Лабораторна робота №3

Файли. Використання файлового введення-виведення

Мета роботи: отримати навички створення та реалізації програм, що реалізують операції введення-виведення із файлами.

Загальні відомості

Більшість комп'ютерних програм працюють з файлами, і тому виникає необхідність створювати, видаляти, записувати читати, відкривати файли.

Що ж таке файл? Файл - іменований набір байтів, який може бути збережений на деякому накопичувачі. Тепер зрозуміло, що під файлом мається на увазі деяка послідовність байтів, яка має своє, унікальне ім'я, наприклад file.txt. В одній директорії не можуть знаходитися файли з однаковими іменами. Під іменем файлу розуміється не тільки його назва, а й розширення, наприклад: file.txt та file.dat – різні файли, хоч і мають однакові назви.

Існує таке поняття, як повне ім'я файлу - це повна адреса до директорії файлу із зазначенням імені файлу, наприклад: `D:\docs\file.txt`. Важливо розуміти ці базові поняття, інакше складно буде працювати з файлами.

Файли у мові C++

Для роботи з файлами необхідно підключити заголовний файл `<fstream>`. У `<fstream>` визначені кілька класів і підключені заголовні файли `<ifstream>` – файлове введення та `<ofstream>` – файлове виведення.

Файлове введення / виведення аналогічне стандартному введенню / виведенню, єдина відмінність - це те, що введення / виведення виконаються не на екран, а в файл. Якщо введення / виведення на стандартні пристрої виконується за допомогою об'єктів `cin` і `cout`, то для організації файлового введення / виводу досить створити власні об'єкти, які можна використовувати аналогічно операторам cin і cout.

Наприклад, необхідно створити текстовий файл і записати в нього рядок

```
Робота з файлами в C++.
```

Для цього необхідно виконати наступні кроки:

1. створити об'єкт класу `ofstream`;
2. зв'язати об'єкт класу з файлом, в який проводитиметься запис;
3. записати рядок у файл;

4. закрити файл.

Чому необхідно створювати об’єкт класу `ofstream`, а не класу `ifstream`? Тому, що потрібно зробити запис даних у файл, а якби потрібно було зчитувати дані з файлу, то створювався б об’єкт класу `ifstream`.

```
// створюємо об'єкт для запису у файл
ofstream /*ім'я об'єкта*/; // об'єкт класу ofstream
```

Назвемо об’єкт - `fout`. Ось що вийде:

```
ofstream fout;
```

Для чого нам об’єкт? Об’єкт необхідний, щоб можна було виконувати запис у файл. Тепер вже об’єкт створений, але не пов’язаний з файлом, в який потрібно записати рядок.

```
fout.open("lab3cpp.txt"); // зв'язуємо об'єкт з файлом
```

Через операцію точка отримуємо доступ до методу класу `open()`, в круглих дужках якого вказуємо ім’я файлу. Зазначений файл буде створений в поточній директорії програми. Якщо файл з таким іменем існує, то існуючий файл буде замінений новим. Отже, файл відкритий, залишилося записати в нього потрібний рядок. Робиться це так:

```
fout << "Робота з файлами у C++"; // записуємо рядок у файл
```

Використовуючи операцію передачі в потік спільно з об’єктом `fout` рядок `Робота з файлами у C++` записується в файл. Так як більше немає необхідності змінювати вміст файлу, його треба закрити, тобто відокремити об’єкт від файлу.

```
fout.close(); // закриваємо файл
```

Підсумок - створений файл з рядком `Робота з файлами у C++`

Примітка. Кроки 1 і 2 можна об’єднати, тобто в одному рядку створити об’єкт і пов’язати його з файлом. Робиться це так:

```
ofstream fout("lab3cpp.txt"); // створюємо об'єкт класу ofstream та зв'язуємо його з
файлом lab3cpp.txt
```

Об’єднаємо весь код і отримаємо таку програму

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

int main()
{
    ofstream fout("lab3cpp.txt"); // створюємо об'єкт класу ofstream та зв'язуємо його
з файлом lab3cpp.txt
    fout << "Робота з файлами у C++"; // запис рядку у файл
    fout.close(); // закриваємо файл
    return 0;
}
```

Залишилося перевірити правильність роботи програми, а для цього відкриваємо файл `lab3cpp.txt` і дивимося його вміст, повинно бути – `Робота з файлами у C++`.

Для того щоб прочитати файл знадобиться виконати ті ж кроки, що і при записі у файл з невеликими змінами:

1. Створити об’єкт класу `ifstream` і пов’язати його з файлом, з якого буде проводитися зчитування;
2. Прочитати файл;
3. Закрити файл.

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
    char buff[50]; // буфер проміжного зберігання тексту, що зчитується з файлу
    ifstream fin("lab3cpp.txt"); // відкрили файл для читання

    fin >> buff; // прочитали перше слово з файлу  (*)
    cout << buff << endl; // вивели на екран це слово

    fin.getline(buff, 50); // прочитали рядок з файлу (**)
    fin.close(); // закриваємо файл
    cout << buff << endl; // надрукували цей рядок

    return 0;
}
```

У програмі показані два способи читання з файлу, перший - використовуючи операцію передачі в потік, другий - використовуючи функцію `getline()`. У першому випадку зчитується тільки перше слово, а в другому випадку зчитується рядок, довжиною 50 символів. Але оскільки у файлі залишилося менше 50 символів, то зчитуються символи включно до останнього. Зверніть увагу на те, що зчитування вдруге (рядок помічений **)

продовжилося, після першого слова, а не з початку, так як перше слово було прочитано в рядку поміченому *.

Результат роботи програми показаний нижче

```
Робота
 з файлами в C++
```

Програма спрацювала правильно, але не завжди так буває, навіть у тому випадку, якщо з кодом все в порядку. Наприклад, в програму передано ім'я неіснуючого файлу або в імені допущена помилка. Що тоді? У цьому випадку нічого не відбудеться взагалі. Файл не буде знайдений, а значить і прочитати його не можливо. Тому програма проігнорує рядки, де виконується робота з файлом. В результаті коректно завершиться робота програми, але нічого на екрані показано не буде. Здавалося б це цілком нормальна реакція на таку ситуацію. Але простому користувачеві не буде зрозуміло, в чому справа і чому на дисплеї не з'явився рядок з файлу. Так ось, щоб все було максимально зрозуміло в C++ передбачена така функція – `is_open()`, яка повертає цілі значення: 1 – якщо файл був успішно відкритий, 0 – якщо файл відкритий не був.

Доопрацюємо програму з відкриттям файлу, таким чином, щоб у випадку якщо файл не відкритий, виводилося відповідне повідомлення.

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
    char buff[50]; // буфер тимчасового зберігання тексту, що читається з файлу
    ifstream fin("lab3cpp.doc"); // (ВВЕЛИ НЕКОРЕКТНЕ ІМ'Я ФАЙЛУ)

    if (!fin.is_open()) // якщо файл не выдкрито
        cout << "Файл не може бути відкрито!\n"; // повідомити про це
    else
    {
    fin >> buff; // прочитали перше слово з файлу
    cout << buff << endl; // вивели це слово

    fin.getline(buff, 50); // прочитали рядок з файлу
    fin.close(); // закриваємо файл
    cout << buff << endl; // виводимо цей рядок
    }

    return 0;
}
```

Результат роботи програми показаний нижче

```
Файл не може бути відкрито!
```

Як видно з результату роботи програми, вона повідомила про неможливість відкрити файл. Тому, якщо програма працює з файлами, рекомендується використовувати цю функцію, `is_open()`, навіть, якщо впевнені, що файл існує.

Режими відкриття файлів

Режими відкриття файлів встановлюють характер використання файлів. Для установки режиму в класі `ios_base` передбачені константи, які визначають режим відкриття файлів

Константа	**Описання**
ios_base::in	відкрити файл для читання
ios_base::out	відкрити файл для запису
ios_base::ate	при відкритті перемістити покажчик в кінець файлу
ios_base::app	відкрити файл для запису в кінець файлу
ios_base::trunc	видалити вміст файлу, якщо він існує
ios_base::binary	відкриття файлу в двійковому режимі

Режими відкриття файлів можна встановлювати безпосередньо при створенні об'єкта або при виконанні функції `open()`

```
ofstream fout("lab3cpp.txt", ios_base::app); // открываем файл для добавления
информации к концу файла
fout.open("lab3cpp.txt", ios_base::app); // открываем файл для добавления информации к
концу файла
```

Режими відкриття файлів можна комбінувати з допомогою поразрядной логічної операції `або |`, наприклад: `ios_base::out | ios_base::trunc` – відкриття файлу для запису, попередньо очистивши його.

Об'єкти класу `ofstream`, при зв'язці з файлами за замовчуванням містять режими відкриття файлів `ios_base::out | ios_base::trunc`. Тобто файл буде створений, якщо не існує. Якщо ж файл існує, то його вміст буде видалено, а сам файл буде готовий до запису. Об'єкти класу `ifstream` зв'язуючись з файлом, мають за замовчуванням режим відкриття файлу `ios_base::in` – файл відкритий тільки для читання. Режим відкриття файлу ще називають – "прапорець", для зручності надалі використовується саме цей термін.

Зверніть увагу на те, що прапорці `ate` і `app` за описом дуже схожі, вони обидва переміщують покажчик в кінець файлу, але прапор app дозволяє проводити запис, тільки в кінець файлу, а прапор ate просто переставляє прапор в кінець файлу і не обмежує місця запису.

Розробимо програму, яка, використовуючи операцію `sizeof()`, буде обчислювати характеристики основних типів даних в C++ і записувати їх у файл.

Характеристики:

- число байт, відводиться під тип даних
- максимальне значення, яке може зберігати певний тип даних.

```
#include <iostream>
#include <fstream> // робота з файлами
#include <iomanip> // маніпулятори введення/виведення
using namespace std;

int main()
{
    // зв'язуємо об'єкт з файлом, при цьому файл відкриваємо в режимі запису,
попередньо видаливши всі дані з нього
    ofstream fout("data_types.txt", ios_base::out | ios_base::trunc);

    if (!fout.is_open()) // якщо файл не було відкрито
    {
     cout << "Файл не може бути відкритим або створеним\n";
     return 1;
    }

        fout << "     data type      " << "byte" << "      "
             << "    max value  " << endl // заголовки стовпців
             << "bool               =  " << sizeof(bool) << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних bool*/
             << (pow(2,sizeof(bool) * 8.0) - 1) << endl
             << "char               =  " << sizeof(char) << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних char*/
             << (pow(2,sizeof(char) * 8.0) - 1) << endl
             << "short int          =  " << sizeof(short int) << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних short int*/
             << (pow(2,sizeof(short int) * 8.0 - 1) - 1) << endl
             << "unsigned short int =  " << sizeof(unsigned short int)  << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних unsigned short int*/
             << (pow(2,sizeof(unsigned short int) * 8.0) - 1) << endl
             << "int                =  " << sizeof(int)                 << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних int*/
             << (pow(2,sizeof(int) * 8.0 - 1) - 1) << endl
             << "unsigned int       =  " << sizeof(unsigned int) << "         "
             << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних unsigned int*/
             << (pow(2,sizeof(unsigned int) * 8.0) - 1) << endl
             << "long int           =  " << sizeof(long int) << "         "
             << fixed << setprecision(2)
```

```
/*обчислюємо максимальне значення для типу даних long int*/
            << (pow(2,sizeof(long int) * 8.0 - 1) - 1) << endl
            << "unsigned long int  =  " << sizeof(unsigned long int) << "         "
            << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних long int*/
            << (pow(2,sizeof(unsigned long int) * 8.0) - 1) << endl
            << "float              =  " << sizeof(float) << "          "
            << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних float*/
            << (pow(2,sizeof(float) * 8.0 - 1) - 1) << endl
            << "long float         =  " << sizeof(long float) << "          "
            << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних long float*/
            << (pow(2,sizeof(long float) * 8.0 - 1) - 1) << endl
            << "double             =  " << sizeof(double) << "          "
            << fixed << setprecision(2)
/*обчислюємо максимальне значення для типу даних double*/
            << (pow(2,sizeof(double) * 8.0 - 1) - 1) << endl;
        fout.close();
// програма більше не використовує файл, тому його треба закрити
    cout << "Дані успішно записані у файл data_types.txt\n";
    return 0;
}
```

Можна помітити, що стандартне введення / виведення і файлове введення / виведення використовуються абсолютно аналогічно. В кінці програми ми явно закрили файл, хоча це й не обов'язково при використанні деяких компіляторів, але вважається хорошим тоном програмування та необхідно для сумісності з різноманітними версіями компіляторів C++. Варто відзначити, що всі функції та маніпулятори використовувані для форматування стандартного введення / виведення актуальні і для файлового введення / виведення.

Файли у мові Kotlin

Запис у файл в Kotlin

Kotlin пропонує різні способи запису у файл у вигляді методів розширення для java.io.File.

Ми використаємо кілька з них, щоб продемонструвати різні способи запису у файли в Kotlin

- writeText – дозволяє писати в файл напряму з рядків String
- writeBytes – використовується для запису напряму з масивів байтів ByteArray
- printWriter – надає нам об'єкт PrintWriter із широкими можливостями
- bufferedWriter – надає можливість запису із використанням об'єкту BufferedWriter

Запис "напряму"

`writeText`

Можливо, найпростіший метод розширення, `writeText` приймає вміст як аргумент `String` і

записує його безпосередньо у вказаний файл. Даний вміст записується із використанням кодування у форматі UTF-8 (за замовчуванням) або будь-яким іншим вказаним кодуванням:

```
File(fileName).writeText(fileContent)
```

Цей метод внутрішньо викликає `writeBytes`, як буде описано нижче. Але спочатку він перетворює заданий вміст у масив байтів, використовуючи вказане кодування

`writeBytes`

Метод writeBytes бере аргумент типу `ByteArray` і безпосередньо записує його у вказаний файл. Це корисно, коли ми маємо вміст як масив байтів, а не як звичайний текст.

```
File(fileName).writeBytes(fileContentAsArray)
```

Якщо даний файл існує, він перезаписується.

Запис у файл за допомогою Writers

Kotlin також пропонує методи розширення, які надають нам об'єкт типу Java Writer

`printWriter`

Якщо ми хочемо використовувати Java `PrintWriter`, Kotlin надає функцію printWriter саме для цієї мети. За допомогою нього ми можемо виводити відформатовані представлення об'єктів у вихідний потік (OutputStream):

```
File(fileName).printWriter()
```

Ця функція (метод) повертає новий екземпляр `PrintWriter`. Далі ми можемо скористатися використанням методу `use` для роботи з ним:

```
File(fileName).printWriter().use { out -> out.println(fileContent) }
```

За допомогою методу (функції) `use` ми можемо виконати функцію на ресурсі, яка закривається після завершення. Ресурс (в даному випадку - файл) буде закрито незалежно від того, успішно виконана функція чи вона призвела до помилки.

`bufferedWriter`

Подібним чином Kotlin також надає функцію `bufferedWriter`, яка надає нам об'єкт Java `BufferedWriter`.

```
File(fileName).bufferedWriter()
```

Подібно до `PrintWriter`, ця функція повертає новий екземпляр `BufferedWriter`, який пізніше

ми можемо використовувати для запису вмісту файлу.

```
File(fileName).bufferedWriter().use { out -> out.write(fileContent) }
```

Розглянемо приклад запису у файл повідомлення "Hello, Kotlin!"

```
import java.io.File

fun main() {
    val writer = File("lab3kt.txt").printWriter()
    writer.println("Hello, Kotlin!")
    writer.close()
}
```

або із використанням функції `use`:

```
import java.io.File

fun main() {
    File("lab3kt.txt").printWriter().use {
        it.println("Hello, Kotlin!")
    }
}
```

Читання з файлу в Kotlin

Спершу створимо вхідний файл, який буде читати програма мовою Kotlin. Ми створюємо файл під назвою example.txt і зберігаємо його в каталог, до якого можна отримати доступ за допомогою нашого коду.

Вміст файлу може бути, наприклад, таким:

```
Hello from Kotlin! It's:
1. Concise
2. Safe
3. Interoperable
4. Tool-friendly
```

Розглянемо різні способи, за допомогою яких ми можемо прочитати цей файл. Ми повинні передати повний шлях до файлу, створеного вище, як вхідні дані для таких методів, або відносний шлях відносно розміщення програми.

`forEachLine`

Читає файл рядок за рядком, використовуючи вказану кодову таблицю (за замовчуванням UTF-8), і викликає дію для кожного рядка:

```
File(fileName).forEachLine { println(it) }
```

В цьому прикладі програма виводить на екран вміст файлу "рядок за рядком". Замість `println(it)` може бути будь-яка дія, чи декілька дій, де `it` - рядок, що прочитаний з файлу.

`useLines`

Викликає вказану операцію "зворотного виклику" у блоку, надаючи йому послідовність усіх рядків у файлі.

Після завершення обробки файл закривається:

```
val listOfLines : List<String> = File(fileName).useLines { it.toList() }
```

Таким чином, елементами списку будуть рядки вхідного файлу

`bufferedReader`

Повертає новий об'єкт `BufferedReader` для читання вмісту файлу.

Отримавши `BufferedReader`, ми можемо прочитати всі рядки в ньому:

```
val listOfLines : List<String> = File(fileName).bufferedReader().readLines()
```

Також, використовуючи BufferedReader можна організувати "більш традиційний" процес обробки файлу у циклі.

```
var line: String?
while (reader.readLine().also { line = it } != null) {
    println(line)
}
reader.close()
```

або із використанням `use`:

```
var line: String?
File("lab3kt.txt").bufferedReader().use { reader ->
    while (reader.readLine().also { line = it } != null) {
        println(line)
    }
}
```

`readLines`

Безпосередньо читає вміст файлу у список рядків:

```
val listOfLines : List<String> = File(fileName).readLines()
```

Примітка. *Цей метод не рекомендується використовувати для величезних файлів*

`inputStream`

Створює новий об'єкт `FileInputStream` для файлу і повертає його в результаті.

Отримавши вхідний потік, ми можемо перетворити його в байти, а потім у звичайний рядок:

```
val s : String = File(fileName).inputStream().readBytes().toString(Charsets.UTF_8)
```

`readText`

Зчитує весь вміст файлу як рядок із зазначеним кодуванням (за замовчуванням UTF-8):

```
val s : String = File(fileName).readText(Charsets.UTF_8)
```

Примітка. *Цей метод не рекомендується використовувати для величезних файлів і має внутрішнє обмеження у 2 ГБ.*

`Scanner`

Для введення даних, як з клавіатури, так і з файлу можна використовувати клас Scanner зі стандартної бібліотеки Java.

```
val scanner = Scanner(File(fileName))
val intValue = scanner.nextInt() // зчитування цілого числа
val doubleValue = scanner.nextDouble() // зчитування дійсного чила типу Double
// ... аналогічно для інших типів даних
```

Варіанти завдань до лабораторної роботи

Для виконання завдань 3.1 та 3.2 розробіть по дві програми, що розв'язують ці завдання: мовою C++ та мовою Kotlin. Порівняйте зручність інструментів цих мов програмування, що використовуються для роботи з файлами.

Завдання 3.1. Розробити програму, що виконує обчислення, використовуючи дані, що знаходяться у файлі.

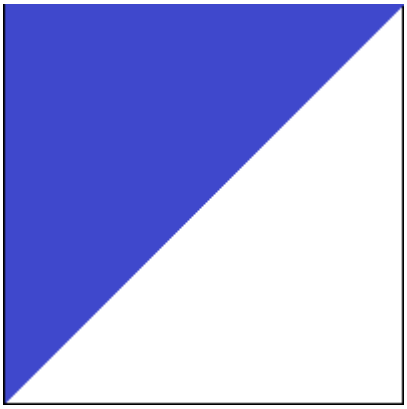
У всіх варіантах дано файл `f.txt`, компоненти якого – дійсні числа. (*Файл підготувати самостійно. Кількість елементів у файлі не менше 20*).

Варіант	Завдання
1	Знайти суму компонентів файлу
2	Знайти добуток компонентів файлу
3	Знайти суму квадратів компонентів файлу
4	Знайти модуль суми та квадрат добутку компонентів файлу
5	Знайти останній компонент файлу.
6	Знайти найменше значення серед компонентів файлу
7	Знайти найменше значення серед компонентів файлу з парними номерами
8	Знайти найбільше значення серед компонентів файлу з непарними номерами
9	Знайти суму найбільшого та найменшого компонентів файлу
10	Знайти різницю першого та останнього компонентів файлу
11	Знайти суму компонентів файлу з непарними номерами
12	Знайти суму від'ємних компонентів файлу
13	Знайти кількість від'ємних компонентів з непарними номерами
14	Знайти суму компонентів, ціла частина яких кратна 3.
15	Знайти суму компонентів між першим та останнім від'ємними
16	Знайти кількість компонентів в файлі
17	Знайти суму першого від'ємного та останнього додатного компонентів файлу
18	Знайти добуток всіх від'ємних компонентів файлу
19	Знайти суму квадратів всіх компонентів файлу.
20	Знайти середнє арифметичне квадратів тих компонентів файлу, які не дорівнюють нулю.

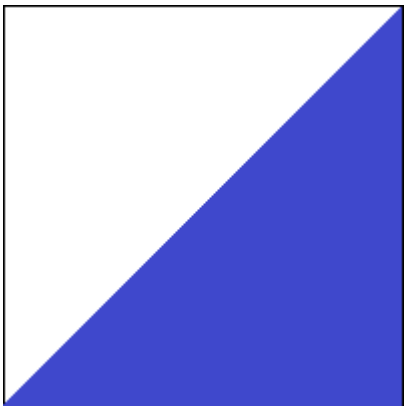
Завдання 3.2. Розробити програму, що зчитує з файлу `z32.txt` елементи двовимірного масиву цілих чисел розміром NxN елементів та опрацьовує його згідно варіанту.

Примітка. *Перший рядок файлу містить одне ціле число N. Наступні N рядків містять по N дійсних чисел – елементи заданого масиву. Файл можна отримати у викладача, або завантажити з сайту http://berkut.mk.ua у розділі «Алгоритмизация и программирование»*

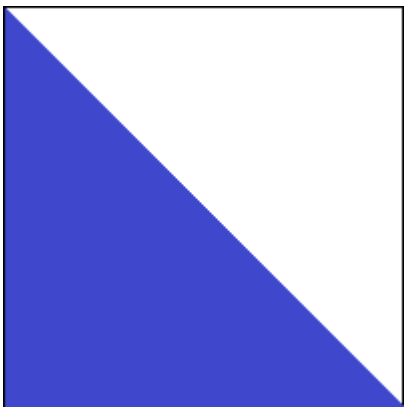
Варіант 1 Знайти суму елементів заштрихованої частини



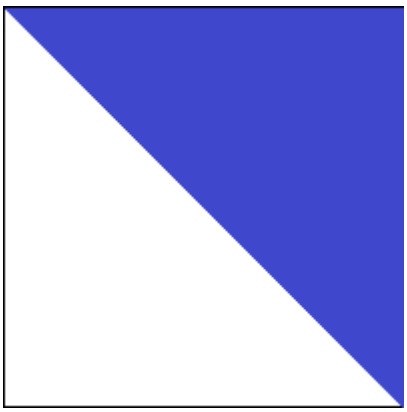
Варіант 2 Знайти індекси і значення найбільшого елемента заштрихованої частини



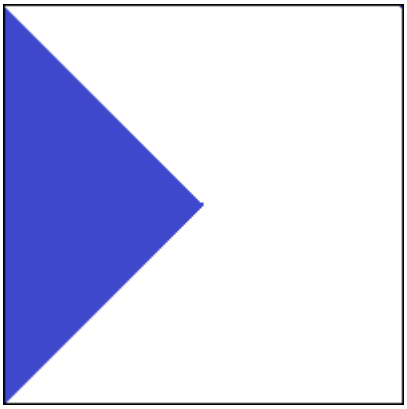
Варіант 3 Знайти суму модулів елементів заштрихованої частини



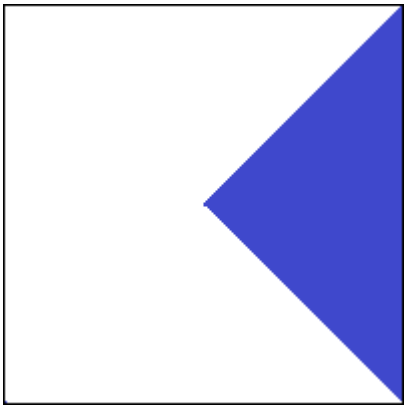
Варіант 4 Знайти індекси і значення найменшого елемента заштрихованої частини



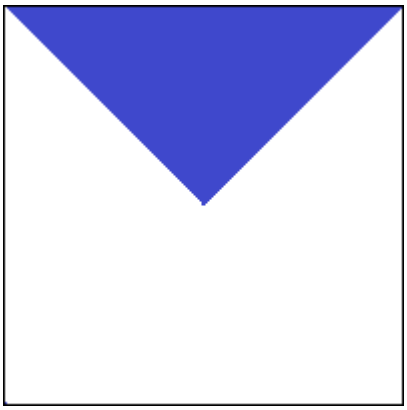
Варіант 5 Знайти суму від’ємних елементів заштрихованої частини



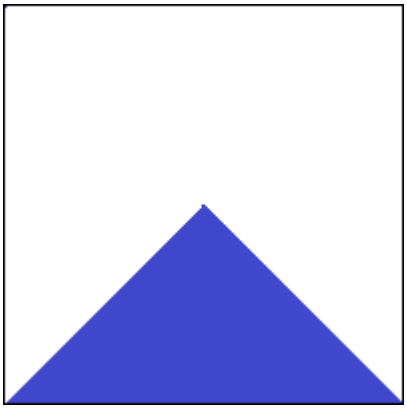
Варіант 6 Знайти індекси і значення найбільшого парного елемента заштрихованої частини



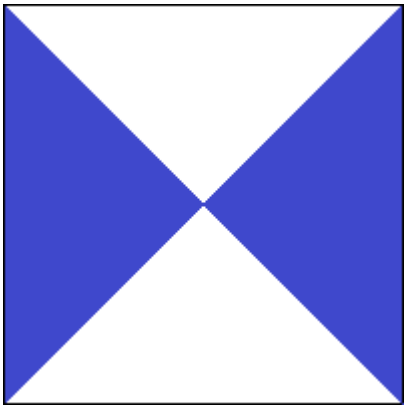
Варіант 7 Знайти суму додатних елементів заштрихованої частини



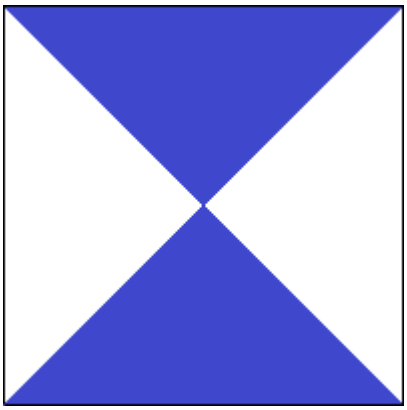
Варіант 8 Знайти суму непарних елементів заштрихованої частини



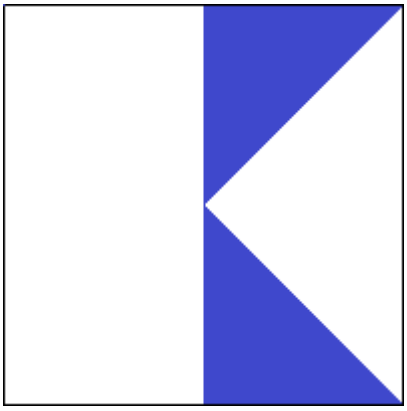
Варіант 9 Знайти кількість від’ємних елементів та суму додатних елементів заштрихованої частини



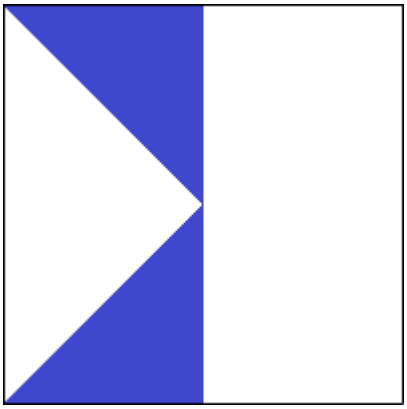
Варіант 10 Знайти суму непарних додатних елементів заштрихованої частини



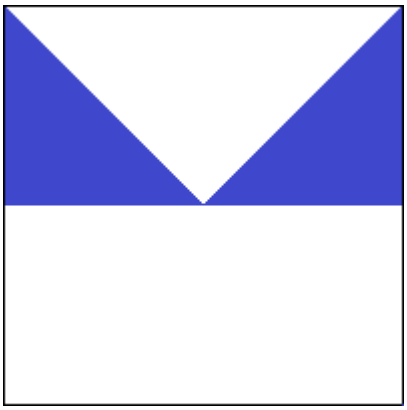
Варіант 11 Знайти індекси найбільшого від’ємного та найменшого додатного елементів заштрихованої частини



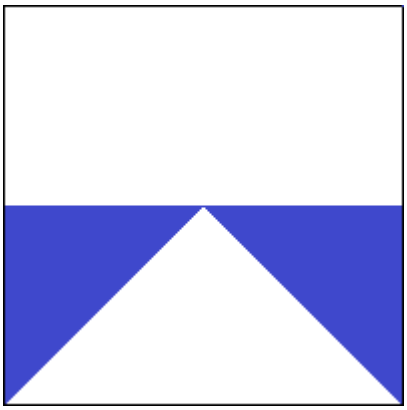
Варіант 12 У заштрихованій частині знайти кількість елементів, що відрізняються від найменшого елемента не більше ніж на 10%



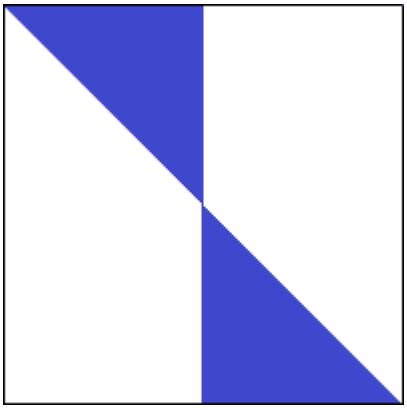
Варіант 13 Знайти суму елементів заштрихованої частини, що діляться на 3



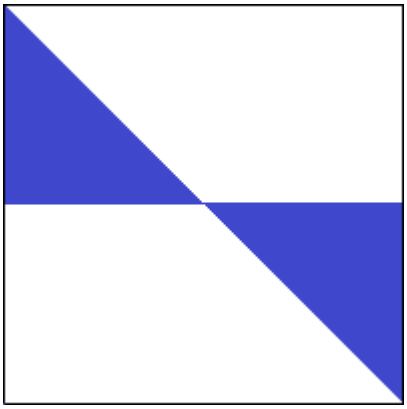
Варіант 14 У заштрихованій частині знайти кількість елементів, що відрізняються від найбільшого елемента не більше ніж на 10%



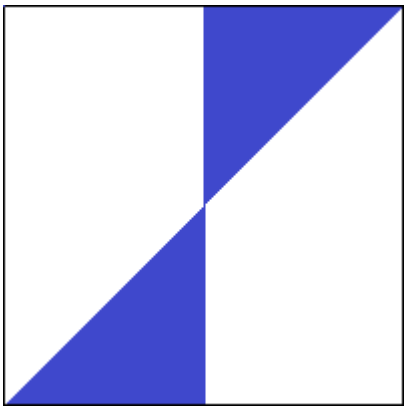
Варіант 15 Знайти різницю найбільшого та найменшого елементів у заштрихованій частині



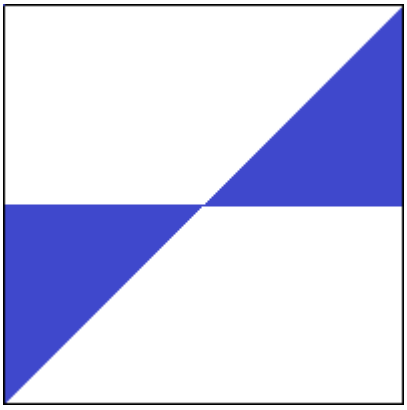
Варіант 16 Знайти суму елементів заштрихованої частини, кратних 5



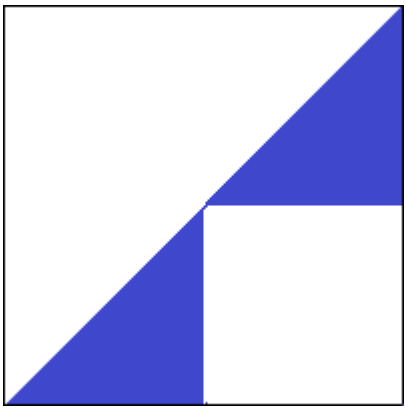
Варіант 17 Обчислити середнє арифметичне від’ємних елементів заштрихованої частини



Варіант 18 Обчислити середнє арифметичне додатних елементів заштрихованої частини



Варіант 19 Знайти суму логарифмів модулів елементів заштрихованої частини



Варіант 20 Знайти суму квадратів елементів заштрихованої частини