
  for (int i = 0; i < n; ++i)
    if (a[i] != 0)
      return false;
  return true;
}
```

Теперь очень легко реализовать проверку заданного диапазона строк: если какая-либо из строк не состоит только лишь из нулей, то возвращаем ложь, если же все проверки прошли успешно, то возвращаем истину:

```
bool are_rows_zero
    (Matrixi const & m, int from, int to) {
  auto const cols = m.cols();
  for (auto row = m[from]; from++ != to; row+=cols)
    if (!only_zeroes(row, cols))
      return false;
  return true;
}
```

Ту же функцию only_zeroes можно применить для реализации проверки столбцов. Ведь диапазон нулевых столбцов соответствует нулевому диапазону в каждой строке матрицы:

```
bool are_cols_zero
    (Matrixi const & m, int from, int to) {
  auto const rows = m.rows(), cols = m.cols();
  auto const n = to - from;
  auto row = m.data();
  for (int i = 0; i < rows; ++i, row += cols)
    if (!only_zeroes(row + from, n))
      return false;
  return true;
}
```

Далее остается написать тест. (Его текст здесь приводить не будем.)

13.2. Варианты заданий

Вариант 1. Диагональная матрица — квадратная матрица (a_{ij}) такая, что $a_{ij} = 0$ при $i \neq j$.

Пример

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Определить, является ли заданная матрица диагональной.

Вариант 2. Симметричная матрица — квадратная матрица (a_{ij}) такая, что $a_{ij} = a_{ji}$ при всех возможных значениях индексов i и j.

Пример

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 6 & 1 \\ 3 & 6 & 0 & 2 \\ 4 & 1 & 2 & 7 \end{pmatrix}$$

Определить, является ли заданная матрица симметричной.

Вариант 3. Антидиагональная матрица — $n \times n$-матрица (a_{ij}) такая, что $a_{ij} = 0$ при любых $i + j \neq n$, если начинать отсчитывать индексы с нуля. Ненулевыми могут быть только элементы на побочной диагонали.

Пример

$$\begin{pmatrix} 0 & 0 & 0 & 4 \\ 0 & 0 & 8 & 0 \\ 0 & -1 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{pmatrix}$$

Определить, является ли заданная матрица антидиагональной.

Вариант 4. Верхнетреугольная матрица — квадратная матрица (a_{ij}) такая, что $a_{ij} = 0$ при $i > j$.

Пример

$$\begin{pmatrix} 1 & -2 & 7 & 1 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Определить, является ли заданная матрица верхнетреугольной.

Вариант 5. Кососимметричная матрица — квадратная матрица (a_{ij}) такая, что $a_{ij} = -a_{ji}$ при всех возможных значениях индексов i и j.

Пример

$$\begin{pmatrix} 0 & -2 & -3 & -4 \\ 2 & 0 & 6 & -1 \\ 3 & -6 & 0 & -2 \\ 4 & 1 & 2 & 0 \end{pmatrix}$$

Определить, является ли заданная матрица кососимметричной.

Вариант 6. Нижнетреугольная матрица — квадратная матрица (a_{ij}) такая, что $a_{ij} = 0$ при $i < j$.

Пример

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 5 & 3 & 0 & 0 \\ 1 & -5 & 2 & 0 \\ 1 & -1 & 1 & 1 \end{pmatrix}$$

Определить, является ли заданная матрица нижнетреугольной.

Вариант 7. L-матрица — квадратная матрица, на главной диагонали которой стоят положительные значения, а все прочие элементы не превосходят нуля.

Пример

$$\begin{pmatrix} 1 & 0 & -2 & -5 \\ 0 & 3 & 0 & -3 \\ -1 & -3 & 3 & -1 \\ -1 & -10 & 0 & 1 \end{pmatrix}$$

Определить, является ли заданная матрица L-матрицей.

Вариант 8. Трехдиагональная матрица — квадратная матрица (a_{ij}) такая, что $a_{ij} = 0$ при $|i - j| > 1$.

Пример

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 5 & 3 & 1 & 0 & 0 \\ 0 & 3 & 2 & 1 & 0 \\ 0 & 0 & 3 & 2 & 1 \\ 0 & 0 & 0 & 3 & 1 \end{pmatrix}$$

Определить, является ли заданная матрица трехдиагональной.

Вариант 9. Матрица с диагональным преобладанием — квадратная матрица (a_{ij}) такая, что $2|a_{ii}| > \sum_j |a_{ij}|$ для всех индексов i.

Пример

$$\begin{pmatrix} 20 & 1 & -5 & 2 \\ 5 & 30 & 0 & 3 \\ 1 & -10 & 20 & -1 \\ 1 & -1 & 1 & -5 \end{pmatrix}$$

Определить, является ли заданная матрица матрицей с диагональным преобладанием.

Вариант 10. Персимметричная матрица — $n \times n$-матрица (a_{ij}) такая, что $a_{ij} = a_{n-j,n-i}$ при всех возможных значениях отсчитываемых с нуля индексов i и j.

Пример

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 8 & 0 & 5 & 3 \\ 1 & 6 & 0 & 2 \\ 7 & 1 & 8 & 1 \end{pmatrix}$$

Определить, является ли заданная матрица персимметричной.

Вариант 11. Матрица Теплица — прямоугольная матрица, в которой на всех диагоналях, параллельных главной диагонали, стоят равные элементы.

Пример

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 1 & 2 & 3 & 4 & 5 & 6 \\ 9 & 8 & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Определить, является ли заданная матрица матрицей Теплица.

Вариант 12. Матрица Ганкеля — квадратная матрица, в которой на всех диагоналях, параллельных побочной диагонали, стоят равные элементы.

Пример

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

Определить, является ли заданная матрица матрицей Ганкеля.

Вариант 13. Циркулянт — квадратная матрица, j-й столбец (кроме самого левого) которой задается циклической перестановкой элементов предыдущего столбца на одну позицию вниз.

Пример

$$\begin{pmatrix} 5 & 3 & 7 & 1 \\ 1 & 5 & 3 & 7 \\ 7 & 1 & 5 & 3 \\ 3 & 7 & 1 & 5 \end{pmatrix}$$

Определить, является ли заданная матрица циркулянтом.

Вариант 14. Антициркулянт — квадратная матрица, каждая строка которой (кроме самой верхней) является циклической перестановкой элементов предыдущей строки на одну позицию влево.

Пример

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

Определить, является ли заданная матрица антициркулянтом.

Вариант 15. Определить, сколько нулевых столбцов содержит матрица. Функция должна возвращать целое число (тип size_t).

Замечание. Данный вариант отличается от прочих тем, что требует не проверки некоего условия, а вычисления значения. Реализовать три теста: с матрицей без нулевых столбцов и с двумя матрицами с несколькими нулевыми столбцами (разное количество нулевых столбцов).

Вариант 16. Ведущим элементом ненулевой строки матрицы назовем самый левый ненулевой элемент этой строки. Тогда матрица является **ступенчатой**, если выполняются условия:

- Если в матрице есть и нулевые, и ненулевые строки, то самая верхняя нулевая строка находится под самой нижней ненулевой строкой.
- Для любой пары ненулевых строк верно, что ведущий элемент нижней строки находится строго правее ведущего элемента верхней строки.

Пример

$$\begin{pmatrix} 1 & 5 & 0 & 1 & 1 \\ 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Определить, является ли заданная матрица ступенчатой.

Вариант 17. Ортогональная матрица — квадратная матрица (a_{ij}), для которой верно $a_{i*}a_{j*}^{\top} = a_{*i}^{\top}a_{*j} = [i = j]$. Здесь a_{i*} — i-я строка, а a_{*j} — j-й столбец.

Строки (или столбцы) ортогональной матрицы составляют ортонормированный базис (являются попарно перпендикулярными друг другу векторами единичной длины). Если A ортогональна, то $A^{\top} = A^{-1}$, т. е. должно выполняться условие $AA^{\top} = I$, где I — единичная матрица.

Определить, является ли произвольная матрица ортогональной. Достаточно проверить попарные скалярные произведения всех строк или всех столбцов. Исходя из соображений вычислительной эффективности, следует выбрать, что использовать — строки или столбцы. Следует также учесть, что ввиду ограниченной точности представления чисел точное равенство нулю или единице может не обеспечиваться на практике.

Пример 1

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Пример 2

$$\begin{pmatrix} 0.242536 & 0.970143 & 0 \\ 0.570782 & -0.142695 & 0.808608 \\ -0.784465 & 0.196116 & 0.588348 \end{pmatrix}$$

Точность не хуже 10^{-5}.

14

Линейные алгоритмы II

Цель самостоятельной работы

- Дальнейшая наработка навыков определения линейных алгоритмов, оперирующих последовательностями значений.

Критерии полноты

В каждом варианте указано, что́ требуется получить в качестве результата обработки последовательности. При этом предполагается, что последовательность может быть любой длины. Требуется:

1. Написать функцию, вычисляющую требуемый результат, обрабатывая как последовательность произвольный массив (функция должна принимать либо адрес и размер массива, либо диапазон, заданный парой указателей).

2. Написать функцию, выполняющую тестирование функции из п. 1 на не менее чем трех массивах (если в задании не сказано иначе), заданных непосредственно в этой функции (т. е. предполагаемый результат работы можно определить заранее).

3. Написать функцию, вычисляющую требуемый результат, читая последовательность произвольной длины (и не сохраняя ее в память) с потока ввода до первой ошибки или конца ввода.

4. Написать функцию, выполняющую тестирование функции из п. 3 на не менее чем трех последовательностях (если в задании не сказано иначе), заданных непосредственно в этой функции (т. е. предполагаемый результат работы можно определить заранее).

5. Написать программу, прогоняющую все тесты и выводящую отчет.

Замечание. Тесты из п. 2 и п. 4 могут использовать одинаковые входные данные (и, соответственно, ожидать одинаковые результаты), различаясь только промежуточным представлением данных (массив и поток). Поэтому разумно разделить сами данные для теста и конкретный способ его выполнения (см. примеры).

14.1. Примеры

Максимальная подпоследовательность дубликатов

Задание. Определить длину максимальной подпоследовательности, состоящей из идущих подряд равных элементов — «дубликатов». Здесь *подпоследовательность* — непрерывный участок исходной последовательности.

Пример. $\{1, 2, 3, 3, 0, 0, 0, 1, 2\} \to 3$ (три нуля подряд).

Для того чтобы, проходя по последовательности, иметь возможность определить, кончилась последовательность дубликатов или нет, нам требуется помимо значения текущего элемента знать и значение предыдущего элемента. Соответственно, если текущий элемент совпадает с предыдущим, то мы увеличиваем

значение переменной cur_run, содержащей длину текущей подпоследовательности дубликатов. В противном случае мы записываем в нее 1, поправляя значение переменной max_run, содержащей максимальную из длин подпоследовательностей дубликатов, обнаруженных к данному моменту.

```
// П. 1: обработка произвольного массива,
//  переданного как адрес + размер
size_t max_duprun(double const a[], size_t n) {
  if (n == 0)
    return 0;
  // Массив не пуст.
  size_t max_run = 1; // максимальная длина
  size_t cur_run = 1; // текущая длина
  for (size_t i = 1; i < n; ++i) {
    if (a[i] != a[i - 1])
      cur_run = 1; // соседние не равны
    // идут равные
    else if (max_run < ++cur_run)
      max_run = cur_run;
  }
  return max_run;
}

// П. 3: считывание чисел с потока ввода.
size_t max_duprun(istream & in) {
  size_t max_run = 0; // максимальная длина
  // x, prev_x — два последних числа.
  if (double x, prev_x; in >> prev_x) {
    // Имеем не менее одного элемента.
    max_run = 1;
    size_t cur_run = 1; // текущая длина
    while (in >> x) {
      if (x != prev_x) {
        // Встретили неравные.
        cur_run = 1;
        prev_x = x;
      }
      // Пропускаем равные.
      else if (max_run < ++cur_run) {
```

```
            max_run = cur_run;
        }
      }
    }
    return max_run;
}
```

Учитывая сходство двух вариантов (для массива и потока) различных линейных алгоритмов, будет естественным задать вопрос, зачем вообще второй вариант? Поскольку работать с массивом обычно удобнее, можно считать данные из потока в массив и затем применить к нему вариант для массива. Часто это действительно разумный выбор. Однако цель данного задания заключается также в том, чтобы продемонстрировать аспекты реализации одного и того же алгоритма в разных условиях, лучше разобраться в самом разрабатываемом алгоритме. С практической же точки зрения не всегда возможно сохранять данные в память. Памяти может попросту не хватить.

Чтобы не писать два теста для двух вариантов одного алгоритма, выполним один общий тест в виде функции, которая вызывает *обертку*[8] [*wrapper*] тестируемой функции для заранее заданного массива и результата. Обертка возвращает истину, если результат совпал.

Обертки принимают заранее заготовленные массив и правильный ответ и имеют следующий вид:

```
// П. 2: тестирование функции из п. 1
// на заданном массиве с заданным результатом.
bool test_max_duprun_array(
    double const a[], size_t n,
    size_t result)
{ return result == max_duprun(a, n); }

// П. 4: тестирование функции из п. 3
// на заданном массиве с заданным результатом.
```

[8] Так называют вспомогательный код, который «оборачивает» обращение к какому-то другому коду, делая его использование удобным в некотором контексте.

```
bool test_max_duprun_stream(
    double const a[], size_t n,
    size_t result) {
  stringstream test;
  for (size_t i = 0; i < n; ++i)
    test << a[i] << ' ';
  return result == max_duprun(test);
}
```

Тип указателя на функцию-обертку:

```
using Max_duprun_tester =
bool(*)(double const a[], size_t n, size_t result);
```

Наконец, сама тестирующая функция:

```
// П. 2 и п. 4: тестирование функции из п. 1 или п. 3
// (выбирается через tester).
int test_max_duprun(Max_duprun_tester tester) {
  double const test1[]
    { 1, 1, 2, 2, 2, 2, 0, 4, 2, 2 };
  if (!tester(test1, 0, 0)) return 1;
  if (!tester(test1, 1, 1)) return 2;
  if (!tester(test1, 2, 2)) return 3;
  if (!tester(test1, 5, 3)) return 4;
  if (!tester(test1, size(test1), 4))
      return 5;
  double const test2[]
    { -4, -3, -2, -1 };
  if (!tester(test2, size(test2), 1))
    return 6;
  double const test3[]
    { 1, 2, 3, 3, 0, 0, 0, 1, 2 };
  if (!tester(test3, size(test3), 3))
    return 7;
  // Все тесты пройдены успешно.
  return 0;
}
```

Частоты цифр

Частота символа [*character frequency*] — отношение количества вхождений данного символа (или класса символов) к числу всех символов (или более широкого класса символов). Иногда частотой также называют само количество вхождений.
Пример: «1101110010» — частота 0 равна 0.4, частота 1 равна 0.6, частота цифр равна 1.
Гистограмма [*histogram*] — таблица частот или исходных количеств.

Задание. Определить частоты различных цифр в последовательности символов (байт). Вывести их частоты относительно всех символов, относительно только графических символов, относительно только цифр.

Для того чтобы определять, является ли символ цифрой, будем использовать функцию isdigit, графическим символом — функцию isgraph. Эти функции объявлены в стандартном заголовочном файле **cctype**.

Вспомогательные определения:

- BYTES — количество разных байт (как правило, 256). Для вычисления этой константы воспользуемся стандартной константой CHAR_BIT (число бит в байте), определенной в **climits**:

```
size_t const BYTES = 1u << CHAR_BIT;
```

- Byte — целочисленный тип со значениями в диапазоне от 0 до (BYTES – 1). Это попросту синоним встроенного типа **unsigned char**:

```
using Byte = unsigned char;
```

- Counter — тип счетчика для символа. В качестве этого типа разумно выбрать тип стандартной библиотеки, предназначенный для представления смещений в файлах (и, соответственно, их размеров):

```
using Counter = std::streamoff;
```

- Histogram (гистограмма) — массив счетчиков:

```
using Histogram = Counter[BYTES];
```

Построенная гистограмма содержит всю необходимую информацию о частотах символов. Накопление гистограммы на основе данных в потоке, по сути, не отличается от работы с массивом — надо для каждого символа увеличить на единицу счетчик по индексу, равному коду этого символа.

Приведение прочитанного **char** к Byte необходимо, так как **char** может быть целым числом со знаком и «верхняя половина» диапазона в таком случае попадает в отрицательные индексы, что влечет выход за пределы массива h:

```
// Строит гистограмму байт потока.
// Добавляет количества к счетчикам h.
// Возвращает общее количество.
Counter byte_histogram(
    istream & in, Histogram & h) {
  Counter bytes_read = 0;
  // Читать по символу из потока,
  // увеличивая на каждой итерации
  // соответствующий счетчик в h.
  for (char ch; in.get(ch); ++bytes_read)
    h[Byte(ch)]++;
  return bytes_read;
}

// Строит гистограмму массива.
// Добавляет количества к счетчикам h.
// Возвращает переданный размер.
size_t byte_histogram(
    char const a[], size_t sz,
    Histogram & h) {
  for (size_t i = 0; i < sz; ++i)
    h[Byte(a[i])]++;
  return sz;
}
```

Остальной код уже не зависит от того, что используется в качестве исходного текста — поток или массив. Например, можно посчитать количество символов, удовлетворяющих заданному предикату. В частности, можно задать функцию, принимающую указатель на функцию-классификатор символов вроде стандартных isdigit, isalpha и isgraph:

```
// Тип: указатель на функцию-предикат символа:
using Char_predicate = int(*)(int);

// Возвращает сумму элементов гистограммы,
// для индексов которых предикат filter
// возвращает истину.
Counter histogram_filter_sum(
    Histogram const & h,
    Char_predicate filter) {
  Counter s = 0;
  for (size_t i = 0; i < BYTES; ++i)
    s += filter(int(i))? h[i]: 0; /*
      i приводится к int, так как стандартные функции —
      символьные предикаты принимают int, а не size_t */
  return s;
}
```

Наконец, напишем функцию digits_freqs, вычисляющую относительные частоты цифр. Данная функция будет принимать общее количество total символов некоторого класса, относительного которого вычисляются частоты. Это значение должно быть как-то получено извне этой функции, например, как результат histogram_filter_sum, если требуется вычислить относительные частоты. Или в качестве total можно взять количество всех символов (сумму гистограммы, возвращаемую функцией byte_histogram), если это абсолютные частоты.

```
// Возвращает суммарную частоту.
double digits_freqs(
    Histogram const & h,
    Counter total,
    double freqs[10]) {
  Counter h_digit_sum = 0;
```

```
  auto const divisor = double(total);
  for (int digit = 0; digit < 10; ++digit) {
    auto const h_digit = h[dec_digit(digit)];
    h_digit_sum += h_digit;
    freqs[digit] = h_digit/divisor;
  }
  return h_digit_sum/divisor;
}
```

Вспомогательная функция dec_digit выполняет преобразование *номер десятичной цифры* → *символ цифры*:

```
inline Byte dec_digit(int n) {
  assert(0 <= n && n < 10);
  return "0123456789"[n];
  // или n["0123456789"]
}
```

Однако авторам неизвестны используемые компьютерными системами кодировки символов, в которых коды десятичных цифр идут не подряд[9]. Поэтому можно считать вполне переносимым более простой и вычислительно эффективный «арифметический» вариант:

```
inline Byte dec_digit(int n) {
  assert(0 <= n && n < 10);
  return '0' + n;
}
```

14.2. Варианты заданий

Вариант 1. Вычислить разность между максимумом элементов последовательности, стоящих на нечетных позициях (если считать с единицы), и минимумом элементов, стоящих на четных позициях.

[9] Впрочем, например, в кодировке, используемой IBM 1401, символ нуля стоял после символа девятки. Но подобные кодировки вряд ли где-либо используются в наши дни.

Пример: $\{5, 1, 6, 10, 3, -1, 4, 2\} \to$
$\to \max\{5, 6, 3, 4\} - \min\{1, 10, -1, 2\} \to 6 - (-1) \to 7.$

Вариант 2. Определить количество смен знаков элементов последовательности.

Пример: $\{1, 2, -1, -2, 0, 0, -2, -1, 0, 0, 5, 0\} \to 2$ (нуль не дает смены знака).

Вариант 3. Найти максимум бегущего геометрического среднего последовательности, игнорируя неположительные элементы.

Пример: $\{1, 2, 4, 8, 0, 0, 4, 4, 16, -1\} \to$

(последовательность без неположительных значений)

$\{1, 2, 4, 8, 4, 4, 16\} \to$

(последовательность накопленных произведений)

$\{1, 2, 8, 64, 256, 1024, 16384\} \to$

(последовательность значений бегущего геометрического среднего)

$\{1, \sqrt{2}, 2, 2\sqrt{2}, 2\sqrt[5]{8}, 2\sqrt[6]{16}, 4\} \to 4.$

Вариант 4. Определить число участков, заполненных нулями.

Пример: $\{0, 1, 2, 3, 0, 0, 0, 0, 4, 1, 2, 0\} \to 3$ (один нуль в начале, четыре нуля в середине и один нуль в конце).

Вариант 5. Определить количество смен направления роста последовательности.

Пример: $\{1, 1, 1, 2, 2, 3, 4, 1, 0, 1\} \to$
$\to \{-, -, -, \uparrow, -, \uparrow, \uparrow, \downarrow, \downarrow, \uparrow\} \to \{\uparrow, \downarrow, \uparrow\} \to 2.$

Вариант 6. Определить, к какому из нижеперечисленных классов принадлежит последовательность. Написать тесты для последовательностей всех классов.

- Класс -1: монотонно (нестрого) убывающая.
- Класс 0: состоящая из одинаковых элементов или пустая.
- Класс $+1$: монотонно (нестрого) возрастающая.
- Класс 2: содержащая как возрастающие, так и убывающие подпоследовательности (в этот класс попадают все последовательности, которые не попали в предыдущие три класса).

Примеры:

- $\{1, 2, 3, 3, 3, 4, 5\} \to +1$;
- $\{4, 4, 4, 3, 1, 0\} \to -1$;
- $\{4, 0, 4, 4, 4, 4\} \to 2$.

Вариант 7. Определить длину самого длинного строго монотонного участка.

Пример: $\{1, 2, 3, -3, -4, -5, 10, 0\} \to 4$ (участок, состоящий из элементов $3, -3, -4, -5$).

Вариант 8. Вычислить максимум гармонических средних участков последовательности, разделенных нулями.

Формула для вычисления среднего гармонического для последовательности $a_1, a_2, \ldots, a_n$ имеет вид:

$$\frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \ldots + \frac{1}{a_n}}.$$

Пример: $\{1, 2, 3, 0, 4, 5, 6, 0, 1, 2, 10, 16, 0, 0, 0, 2\} \to$ участки, разделенные нулями: $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{1, 2, 10, 16\}$, $\{2\} \to$ гармонические средние: $1.(63)$, $4.(864)$, 2.406015, $2 \to$ максимум из них равен $4.(864)$.

Вариант 9. Определить частоты всех возможных значений отдельных байт и сочетаний пар соседних байт в последовательности байт. Выводить только те байты и пары байт, что имеют ненулевые частоты.

Вариант 10. Определить частоты всех сочетаний трех соседних букв в последовательности символов. Для определения факта принадлежности символа к множеству *буквы* использовать стандартную функцию isalpha. Выводить только те сочетания, что имеют ненулевые частоты.

Вариант 11. Пусть $K \in \mathbb{N}$ и a_i — последовательность натуральных чисел. Будем называть $\{ a_i \}$ K-последовательностью, если для всех a_i и a_{i+1} выполнено

$$a_i + 1 \equiv a_{i+1} \pmod{K}.$$

Требуется найти в заданной последовательности длину наибольшего участка, являющегося K-последовательностью (K — параметр функции).

Пример: $K = 3$, $a = \{1, 2, 6, 4, 8, 7, 6, 10, 2, 3, 1, 20, 3, 22\}$, если взять остаток от деления на K, получим

$$\{1, 2, 0, 1, 2, 1, 0, 1, 2, 0, 1, 2, 0, 1\}.$$

Здесь видим K-последовательности: 1, 2, 0, 1, 2 (длина 5); 0, 1, 2, 0, 1, 2, 0, 1 (длина 8). Ответ: 8.

Вариант 12. Пусть задана константа времени компиляции CHUNK_SIZE > 0. Вычислить максимум сумм всех групп соседних элементов размера CHUNK_SIZE для заданной последовательности. Если длина этой последовательности меньше или равна CHUNK_SIZE, то вернуть сумму ее элементов.

Пример. Даны CHUNK_SIZE $= 4$ и последовательность $\{1, -1, 2, 3, -3, 5, 6, 7, -5\}$, имеем группы $(1, -1, 2, 3) \to 5$, $(-1, 2, 3, -3) \to 1$, $(2, 3, -3, 5) \to 7$, $(3, -3, 5, 6) \to 11$, $(-3, 5, 6, 7) \to 15$, $(5, 6, 7, -5) \to 13$, откуда получаем, что максимум сумм равен 15.

Вариант 13. Пусть задана константа времени компиляции WINDOW_SIZE > 0. Вычислить максимум разностей максимума и минимума элементов всех групп соседних элементов размера WINDOW_SIZE для заданной последовательности.

Вернуть разность максимума и минимума элементов исходной последовательности, если ее длина меньше или равна WINDOW_SIZE.

Пример. Пусть WINDOW_SIZE = 4, дана последовательность $\{1, -1, 2, 3, -3, 5, 6, 7, -5\}$, имеем группы $(1, -1, 2, 3) \to 4$, $(-1, 2, 3, -3) \to 6$, $(2, 3, -3, 5) \to 8$, $(3, -3, 5, 6) \to 9$, $(-3, 5, 6, 7) \to 10$, $(5, 6, 7, -5) \to 12$, откуда получаем, что максимум равен 12.

Вариант 14. Пусть дана отсортированная последовательность. **Медианой** назовем значение, которое стоит в середине, если число членов последовательности нечетно, и арифметическое среднее последнего элемента первой половины и первого элемента второй половины последовательности, если число ее членов четно. Таким образом, не менее половины значений последовательности не меньше медианы и не менее половины значений не больше медианы.

Требуется вычислить арифметическое среднее медиан подряд идущих троек элементов произвольной заданной последовательности. Если в конце не хватило элементов на тройку, то взять их арифметическое среднее.

Пример. Дана последовательность

$$\{1, 5, 1, 4, 5, 6, 2, 1, 3, 4, 4, 4, 5, 7\}.$$

Разбиваем на тройки: $\{1, 5, 1\}$, $\{4, 5, 6\}$, $\{2, 1, 3\}$, $\{4, 4, 4\}$, $\{5, 7\}$, последняя группа состоит из двух элементов — берем среднее. Получаем набор медиан: 1, 5, 2, 4, 6. Арифметическое среднее этих значений равно 3.6.

Вариант 15. Представление дискретного сигнала в виде последовательности приращений (разностей между соседними значениями) называют **дельта**[10]**-кодированием** [*delta-coding*].

[10] От традиционного обозначения разности буквой Δ.

Обозначим исходный сигнал через последовательность замеров $\{ s_i \mid 0 \leqslant i \leqslant S \}$, а дельта-кодированный сигнал, соответственно, через $\{ d_i \mid 0 \leqslant i \leqslant S \}$. Тогда они связаны следующими соотношениями: $d_0 = s_0$, $d_i = s_i - s_{i-1}$. Соответственно, $s_i = d_0 + d_1 + \ldots + d_i = s_{i-1} + d_i$.

Дельта-кодирование часто используется как составная часть алгоритмов сжатия данных. В частности, дельта-кодирование превращает последовательность повторяющихся значений в последовательность нулей. Дельта-кодирование уменьшает амплитуду медленно изменяющихся сигналов.

Кодирование длин серий [*run length encoding, RLE*] — представление цифрового сигнала в виде последовательности пар (*значение, количество идущих подряд повторений*).

Собственно RLE обычно не так полезно (размер представления чаще вырастает, а не снижается). Композиция RLE и дельта-кодирования может представлять практический интерес, если сделать следующую модификацию: *указывать число повторений только для нулей* (кодирование длин серий нулей, ZRLE). Таким образом, если при кодировании встретили ненулевое значение, то пропускаем его «как есть», а если нулевое, то записываем нуль и дальше считаем идущие подряд нули, записываем значение счетчика, когда они кончатся или при достижении счетчиком максимального допустимого значения.

Требуется реализовать дельта-кодирование и кодирование длин серий нулей. Закодировать и раскодировать произвольную строку (дельта-кодирование $\rightarrow$ кодирование длин серий нулей $\rightarrow$ декодирование длин серий нулей $\rightarrow$ дельта-декодирование). Можно реализовать либо только работу с массивами символов (строками), либо только работу с потоками ввода-вывода.

Исходная и декодированная строки должны совпадать, последовательность из повторяющихся символов сокращается до одного этого символа, за которым идут нуль и количество повторений.

15

Разные задачи I

Цели лабораторной работы

- Дальнейшее развитие навыков алгоритмизации.
- Закрепление навыков процедурного программирования на языке C++ и алгоритмизации.

В данной работе предполагается, что студенты будут решать задания по очереди «на проекторе».

Задача 1. Случайная перестановка

Дан код:

```
#include <utility> // std::swap
#include <random>
using namespace std;
// Следующее псевдослучайное число.
size_t random() {
  // Генератор типа «вихрь Мерсенна».
  // Инициализация случайным зерном (зависит от реализации).
  thread_local mt19937_64 rng { random_device{}() };
  // Получить следующий элемент последовательности.
  return rng();
}
```

Реализовать с помощью него функцию, выполняющую случайную перестановку элементов массива.

Алгоритм: обменять каждый элемент массива (цикл for по всем элементам) со случайно выбранным элементом.

Написать простой тест. Например, заполнить массив числами от 0 до n, выполнить перестановку и вывести результат на экран.

Задача 2. Перестановка элементов массива

Пусть задана перестановка элементов (массив индексов). Применить ее к заданному массиву на месте. В процессе допускается изменять применяемую перестановку.

Алгоритм: обменять каждый элемент массива с тем элементом, индекс которого стоит в перестановке. После обмена «первый» элемент стоит на месте, а значит, потом мы не имеем права менять его с другими. Следовательно, менять элементы можно только в том случае, если индекс в перестановке больше индекса текущего элемента.

Пример

Перестановка:

индекс	*0*	*1*	*2*	*3*	*4*	*5*	*6*
значение	2	4	0	1	3	6	5

Массив:

индекс	*0*	*1*	*2*	*3*	*4*	*5*	*6*
значение	0.1	1.5	−1.0	2.0	−0.5	7.7	3.4

После применения перестановки к массиву в данном примере получим следующее содержимое массива:

индекс	*0*	*1*	*2*	*3*	*4*	*5*	*6*
значение	−1.0	−0.5	0.1	1.5	2.0	3.4	7.7

Задача 3. Обращение перестановки

Задача, обратная предыдущей. Дана перестановка и переставленный в соответствии с ней массив. Восстановить исходное состояние массива на месте.

В процессе работы можно изменять саму перестановку.

Алгоритм: для каждого элемента массива a[i] выбираем потенциальную обменную пару j = p[i], k = p[j] (здесь p — массив индексов перестановки); если j и k не равны, выполняем обмен a[j], a[k] и p[j], p[k].

В результате к массиву и перестановке будет применена обратная перестановка p^{-1}. Перестановка p в этом случае должна превратиться в тождественную.

Пример

Перестановка:

индекс	*0*	*1*	*2*	*3*	*4*	*5*	*6*
значение	2	4	0	1	3	6	5

Массив:

индекс	*0*	*1*	*2*	*3*	*4*	*5*	*6*
значение	−1.0	−0.5	0.1	1.5	2.0	3.4	7.7

После применения обратной перестановки к массиву в данном примере получим:

индекс	*0*	*1*	*2*	*3*	*4*	*5*	*6*
значение	0.1	1.5	−1.0	2.0	−0.5	7.7	3.4

Задача 4. Сортировка выбором

Реализовать сортировку массива выбором максимального элемента.

Алгоритм

- Пусть массив a имеет размер n.
- Для всех i от n−1 до 1 найти индекс максимального элемента среди элементов с индексами от 0 до i, обменять этот максимальный элемент с элементом a[i].
- Результат: a упорядочен по возрастанию.

Задача 5. Сортировка строк подсчетом по одному символу

Пусть задан массив строк, который требуется отсортировать по символу в заданной позиции (предположим, что все они достаточно длинны для этого).

Предположим также, что строки заданы в 8-битной кодировке и порядок сортировки совпадает с порядком кодов символов. В этом случае у нас есть не более 256 разных значений символа, и каждая строка попадает в ту или иную группу строк, определяющуюся кодом символа.

Пусть задан номер символа, по которому осуществляется сортировка, обозначим его через k. Предположим, что все сортируемые строки включают более k символов (достаточно длинны).

Алгоритм

- Вычислить, сколько раз встречаются строки с каждым кодом в k-й позиции. Это можно сделать с помощью 256 счетчиков.

- Просуммировать эти количества, чтобы получить смещения каждой группы в итоговой сортировке.

- «Выложить» строки в порядке сортировки (все группы подряд).

Для выполнения второго пункта можно использовать следующую конструкцию:

```
size_t offs[256] {};
// ...
// Вычисление смещений (частичные суммы размеров).
for (size_t s = 0, i = 0; i < 256; ++i) {
  auto const t = offs[i] + s;
  offs[i] = s;
  s = t;
}
```

Для выполнения третьего пункта предлагается использовать вспомогательный массив для записи в него сортирующей перестановки и применить затем метод из задачи 2. Заголовок процедуры сортировки может иметь такой вид:

```
void ch_sort(string a[], size_t n, size_t k,
             size_t p[])
```

Задача 6. Длина пути и периметр

Дана последовательность точек на плоскости (каждая точка задана парой координат).

Вычислить длину пути, проходящего через все точки последовательности в порядке их перечисления.

Вычислить периметр многоугольника, заданного последовательностью его вершин (= путь + последнее ребро от последней точки к начальной).

Задача 7. Треугольник, имеющий наименьший периметр

Дан массив точек на плоскости (решить, как представлять точки). Найти треугольник, имеющий наименьший периметр.

Программа должна выводить треугольник (три его вершины) и его периметр.

Задача 8. Список простых чисел

Пользователь вводит число n. Программа должна вычислить и вывести первые n простых чисел.

16

Конечные автоматы

Цель самостоятельной работы

- Приобретение навыков конструирования и программной реализации простых детерминированных конечных автоматов.

Критерии полноты

1. Составлена схема конечного автомата (если задан конкретный конечный автомат).

2. Реализована отдельная функция или объект, воплощающие логику конечного автомата.

3. Предоставлен пример входной последовательности, достаточно по́лно демонстрирующей поведение автомата (автомат посещает все свои состояния в процессе обработки этой последовательности).

16.1. Примеры

Задание. Пусть дан файл с исходным кодом на языке C++. Требуется удалить из него все комментарии.

Будем решать эту задачу с помощью конечного автомата, получающего на вход исходный файл и выдающего на выход тот же файл, но уже без комментариев. На схеме (рис. 16.1) указаны следующие состояния:

- Normal mode — начальное состояние, «обычный текст программы».
- Char literal — идет символьный литерал.
- Char escape — в символьном литерале встретился escape-символ (обратная косая черта).
- String literal — идет строковый литерал.
- String escape — в строковом литерале встретился escape-символ (обратная косая черта).
- Possible comment — в коде встретилась косая черта, возможное начало комментария.
- Oneliner — идет однострочный комментарий (встретилась комбинация //).
- Comment concat — в конце однострочного комментария встретился символ \, возможно, присоединяющий следующую строку к комментарию.
- Multiliner — идет многострочный комментарий (встретилась комбинация /*).
- Possible end — возможное завершение многострочного комментария (встретилась звездочка), на диаграмме /sp обозначает вывод пробела.

Схема несколько упрощена по сравнению с современным синтаксисом C++ (не поддерживаются ' внутри числовых литералов, а также raw-литералы символов и строк).

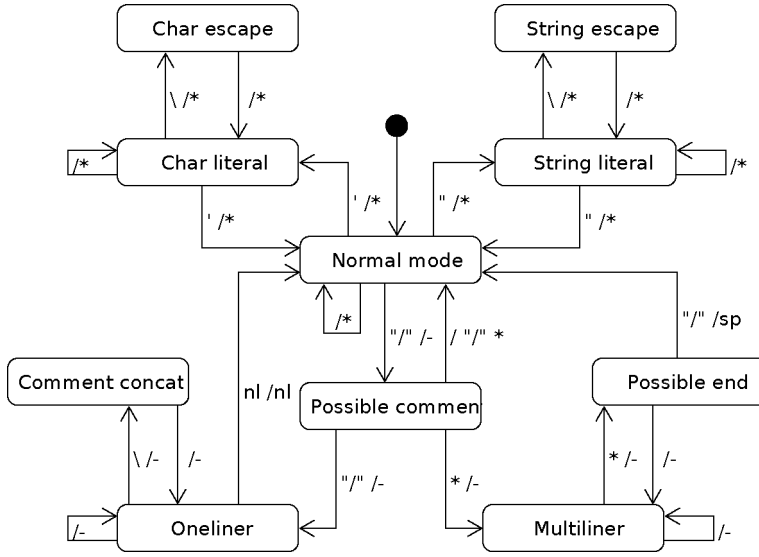

Рис. 16.1. Схема конечного автомата, удаляющего комментарии C++

Приведенный ниже код данного автомата рассчитан на обработку одного файла исходного кода, целиком размещенного в оперативной памяти. Для выполнения этой задачи в память считывается все содержимое потока ввода. Затем полученный массив символов обрабатывает автомат, и результат записывается в поток вывода.

В процессе работы автомата from указывает на текущий символ, to указывает на символ, следующий сразу за последним символом обрабатываемого текста (и не меняется). Указатель out установлен на начало массива символов, в который записывается результат.

Пример 16.1. Удаление комментариев из кода C++

```
char * cpp_decomment(
    char const * from, char const * to,
    char * out) {
Normal_mode:
  while (from != to) {
    switch (char const in = *from++) {
    case '\'':
      *out++ = in;
      goto Char_literal;
    case '\"':
      *out++ = in;
      goto String_literal;
    case '/':
      goto Possible_comment;
    default:
      *out++ = in;
    }
  }
  return out;

Char_literal:
  while (from != to) {
    switch (char const in = *from++) {
    case '\\': // Char_escape
      *out++ = in;
      if (from != to)
        *out++ = *from++;
      break;
    case '\'':
      *out++ = in;
      goto Normal_mode;
    default:
      *out++ = in;
    }
  }
  return out;

String_literal:
  while (from != to) {
```

```
    switch (char const in = *from++) {
    case '\\': // String_escape
      *out++ = in;
      if (from != to)
        *out++ = *from++;
      break;
    case '\"':
      *out++ = in;
      goto Normal_mode;
    default:
      *out++ = in;
    }
  }
  return out;

Possible_comment:
  if (from == to)
    return *out++ = '/', out;

  switch (char const in = *from++) {
  case '/': goto Oneliner;
  case '*': goto Multiliner;
  default:
    out[0] = '/';
    out[1] = in;
    out += 2;
    goto Normal_mode;
  }

Oneliner:
  while (from != to) {
    switch (const char in = *from++) {
    case '\n':
      *out++ = in;
      goto Normal_mode;
    case '\\': // Comment_concat
      if (from != to)
        ++from; // просто пропустим еще символ
      break;
    default:; // пропустим символ
```

```
    }
  }
  return out;

Multiliner:
  while (from != to) {
    if (*from++ == '*')
      goto Possible_end;
  }
  return out;

Possible_end:
  if (from != to && *from++ == '/') {
    *out++ = ' ';
    goto Normal_mode;
  }
  goto Multiliner;
}
```

В нескольких вариантах данной работы для ряда языков программирования предлагается реализовать аналогичный конечный автомат, выполняющий удаление комментариев.

Удобно проверять вхождение коротких подстрок сразу одним условием, не вводя промежуточные состояния. Например, для проверки, стоит ли далее последовательность """, можно написать такой код:

```
switch (const char in = *from++) {
case '"':
  if (to - from > 1
      && from[0] == '"' && from[1] == '"')
    from += 2; // встретили "
    ...
```

Эту конструкцию можно обобщить и записать в виде отдельной функции, проверяющей совпадение префикса ввода с заданной строкой.

От проверок достаточной длины остатка вроде $\mathrm{to} - \mathrm{from} > 1$ можно избавиться, если заранее дополнить считанный текст достаточно длинной последовательностью нулевых символов.

16.2. Варианты заданий

Вариант 1. Удаление комментариев из исходных файлов на языке MATLAB.

Синтаксис:

- однострочные комментарии (вариант 1) начинаются с `%` или (вариант 2) с `...`;
- многострочные комментарии начинаются с `%{` и заканчиваются `%}`;
- строковые константы отделяются апострофами: `'simple'`. Экранирование специальных символов как в C.

Вариант 2. Удаление комментариев из исходных файлов на языке Pascal (предполагается диалект Free Pascal).

Синтаксис:

- однострочные комментарии начинаются с `//`;
- многострочные комментарии (вариант 1): начинаются с `{`, заканчиваются `}`;
- многострочные комментарии (вариант 2): начинаются с `(*`, заканчиваются `*)`;
- строковые константы, выделенные апострофами, например, `'no escape-seqs'`. Экранирование не предусмотрено.

Вариант 3. Удаление комментариев из исходных файлов на языке Python.

Упрощенный синтаксис:

- комментарии — только однострочные: начинаются с `#`;
- однострочные строковые константы: отделенные апострофами `'lewy'` или кавычками `"prawy"`. Экранирование специальных символов как в C;

- многострочные строковые константы: тройные апострофы `'''"what a 'nice' doggy here"'''` или тройные кавычки `"""say "Abracadabra"!"""`. Экранирования нет.

Вариант 4. Удаление комментариев из исходных файлов на языке Lua.

Упрощенный синтаксис:

- однострочные комментарии начинаются с `--`;
- многострочные комментарии начинаются с `--[[`, заканчиваются `]]`;
- строки, отделенные апострофами: `'say "Hi!"\n'`. Экранирование специальных символов как в C;
- строки в кавычках: `"don't say \"Hi!\"\0"`. Экранирование специальных символов как в C;
- строки от `[[` до `]]` без экранирования;
- строки от `[=[` до `]=]` без экранирования.

Количество знаков `=` в открывающей скобке может быть произвольным: столько же знаков `=` должно быть и в закрывающей скобке (последние два пункта — частный случай). Для корректной обработки таких строк следует ввести дополнительный счетчик знаков `=`.

Вариант 5. Удаление комментариев из исходных файлов на языке Julia.

Упрощенный синтаксис:

- однострочные комментарии начинаются с `#`;
- многострочные комментарии начинаются с `#=`, заканчиваются `=#`. Они ведут себя как скобки, т. е. сколько было открывающих `#=`, столько затем должно быть и закрывающих `=#`. Для корректной обработки таких комментариев следует ввести дополнительный счетчик текущей глубины вложенности (количества пока не закрытых `#=`);

- символьные и строковые константы могут размещаться между апострофами: `'\\'` или кавычками: `"This is a \"quote\""`. Экранирование символов как в C;
- строковые константы в утроенных кавычках: `"""Contains "quote" characters"""` без экранирования;
- пользовательские строковые литералы: буква, за которой сразу следует строка в кавычках, но без экранирования, например: `r"^\s*(?:#|$)"`.

Вариант 6. Удаление комментариев из исходных файлов на языке Haskell.

Упрощенный синтаксис:

- однострочные комментарии открываются `--`, после чего идет либо символ `-`, либо пробел, либо буква (это важно, так как в коде на Haskell возможна запись операции вроде `-->`);
- многострочные комментарии открываются `{-`, закрываются `-}`. Могут быть вложенными как скобки, т. е. на каждую открывающую `{-` должна приходиться своя закрывающая `-}`. Корректная обработка таких комментариев может быть организована с помощью дополнительного счетчика, хранящего количество открытых, но еще не закрытых `{-`;
- символьные и строковые константы, отделенные апострофами `'\t'` или кавычками: `"Hello,\n"`. Экранирование специальных символов как в C.

Вариант 7. Заменить маленькие буквы, с которых начинаются предложения, на большие. Также определить количество предложений (дополнительное состояние: счетчик предложений). Каждое предложение заканчивается, по крайней мере, одной буквой, за которой идет последовательность точек, восклицательных или вопросительных знаков, состоящая не менее

чем из одного символа. Если ввод на этом не заканчивается, то далее должен стоять пробельный символ.

Вариант 8. Выполнить распознавание записи целого числа в формате, принятом в языке C. Помимо десятичной предусмотреть корректное распознавание двоичных (начинаются на `0b` или `0B`) и шестнадцатеричных (начинаются на `0x` или `0X`) форм.

Примеры: `129`, `0`, `0b110`, `0XDeed`.

Вариант 9. Выполнить распознавание записи числа с плавающей точкой в формате, принятом в языке C. Автомат должен определять правую границу записи и сохранять распознанное число по переданной ссылке на переменную.

Примеры: `.10`, `1.`, `2e2`, `1.05E-50`, `500.e+100`.

Вариант 10. Реализовать автомат, распознающий последовательности, состоящие из двух монотонных (произвольного направления или констант) кусков.

Возможные классы:

- один кусок
  - константа (`1, 1, 1, 1`),
  - возрастающая (`1, 1, 2, 3`),
  - убывающая (`2, 2, 1, 1`);
- два куска
  - возрастающая-возрастающая (`0, 0, 1, 2, 0, 0, 1`),
  - возрастающая-убывающая (`0, 0, 1, 2, 4, 2, 1`),
  - возрастающая-константа (`0, 0, 1, 2, 0, 0`),
  - убывающая-возрастающая (`4, 4, 1, 5, 5, 8`),
  - убывающая-убывающая (`4, 4, 1, 5, 5, 0`),
  - убывающая-константа (`4, 4, 1, 3, 3, 3`).

Если удалось определить наличие второго куска, то автомат должен как-то возвращать его начало.

Вариант 11. Создать конечный автомат, распознающий последовательности из нулей и единиц, содержащие четное число единиц и не содержащие единиц, идущих подряд друг за другом.

Примеры

- 00000 — принять,
- 01000 — не принимать (нечетное число единиц),
- 00011 — не принимать (две единицы подряд),
- 10100 — принять,
- 10110 — не принимать (нечетное число единиц и две единицы подряд).

17

Разные задачи II

Цели лабораторной работы

- Закрепление навыков процедурного программирования на языке C++.
- Дальнейшее развитие навыков алгоритмизации.

В данной работе предполагается, что студенты будут решать задания по очереди «на проекторе».

Задача 1. Кольцевой буфер

Пусть дана входная последовательность $\{ a_i \}_{i=0}^{\infty}$ и дан набор *весов* $\{ w_j \}_{j=0}^{W}$, $W \in \mathbb{N}$. Положим $a_i = 0$ при $i < 0$.

Требуется реализовать программу, вычисляющую последовательность $\{ b_i \}_{i=0}^{\infty}$ как свертку:

$$b_i = \sum_{j=0}^{W} a_{i-j} w_j.$$

Выполнить данную задачу можно с помощью структуры данных, известной как *кольцевой буфер* [*ring buffer*]. Заготовка исходного кода:

```
#include <cstddef> // size_t
#include <iostream>
using namespace std;
// Тип значения.
using NT = float;
// Количество обрабатываемых элементов.
size_t const WINDOW = 4;
// WINDOW должно быть степенью двойки.
static_assert((WINDOW & (WINDOW-1)) == 0);

// Кольцевой буфер.
struct Ring_buffer {
  NT data[WINDOW] {};

  // Положить новый элемент.
  void put(NT item) { /* TODO */ }
  // Элемент i шагов назад.
  NT const & operator[](size_t i) const
  { /* TODO */ }

private:
  size_t _pos = -1;
};

int main() {
  Ring_buffer buf;
  // Коэффициенты взвешенного среднего. Сумма == 1.
  NT weight[WINDOW]
    { 1.f/2, 1.f/4, 3.f/16, 1.f/16 };
  // Читать последовательность в буфер
  // и применять фильтр на каждом шаге.
  for (NT val; cin >> val;) {
    buf.put(val);
    // Свертка.
    NT s = 0;
    // TODO: дописать код здесь.
    cout << val << '\t' << s << '\n';
  }
  return 0;
}
```

Задача 2. Матрица инцидентности

Проверить, что каждая строка матрицы состоит из нулей и единиц и содержит ровно две единицы.

Пример матрицы инцидентности:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Задача 3. Умножение матриц

Реализовать произведение матриц.

Обратить внимание на производительность полученного алгоритма: она может весьма сильно зависеть от порядка обращения к памяти.

Наивный алгоритм представляет собой непосредственную реализацию определения произведения матриц, что означает вычисление элементов результирующей матрицы друг за другом в двойном цикле, в который вложен цикл, вычисляющий скалярное произведение строки и столбца.

Более эффективный подход заключается в одновременном вычислении строки результирующей матрицы. Для этого следует переставить два внутренних цикла наивного алгоритма так, чтобы наиболее вложенный цикл пробегал по строке результирующей матрицы, добавляя к каждому элементу очередное слагаемое скалярного произведения.

Рекомендуется реализовать тест, позволяющий наглядно убедиться в различии получаемых уровней производительности.

Заготовка исходного кода для данной задачи (с замером времени выполнения операции умножения):

```
#include <cstddef> // size_t
#include <ctime>   // clock
#include <iostream>
```

```
using namespace std;
using NT = double;
void fill(NT m[], size_t sz, NT val = 0.)
{ /* TODO */ }

// A is n*r, X is r*m, Y is n*m.
// Y += A*X
// (A*X)ij = sum(k)(Aik*Xkj).
void axpy(
  NT const a[], size_t n, size_t r,
  NT const x[], size_t m,
  NT y[]) {
  // TODO
}

int main() {
  size_t n = 0, r = 0, m = 0;
  cout << "Sizes? ";
  cin >> n >> r >> m;
  auto A = new NT[n*r];
  auto X = new NT[r*m];
  auto Y = new NT[n*m] {};
  fill(A, n*r, 1.0);
  fill(X, r*m, 0.5);

  auto const t0 = clock();
  axpy(A, n, r, X, m, Y);
  cout << Y[0] << '\t' << (clock() - t0) << endl;
  return 0;
}
```

Задача 4. Возведение матриц в целочисленную степень

Используя операцию умножения из предыдущей задачи, реализовать возведение квадратной матрицы в целочисленную неотрицательную степень, выполняющее логарифмическое число умножений.

В качестве примера приведем алгоритм возведения в целочисленную степень, реализованный для чисел:

```
float ipow(float x, unsigned long n) {
  float p = 1;
  while (n != 0) {
    if (n % 2 == 1)
      p *= x;
    n /= 2;
    x *= x;
  }
  return p;
}
```

При возведении матрицы в степень следует отдельно проверять случай нулевого показателя степени (и возвращать единичную матрицу), в остальных случаях начинать с исходной матрицы, экономя таким образом одно матричное умножение.

18

Си-строки

Цели самостоятельной работы

- Приобретение навыков работы с массивами массивов и нуль-терминированными строками.
- Закрепление навыков организации двойных циклов.

Критерии полноты

1. Реализована отдельная функция, решающая поставленную задачу.
2. «Тексты» (наборы строк) передаются в функцию и возвращаются из функции в виде указателей на нуль-терминированные массивы указателей на си-строки.
3. Элементарные действия над си-строками вынесены в отдельные функции, возможно, дублирующие функционал, предоставляемый стандартным заголовком **cstring** (пример: вычисление длины строки, копирование строки).
4. В виде отдельной функции (не в main) реализовано автоматическое тестирование функции из п. 1 с помощью, по крайней мере, двух не слишком простых тестов.

18.1. Примеры

Операции с массивами си-строк

Покажем на примере реализации операций чтения массива строк из стандартного потока ввода и вывода массива строк в стандартный поток вывода (в «стиле C»). Для реализации данных операций используется стандартная библиотека C.

Для вывода строки в поток будем использовать стандартную функцию puts (заголовочный файл **cstdio**).

Вывод массива строк, учитывая соглашение о том, что последний элемент — нулевой указатель, можно записать в виде простого цикла.

Пример 18.1. Вывод массива си-строк

```
// Вывести в стандартный поток вывода массив строк.
// lines — указатель на массив константных
//  указателей на массивы константных символов.
void putlines(char const* const* lines) {
  if (!lines)
    return;
  while (*lines)
    puts(*lines++);
}
```

Аналогично будет выглядеть и операция освобождения памяти. Предположим, что для выделения памяти на все строки массива и сам массив указателей на строки использовалась стандартная функция malloc — тогда освобождать память надо будет функцией free.

Пример 18.2. Удаление массива си-строк

```
/// Освободить память, занимаемую массивом строк.
void freelines(char ** lines) {
  if (!lines)
    return;
  // Освободить каждую строку.
  for (auto p = lines; *p; ++p)
    free(*p);
```

```
    // Освободить массив указателей.
    free(lines);
}
```

Выполнить чтение строки с потока ввода уже значительно труднее, так как мы не знаем заранее ее длину. Стандартная функция gets читает в буфер, размер которого выбирает пользователь, предполагая, что его «хватит». Такой подход приводит к ошибкам и уязвимостям в коде, поэтому функция gets была объявлена устаревшей и убрана из стандарта C++14.

Вместо gets следует использовать функцию fgets, которая принимает размер буфера. Но она все равно не решает исходную задачу: буфер фиксированного размера мы должны заготовить заранее под строку неизвестной длины. (В стандартной библиотеке C++ функция getline решает за нас эту задачу.)

В данной ситуации у нас есть две опции:

1. Читать из потока ввода посимвольно и динамически увеличивать хранилище по мере надобности.

2. Задать ограничение на длину возможной строки и разбивать слишком длинные строки на несколько строк.

В примере выбран подход (2), а подход (1) применим к массиву указателей на строки (количество строк ввода тоже заранее неизвестно).

Подход (2) имеет очевидный недостаток: «прочитанный» текст может не совпасть с исходным текстом из-за вставки разрывов строк, которых в исходном тексте не было. Однако применение данного подхода все же может быть оправдано следующими соображениями:

- во-первых, наш буфер может быть достаточного размера для реальных текстов (большинство текстов не содержат слишком длинных строк);

- во-вторых, посимвольное чтение строки «вручную» будет работать на порядок медленнее, чем оптимизирован-

ный библиотечный вариант fgets, что может быть важно в некоторых случаях.

Пример 18.3. Ввод си-строки

```
// Максимальная длина строки (не включая '\0'),
// возвращаемой нашим вариантом gets.
size_t const MAX_GETS_LENGTH = 1023;
// Мы реализуем собственную функцию gets,
// создающую строку в динамической памяти.
// Если строка на вводе слишком длинная,
// отрезаем на MAX_GETS_LENGTH символах.
// Если не удалось прочитать ни одного символа,
// возвращаем нулевой указатель.
char * gets() {
  char buffer[MAX_GETS_LENGTH + 1];
  if (!fgets(buffer, MAX_GETS_LENGTH + 1, stdin))
    return nullptr;

  // Узнать длину считанной строки.
  auto alloc_len = strlen(buffer) + 1;
  // fgets записывает в буфер и перевод строки.
  // Уберем этот перевод строки.
  if (alloc_len > 1
   && buffer[alloc_len - 2] == '\n') {
    buffer[alloc_len - 2] = '\0';
    --alloc_len;
  }

  // Выделить в динамической памяти место на копию.
  char * const new_str = (char*)malloc(alloc_len);
  if (!new_str) // не хватило памяти?
    return nullptr;

  // Скопировать содержимое буфера.
  memcpy(new_str, buffer, alloc_len);
  return new_str;
}
```

По мере необходимости будем увеличивать массив строк динамически. Для этого удобно использовать стандартную функцию realloc.

Задействуем следующую стратегию: будем выделять пространства больше, чем занято, и на каждой итерации проверять, не исчерпана ли *емкость* [*capacity*] (доступное пространство). Если исчерпана, то пробуем увеличить ее в $1 < g \leqslant 2$ раз, но не менее чем на один элемент. В примере выбрано значение $g = 1.5$. В случае неудачи пробуем увеличить хранилище на один элемент.

```
// Читаем построчно и складываем
//  в динамический массив указатели.
// Последний указатель в массиве всегда нулевой.
// Функция возвращает нулевой указатель в
//  случае невозможности выделить память хотя бы
//  на два указателя.
// Увеличиваем массив по мере необходимости.
char ** getlines() {
  // calloc сразу обнуляет выделяемую память.
  char ** lines = (char**)calloc(2, sizeof(char*));
  if (!lines) // не хватило памяти?
    return nullptr;
  // Цикл по строкам.
  for (size_t item = 0, capacity = 2;
    lines[item++] = gets();) {
    // Необходимо увеличить массив?
    if (capacity == item + 1) {
      // Попытка 1:
      // попробуем увеличить capacity в 1.5 раза.
      capacity += capacity / 2 + 1;
      // Для больших capacity возможно переполнение!
      assert((capacity*sizeof(char*))/sizeof(char*)
             == capacity);

      if (char ** new_lines =
        (char**)realloc(lines,
                        capacity*sizeof(char*))) {
        lines = new_lines;
```

```
        }
        else
        // Попытка 2:
        // попробуем увеличить capacity на 1 элемент.
        if (char ** new_lines =
          (char**)realloc(lines,
                          (item + 2)*sizeof(char*)))
        {
          capacity = item + 2;
          lines = new_lines;
        }
        else // не получается выделить память
        {
          // Закрывающий нулевой указатель.
          lines[item] = nullptr;
          break;
        }
      }
    }

    return lines;
}
```

Удаление однострочных комментариев

Удалить однострочные комментарии (открывающий комментарии символ задан в виде параметра) без учета возможных строковых литералов (пренебрегаем возможной ситуацией, когда «комментарий» на самом деле может быть не комментарием, а частью строковой константы).

В C++ однострочный комментарий открывается парой символов //, поэтому решение, приведенное ниже, неприменимо к коду на языке C++.

Алгоритм. Для каждой строки найти первое вхождение заданного символа и обнулить этот байт.

Пример 18.4. Удаление комментариев в конце строк

```
// Найти в си-строке str символ ch.
```

```
// Возвращает указатель на найденный символ
// или нулевой указатель, если символа в строке нет.
char* find_char(char *str, char ch) {
  for (; *str != '\0'; ++str)
    if (*str == ch)
      return str;
  return nullptr;
}

// Затереть нулем первый символ комментария,
// закончив таким образом на нем си-строку.
void remove_comment(
    char * line, char comment_mark) {
  if (auto pos = find_char(line, comment_mark))
    *pos = '\0';
}

// Удалить комментарии из текста.
void remove_comments(
    char * text[], char comment_mark) {
  for (; *text != nullptr; ++text)
    remove_comment(*text, comment_mark);
}
```

Тестирование: массив строк input содержит вход, массив строк reference содержит ожидаемый результат.

```
/// Тест функции remove_comments.
bool test_remove_comments() {
  // Исходный текст.
  char const * input[] {
    "",
    "# comment",
    "something; # comment",
    "'hello, world!'",
    "'# OK to cut here '... # a real comment",
    nullptr
  };

  // Ожидаемый результат.
  char const * reference[] {
```

```
    "",
    "",
    "something;␣",
    "'hello,␣world!'",
    "'",
    nullptr
  };
  // ...
}
```

Данная функция не дописана, потому что не хватает ряда вспомогательных определений. Дело в том, что input нельзя подать на обработку функцией remove_comments непосредственно. Наши тестовые массивы содержат указатели на содержимое строковых констант, которые нельзя изменять — попытка сделать это влечет неопределенное поведение. Поэтому для выполнения теста придется копировать исходный текст в динамические массивы, которые изменять можно.

В отличие от предыдущего примера здесь для управления памятью воспользуемся **new**/**delete**.

```
#include <cstring>
// Определить размер массива (количество строк).
std::size_t lines_in_text(
    char const * const source[]) {
  std::size_t source_size = 0;
  for (auto p = source; *p != nullptr; ++p)
    ++source_size;
  return source_size;
}

// Вспомогательная функция для копирования текста.
// Она нужна для того, чтобы не изменять значения,
// заданные строковыми литералами
// (что чревато неопределенным поведением).
char ** copy_text(char const * const source[]) {
  // Создать массив строк.
  auto const source_size = lines_in_text(source);
  auto copy = new char*[source_size + 1];
  // Скопировать массив построчно.
```

```
  for (size_t i = 0; i < source_size; ++i) {
    // Выделить память для новой строки.
    auto const line_len =
        std::strlen(source[i]) + 1;
    copy[i] = new char[line_len];
    // Скопировать строку и завершающий нуль.
    std::memcpy(copy[i], source[i], line_len);
  }
  // Записать завершающий нуль.
  copy[source_size] = nullptr;
  // Вернуть результат.
  return copy;
}

// Освобождение памяти, занятой текстом,
// созданным с помощью функции copy_text.
void free_text(char * text[]) {
  for (auto p = text; *p != nullptr; ++p)
    delete[] *p;
  delete[] text;
}
```

Сравнивать результат работы remove_comments и содержимое массива reference будем с помощью вспомогательной функции are_equal (ее определение и полный текст функции тестирования приведены ниже).

```
// Сравнение текстов на равенство.
bool are_equal(
    char const * const text1[],
    char const * const text2[]) {
  for (; *text1 && *text2; ++text1, ++text2) {
    // Сравнение си-строк на равенство
    // с помощью стандартной функции.
    if (std::strcmp(*text1, *text2) != 0)
      return false;
  }
  // Оба указателя должны быть нулевыми, если тексты совпадают.
  return *text1 == *text2;
}
```

```
// Тест функции remove_comments.
bool test_remove_comments() {
  // Исходный текст.
  char const * input[] {
    "",
    "# comment",
    "something; # comment",
    "'hello, world!'",
    "'# OK to cut here '... # a real comment",
    nullptr
  };

  // Ожидаемый результат.
  char const * reference[] {
    "",
    "",
    "something; ",
    "'hello, world!'",
    "'",
    nullptr
  };

  auto text = copy_text(input);
  remove_comments(text, '#');
  // Сравнить на равенство и освободить память.
  auto const result = are_equal(text, reference);
  free_text(text);
  return result;
}
```

Разбиение строки на слова

Дана си-строка, требуется извлечь из нее слова, заполнив массив указателями на первые буквы слов, а пробельные символы заменив нулевыми символами. Память под массив выделяется извне (пользователем). Решение данной задачи оформим в виде функции, принимающей указатель на массив слов и его размер и возвращающей указатель на последний необра-

ботанный символ строки. Этот указатель будет нулевым, если вся строка обработана. Он может быть ненулевым из-за того, что место в массиве слов может закончиться раньше, чем будет достигнут конец строки. Массив слов является массивом указателей на си-строки и должен завершаться нулевым указателем.

Алгоритм. Пока исходная строка не кончилась, циклически повторяем два действия:

1. Проходим по исходной строке, затирая нулями пробельные символы. Если встретился непробельный символ, то записываем указатель на него на очередную позицию в массиве.

2. Двигаемся по строке до первого пробельного символа.

Итак, реализуем эти два действия в виде функций:

```
#include <cctype> // isspace
// Затереть последовательность пробельных символов
// нулевым символом.
char * zero_spaces(char * text) {
  while (std::isspace(*text))
    *text++ = '\0';
  return text;
}

// Пропустить все непробельные символы
// (кроме завершающего нуля).
char * skip_non_spaces(char * line) {
  while (*line != '\0' && !std::isspace(*line))
    ++line;
  return line;
}
```

Реализация основной функции (назовем ее split):

```
// Разбить строку text на слова words.
// Массив words создается кодом, вызывающим split.
// Параметром words_size передается размер words.
char * split(char * text,
```

```
    char * words[], std::size_t words_size) {
  assert(words_size > 1);
  auto const max_words = words_size - 1;
  for (std::size_t word = 0;
       word < max_words; ++word) {
    text = zero_spaces(text);
    if (*text == '\0') {
      words[word] = nullptr;
      return nullptr;
    }

    words[word] = text;
    text = skip_non_spaces(text);
  }

  // Завершающий слова нуль.
  words[max_words] = nullptr;
  // Перейти к началу следующего слова.
  text = zero_spaces(text);
  // Если строка кончилась, вернуть nullptr.
  return *text != '\0'? text : nullptr;
}
```

Тестирующая функция возвращает код ошибки. Это нуль, если ошибок не было обнаружено. Нам также понадобится функция are_equal из предыдущего примера.

```
int test_split() {
  char text[] = "Just a simple sentence.";
  char const * reference[] {
    "Just",
    "a",
    "simple",
    "sentence.",
    nullptr
  };

  // Проверка случая, когда
  // передан массив достаточного размера.
  char * words[10];
  // split должна вернуть нулевой указатель
```

```
  if (split(text, words, 10))
    return 1;
  if (!are_equal(words, reference))
    return 2;

  // Проверка случая, когда
  // передан массив недостаточного размера.
  char long_text[] =
    "This program is free software;"
    "you can redistribute it and/or modify\n"
    "it under the terms of the GNU General"
    "Public License as published by\n"
    "the Free Software Foundation; either"
    "version 3 of the License, or (at\n"
    "your option) any later version.";

  auto const last_pos = split(long_text, words, 10);
  if (!last_pos
   || sizeof(long_text) < last_pos - long_text)
    return 3;
  if (*last_pos != 'a') // следующее «слово»
    return 4;           // было бы «and/or»

  char const * long_reference[] {
    "This",
    "program",
    "is",
    "free",
    "software;",
    "you",
    "can",
    "redistribute",
    "it",
    nullptr
  };

  if (!are_equal(words, long_reference))
    return 5;
  return 0; // ошибки не обнаружены
}
```

18.2. Варианты заданий

Вариант 1. Построить строку, содержащую подряд все строки исходного массива, перемежающиеся заданной строкой-разделителем. Сложность алгоритма должна быть линейной по общему количеству символов. Массив передается по указателю типа **char const** $*$ **const** $*$. Длина массива заранее не известна и не передается, поскольку последний элемент массива — нулевой указатель (завершающий нуль аналогично си-строке). Функция также должна принимать указатель на си-строку, которая будет вставляться между строками массива, выполняя роль разделителя.

Пример. Пусть дан входной массив вида:

```
{
  "alpha",
  "beta",
  "gamma",
  nullptr
};
```

и строка-разделитель: ",␣". Тогда результирующая строка будет равна "alpha,␣beta,␣gamma".

Вариант 2. Построить набор строк (массив указателей на строки), заменив в исходном файле заданный символ на нулевые символы.

Пример. Дан текст "One␣Two␣Three" и символ-разделитель '␣' (пробел). Результирующий массив должен содержать указатели на три си-строки, равные "One", "Two" и "Three" и завершающий нулевой указатель.

Вариант 3. Пусть задана максимальная допустимая длина строки. Построить новый массив строк, выполнив перенос остатков на новые строки.

Пример. Пусть даны ограничение на ширину строки 6 символов и входной массив вида:

```
{
  "short",
  "too long",
  "very very long",
  nullptr
};
```

Результирующий массив должен быть равен следующему массиву:

```
{
  "short",
  "too lo",
  "ng",
  "very v",
  "ery lo",
  "ng",
  nullptr
};
```

Вариант 4. Заменить символы табуляции '\t' в начале каждой строки на пробелы. Количество пробелов, соответствующих одному символу табуляции, передавать параметром.

Вариант 5. Заменить пробелы в начале каждой строки на символы табуляции '\t'. Количество пробелов, соответствующих одному символу табуляции (т. е. задающее ширину колонки), передавать параметром. Если количество пробелов в начале строки не делится нацело на заданную ширину колонки, то оставлять после символов табуляции дополнительные пробелы, количество которых равно остатку от деления количества пробелов на ширину колонки.

Вариант 6. «Раскомментировать» строки. Комментарий начинается с заданного символа и продолжается до конца строки. Удалить первый такой символ (после возможно пустой последовательности пробельных символов) в каждой строке.

Пример. Пусть даны символ ';', открывающий комментарий, и входной массив вида:

```
{
  "; only a comment here",
  "; a = 10; commented code here",
  "b = 20;; code and comment",
  nullptr
};
```

Результирующий массив должен быть равен следующему массиву:

```
{
  "; only a comment here",
  " a = 10; commented code here",
  "b = 20;; code and comment",
  nullptr
};
```

Вариант 7. «Закомментировать» заданное число строк. Комментарий начинается с заданного символа и продолжается до конца строки. Вставить такой символ в начало каждой строки.

Пример. Пусть даны символ ';', открывающий комментарий, и входной массив вида:

```
{
  "; only a comment here",
  " a = 10; commented code here",
  "b = 20;; code and comment",
  nullptr
};
```

Результирующий массив должен быть равен следующему массиву:

```
{
  "; ; only a comment here",
  "; a = 10; commented code here",
  "; b = 20;; code and comment",
  nullptr
};
```

Вариант 8. Пусть исходный набор строк можно условно представить в виде следующих друг за другом блоков (A и X — некоторые последовательности строк) как AXA, т. е. блок A повторяется в конце набора. При этом A — максимальный по размеру такой блок.

Требуется удалить финальный блок A. Результирующий набор строк должен иметь вид: AX.

Вариант 9. Удалить из набора подряд идущие дубликаты строк (т. е. из идущих подряд повторяющихся строк оставлять только первую). Алгоритм должен быть линейным по числу строк.

Вариант 10. Если префикс строки в массиве совпадает с заданной строкой — удалить его из этой строки.

Пример. Пусть дан префикс "cl" и массив:

```
{
  "clCreateKernel",
  "clRetainKernel",
  "clReleaseKernel",
  "clSetKernelArg",
  "omp_get_initial_device",
  nullptr
};
```

Результирующий массив должен быть равен следующему массиву:

```
{
  "CreateKernel",
  "RetainKernel",
  "ReleaseKernel",
  "SetKernelArg",
  "omp_get_initial_device",
  nullptr
};
```

Вариант 11. Если с заданной строкой совпадает суффикс строки — удалить эту строку из массива (удалять «на месте», т. е. изменять исходный массив). Алгоритм должен быть линейным по числу строк.

Пример. Пусть дана строка "Socket" и массив:

```
{
  "CreateWindow",
  "OpenSocket",
  "ReleaseHandle",
  "CloseSocket",
  "Socket",
  "Sockets",
  nullptr
};
```

Результирующий массив должен быть равен следующему массиву:

```
{
  "CreateWindow",
  "ReleaseHandle",
  "Sockets",
  nullptr
};
```

Вариант 12. Разбить массив на два массива строк: лексикографически меньших заданной строки и прочих.

Пример. Пусть дана строка "kaf" и массив:

```
{
  "alef",
  "bet",
  "gimel",
  "resh",
  "vav",
  "ayin",
  "kaf",
  nullptr
};
```

Результирующие массивы:

```
{
  "alef",
  "bet",
  "gimel",
  "ayin",
  nullptr
};

{
  "resh",
  "vav",
  "kaf",
  nullptr
};
```

Вариант 13. Удалить из массива все строки, являющиеся повторением заданной строки. Строки удалять «на месте», т. е. не создавать новый массив.

Пример. Пусть даны строка "ne" и массив:

```
{
  "meager",
  "ne",
  "nen",
  "nene",
  "nenee",
  "new",
  "neven",
  "nenene",
  nullptr
};
```

Результирующий массив должен быть равен следующему массиву:

```
{
  "meager",
  "nen",
  "nenee",
```

```
  "new",
  "neven",
  nullptr
};
```

Вариант 14. Пусть дан массив си-строк. Выполнить в нем удвоение заданного символа.

Пример. Пусть даны символ '%' и массив:

```
{
  "Hi, gals!\n",
  "%d%c%c%s",
  "We were growing 3% each year.",
  nullptr
};
```

Результирующий массив должен быть равен следующему массиву:

```
{
  "Hi, gals!\n",
  "%%d%%c%%c%%s",
  "We were growing 3%% each year.",
  nullptr
};
```

Вариант 15. Пусть дан текст, заданный нуль-терминированным массивом си-строк. Пусть дано слово. Найти первое вхождение этого слова в тексте при условии, что оно в любой позиции может быть разбито переносом, т. е. знаком –, завершающим строку. Остаток слова переносится на следующую строку. Вхождение определяется как пара чисел: индекс строки в массиве и индекс первого символа слова в строке, где оно найдено.

Рекомендация: для каждой строки текста сначала попробовать найти в ней первое вхождение слова целиком (можно воспользоваться для этого стандартной функцией strstr из заголовочного файла **cstring**). Если слово не найдено, а строка

заканчивается символом $-$, то взять конкатенацию $(n-1)$ символов перед знаком $-$ и $(n-1)$ первых символов следующей строки (естественно, следует проверять, возможно ли это) и попробовать найти слово в полученной строке. Для этого разумно заранее определить локальный массив размера $(2n-2)$, где n — длина слова.

19

Введение в численные методы

Цели лабораторной работы

- Изучение работы с многочленами.
- Применение метода бисекции для поиска решений алгебраических уравнений.

Поиск корней многочленов

Решение данной части фактически сводится к написанию трех циклов, один из которых должен находиться в отдельной функции. Требуется:

• реализовать чтение коэффициентов многочлена произвольной степени со стандартного потока ввода в глобальный динамический массив. Количество коэффициентов многочлена на единицу больше его степени. Для удобства можно пронумеровать элементы массива в обратном порядке (от старших к младшим);

• на основе метода Горнера реализовать функцию вычисления значения многочлена, заданного массивом коэффициен-

тов. Метод Горнера сводится к простому циклу, напоминающему вычисление суммы:

$$P(x) = \sum_{i=0}^{n} a_i x^i = s_0,$$
$$s_i = s_{i+1}x + a_i, \quad s_n = a_n;$$

- реализовать сеточный поиск корней многочлена. Пользователь вводит границы поиска l и r и количество шагов (узлов сетки) k. Шаг между узлами $s = \frac{r-l}{k}$, узел $l_i = l + is$.

Переберем все промежутки ($k+1$ «штука»). Если значение многочлена принимает разные знаки на l_i и l_{i+1}, то используем бисекцию, чтобы найти корень многочлена на $[l_i, l_{i+1}]$.

Автоматическая посадка

Задача. Модифицируйте программу из лабораторной работы 6 таким образом, чтобы при обнаружении пересечения уровня $x = 0$ она находила момент времени пересечения и параметры системы в этот момент. Для решения уравнения можно использовать бисекцию.

Идея: на некотором шаге случается, что $x_0 > 0$, а $x_1 < 0$. Теперь можно численно решить неявно заданное уравнение $x(\Delta t) = 0$, где Δt ограничен снизу 0, а сверху введенным пользователем значением.

Достигнутый успех можно развить, если уметь подобрать Δm и Δt такие, что $x_1 \approx 0$ и $v_1 \approx 0$ (мягкая посадка). Как это можно сделать?

Фактически требуется решить систему из двух уравнений относительно двух неизвестных:

$$\begin{cases} x(\Delta m, \Delta t) &= 0, \\ v(\Delta m, \Delta t) &= 0. \end{cases}$$

Скалярный решатель можно применить традиционным при «ручном» решении систем уравнений способом: выразить одну

переменную через другую:

$$M(\tau) \in \{\, \mu \mid x(\mu, \tau) = 0 \,\},$$
$$\Delta t \in \{\, \tau \mid v(M(\tau), \tau) = 0 \,\},$$
$$\Delta m = M(\Delta t).$$

Учебное издание

Кувшинов Дмитрий Рустамович
Осипов Сергей Иванович

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык C++

Практикум

Заведующий редакцией	М. А. Овечкина
Редактор	С. Г. Галинова
Корректор	С. Г. Галинова
Компьютерная верстка	Д. Р. Кувшинов

Подписано в печать 01.07.2021 г. Формат $60 \times 84^{1}/_{16}$.
Бумага офсетная. Цифровая печать. Усл. печ. л. 10,5.
Уч.-изд. л. 5,5. Тираж 30 экз. Заказ 112.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: +7 (343) 389-94-79, 350-43-28
E-mail: rio.marina.ovechkina@mail.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620083, Екатеринбург, ул. Тургенева, 4.
Тел.: +7 (343) 358-93-06, 350-58-20, 350-90-13
Факс +7 (343) 358-93-06
http://print.urfu.ru

Для заметок

Для заметок


УНИВЕРСИТЕТСКОГО КНИГОИЗДАНИЯ
2020


9 785799 632564