Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 1 з дисципліни
«Мультипарадигменне Програмування»

„ **Імперативне програмування**”

Виконав(ла) *ІП-02 Науменко Р. О.*
(шифр, прізвище, ім'я, по батькові)

Перевірив _____
(прізвище, ім'я, по батькові)

Київ 2022

Зміст

1 Завдання

Практична робота складається із трьох завдань, які самі по собі є досить простими. Але, оскільки задача - зрозуміти, як писали код наші славні пращури у 1950-х, ми введемо кілька обмежень:

- Заборонено використовувати функції
- Заборонено використовувати цикли
- Для виконання потрібно взяти мову, що підтримує конструкцію GOTO

Завдання 1:

Обчислювальна задача тут тривіальна: для текстового файлу ми хочемо відобразити N (наприклад, 25) найчастіших слів і відповідну частоту їх повторення, упорядковано за зменшенням. Слід обов’язково нормалізувати використання великих літер і ігнорувати стоп-слова, як «the», «for» тощо. Щоб все було просто, ми не піклуємося про порядок слів з однаковою частотою повторень. Ця обчислювальна задача відома як **term frequency**.

Ось такий вигляд матимуть ввід і відповідно вивід результату програми:

Input:

```
White tigers live mostly in India
Wild lions live mostly in Africa
```

Output:

```
live - 2
mostly - 2
africa - 1
india - 1
lions - 1
tigers - 1
white - 1
wild - 1
```

Завдання 2:

Тепер, нам потрібно виконати задачу, що називається словниковим індексуванням. Для текстового файлу виведіть усі слова в алфавітному порядку разом із номерами сторінок, на яких Ці слова знаходяться. Ігноруйте всі слова, які зустрічаються більше 100 разів. Припустимо, що сторінка являє собою послідовність із 45 рядків. Наприклад, якщо взяти книгу Pride and Prejudice, перші кілька записів індексу будуть:

```
abatement - 89
abhorrence - 101, 145, 152, 241, 274, 281
abhorrent - 253
abide - 158, 292
```

2 Виконання

Обов’язковою умовою в цій лабораторній роботі була відсутність функцій та циклів. Вочевидь зчитати дані з файлів, не використовуючи деякі вбудовані функції, було б невиправдано складно, тому я прийняв рішення використовувати функцію getline з модулями <string> та <fstream>. Також, при роботі з об’єктами класа string, я використовував методи size() та empty().

2.1 Завдання №1

Це завдання можна поділити на декілька частин:

- Зчитування вхідних даних з файлів (сам текст та список стоп слів).
- Збереження даних про зчитані слова та їх частоту.
- Сортування слів за частотою
- Виведення результату.

Зчитування даних з файлів було порядкове:

```
Reading: // reading file line by line
     i = 0;
     if (!std::getline(inFile, line)) goto _Reading;
```

```
     word = "";
     goto ParsingLine;
     _ParsingLine:

     goto Reading;
_Reading:
```

Зациклення в коді було досягнуто за допомогою переходу на мітку Reading. Якщо ж зчитування файлу завершено, то getline повертає false, та виконання програми продовжується з мітки _Reading.

Додавання нових та оновлення кількості старих слів у словнику відбувалось у два етапи:

1. Пошук слова у словнику (findWord)
2. Оновлення інформацію про нього, або додавання, якщо слова ще немає у словнику. (UpdatingVocabulary)

```
UpdatingVocabulary: // adding new or updating existing
word count
     wordPosition = -1;
     goto findWord;
     _findWord:
     if (wordPosition == -1){
          vocabulary[wordsCount] = {1, word};
          wordsCount++;
     }
     else {
          vocabulary[wordPosition].count++;
     }
     goto _UpdatingVocabulary;

findWord: // searching for word in vocabulary
     j = 0;
     loop:
          if (vocabulary[j].word == word) {
```

```
            wordPosition = j;
            j = wordsCount;
        }
        j++;
        if (j < wordsCount) goto loop;
    goto _findWord;
```

Для сортування було використано тривіальний алгоритм bubble sort

```
Sorting: // Sorting vocabulary
OutterLoop:
    in = 1;
InnerLoop:
    if (vocabulary[in - 1].count < vocabulary[in].count)
    {
        Token temp = vocabulary[in - 1];
        vocabulary[in - 1] = vocabulary[in];
        vocabulary[in] = temp;
    }
    in++;
    if (in < wordsCount - out) goto InnerLoop;
    out++;
    if (out < wordsCount - 1) goto OutterLoop;
_Sorting:
```

При виведені результату (PrintingResult) відбувається перевірка належності слова до стоп слів (CheckStopWord).

```
     PrintingResult: // printing resulting tokens: "word -
     ???"
        wordToPrint++;
        isStopWord = false;
        i = 0;
        goto CheckStopWord;
        _CheckStopWord:

        if (!isStopWord)
```

```
    {
        topN--;
        std::cout << vocabulary[wordToPrint].word <<
" - " << vocabulary[wordToPrint].count << std::endl;
    }

    if (wordToPrint < wordsCount && topN) goto
PrintingResult;
_PrintingResult:

CheckStopWord: // Check if word is not a stop word
before printing it
    if (vocabulary[wordToPrint].word == stopWords[i])
    {
        isStopWord = true;
        goto _CheckStopWord;
    }
    i++;
    if (i < stopWordsCount) goto CheckStopWord;
    goto _CheckStopWord;
```

2.2 Завдання №2

Відмінність цього завдання від першого полягає лише в алгоритмі додавання слів у словник та виведені результатів.

```
UpdatingVocabulary: // adding new or updating existing
word count

     wordPosition = -1;

     goto findWord;

     _findWord:

     if (wordPosition == -1) {

          vocabulary[wordsCount] = { 1, 1, new int[100],
          word };

          vocabulary[wordsCount].pages[0] = currentPage;

          wordsCount++;

     }

     else {

          vocabulary[wordPosition].count++;

          if (vocabulary[wordPosition].count <= 100) {

               if
               (vocabulary[wordPosition].pages[vocabulary[w
               ordPosition].lastPage - 1] != currentPage)

               {

                    vocabulary[wordPosition].pages[vocabula
                    ry[wordPosition].lastPage] =
                    currentPage;

                    vocabulary[wordPosition].lastPage++;

               }

               vocabulary[wordPosition].count++;
```

```
        }
    }
    goto _UpdatingVocabulary;
```

Тут ми перевіряли, чи не перевищила кількість входжень слова 100, та зберігали сторінки, на яких воно зустрічається.

Висновок

В рамках даної лабораторної роботи я познайомився з принципами імперативного програмування та створив дві програми без використання функцій, циклів та з використанням оператора стрибку goto. Зробивши це, я можу зробити висновок, що goto - це досить потужний інструмент, що може відчутно пришвидшити роботу програми, при цьому не сильно ускладнюючи її читаємість. Але лише за умови комбінування його із сучасними засобами, такими як: функції та цикли. В іншому випадку об’єм коду стає незрівнянно великим а його складність зростає в рази.