2021.04.09

Требования и рекомендации по заданию 3 (1 курс, 2 семестр)

Третье задание — обработка текстов.

Суть задания — Манипуляции с текстовым файлом с использованием библиотечных функций работы со строками (string.h или подобных).

Для простоты в данном варианте заданий считаем, что текст не содержит русских букв (и других национальных алфавитов), и, следовательно, каждый печатный символ представляется одним байтом (char).

Как правило, программа получает на входе некоторый файл и на выходе формирует другой файл, представляющий собой решение задачи (т.е. исходный файл не меняется).

Решение задач может быть реализовано одним из двух общих подходов.

1. Символьный подход. Программа считывает в цикле по одному символу из исходного файла, анализирует прочитанный символ и выполняет требуемую работу, формируя необходимые символы вывода. Возможно, для решения некоторых задач потребуется накапливать некоторый внутренний буфер (массив) из нескольких прочитанных символов, чтобы осуществить требуемый анализ файла. Таким образом, программа последовательно считывает символы и в зависимости от своего внутреннего состояния и очередного прочитанного символа выдает на выход требуемые символы результата.

2. Строчный подход. Программа считывает из файла в свой внутренний буфер (массив) сразу одну текущую строку. Анализирует эту строку и формирует требуемый вывод. Преимущество этого метода состоит в том, что можно многократно просматривать введенную строку и иметь доступ к любым позициям этой строки. Однако возникает проблема определения и контроля длины буфера для сохранения строки, имея в виду, что в файле могут встретиться строки, превышающие размеры буфера. Таким образом, программа считывает строку, анализирует и как-то преобразовывает эту строку (возможно, по частям), и потом выдает результат на выход.

Какой из этих двух методов выгоднее использовать на практике зависит от специфики конкретной задачи.

Для анализа строк и ввода/вывода тестовых данных нужно по возможности пользоваться библиотечными процедурами. Наиболее простые процедуры стандартной библиотеки clib подобного типа объявлены в заголовочном файле string.h. Эти процедуры позволяют выполнять поиск символов и подстрок в данной строке, определять различные характеристики строк и т.п. (см. описания функций string.h в любом руководстве).

В языке C++ есть специальный тип string, который также реализует работу со строками. По сути он повторяет функциональность библиотеки C (string.h), скрывая от пользователя операции перевыделения памяти при изменении размеров строки. Поэтому код C++ может быть чуть проще для реализации, но, как правило, работает он медленнее, чем аккуратно написанный код C.

Полезные замечания.

Библиотеки C работают с так называемым типом ANSI string. Это есть массив байтов (char), который заканчивается нулевым байтом (значение есть 0). Нулевой байт служит признаком конца строки. Все библиотечные процедуры используют это соглашение, и строки, которые они выдают, всегда оканчиваются нулем. Соответственно и все строки, поступающие на вход этих функций, должны иметь завершающий нулевой байт. Если на вход будет подана строка (массив) без завершающего

нулевого байта, то функции могут начать просмотр памяти до обнаружения такого байта, и, как результат, выйти за границы массивов и испортить значения других переменных или вызвать отказ по доступу к памяти (segmentation fault). Поэтому при работе с ANSI строками надо всегда предусматривать место в массиве байтов для размещения завершающего нуля и не забывать его туда реально записывать.

По историческим причинам в Windows и Unix системах используются разные соглашения об обозначении перехода на новую строку (символ $\backslash n$). Так, в Unix системах для этого используется один байт со значением 0xA (т.е. десятичное 10). В Windows системах конец строки обозначается двумя байтами 0xD 0xA (десятичные 13 и 10). Таким образом, текстовые файлы с идентичным текстом, подготовленные в разных системах, могут (и будут) отличаться по своему внутреннему представлению переходов на следующую строку. Утилиты, работающие с текстами в этих системах, обычно умеют определять способ кодирования концов строк и отображают текст с правильным переходом от строки к строке. Однако в некоторых случаях отображение текста может оказаться и несколько "испорченным", например, в конце строк могут появиться дополнительные значки (windows текст при отображении в unix) или новые строки не будут прижиматься к началу страницы, а будут выстраиваться лесенкой (unix текст в windows системе). При написании своих программ этот факт надо иметь в виду и аккуратно проверять как именно закодирован переход на новую строку в обрабатываемом тексте, чтобы программа могла правильно обработать текст, подготовленный в любой системе.

Как уже говорилось выше, строчный подход к обработке текста предполагает выделение отдельного буфера для сохранения очередной прочитанной строки. Таким образом, перед чтением строки надо определиться с размером этого буфера. Если очередная строка в файле окажется больше размера буфера и чтение этой строки будет выполнено недостаточно аккуратно, то буфер может переполниться с соответствующими фатальными последствиями для вашей программы (искажение других переменных или segmentation fault). Таким образом, нужно использовать для чтения только те процедуры, которые позволяют контролировать размер прочитанных данных (например, fgets). Если строка окажется прочитанной не полностью, то, исходя из смысла задачи, можно либо анализировать строку по частям, либо попытаться создать новый буфер и прочитать в него последующие части строки. Аккуратная реализация подобного процесса чтения может оказаться не совсем тривиальной задачей.

В некоторых случаях в качестве упрощения можно задавать буфер некоторого "достаточно большого" размера, надеясь, что все строки файла в него поместятся (например, с использованием инструкции типа `#define BUFSIZE 1024`). При появлении более длинной строки, программа может, например, выдать сообщение о невозможности обработать такой файл и закончить работу. Это может рассматриваться как тестовый вариант для того, чтобы на время перенести внимание на реализацию и отладку других частей алгоритма. В окончательном варианте программа должна все же быть реализована для работы со строками произвольной длины.

Реализация программы не должна зависеть от размеров самого файла и также должна быть достаточно "экономной". Например, нельзя сохранять весь файл целиком во внутреннем буфере программы, поскольку такой буфер может оказаться очень большим (соответственно размеру входного файла). Трудоемкость программы также должна, как правило, линейно зависеть от размеров входного файла.

Общие требования к отчету по задачам обработки текстов.

Решение каждой задачи должно содержать

— файлы программы с инструкцией как их скомпилировать и как запустить тесты;

— набор тестовых файлов, т.е. несколько файлов, которые должны обрабатываться программой; каждый файл иллюстрирует определенные особенности задачи и возможности алгоритма; тестирующая программа последовательно запускает обработку этих файлов;

— описание тестов, т.е. какие именно ситуации должен проиллюстрировать или проверить каждый отдельный тест, какие ограничения имеет данная версия программы (т.е. какие ситуации она обрабатывает, а какие не может).

Комментарии по отдельным задачам.

1. *Заменить в файле каждую последовательность заданных одинаковых символов на один такой символ.*

Легко решается в рамках символьного подхода. Заданные символы сохраняем в отдельной строке. Читаем посимвольно исходный файл и сохраняем предыдущий прочитанный символ. Если текущий и предыдущий символы совпадают и присутствуют в строке заданных символов, то не выводим текущий символ, иначе выводим (т.е. копируем содержимое файла).

2. *Заменить всюду в файле один заданный набор символов на другой (с учетом разницы в их длине).*

Это классическая задача — поиск заданного образца в строке. Известно несколько алгоритмов (поиск в помощь), но в данном варианте задания не требуется особой изощренности, главное — решить требуемую задачу "как-нибудь", но с правильным результатом. Простейший вариант — чтение файла о строкам, поиск образца в строке с помощью библиотечных функций, вывод части строки до образца, вывод требуемой замены вместо образца и повторение этой процедуры для оставшейся части строки.

3. *Вывести все слова из данного файла в другой файл в порядке их появления по одному слову на строке.*

Фактически надо разделить содержимой файла на "слова" и "не слова (разделители)". В такой постановке нужно предусмотреть задание символов-разделителей, которые мы не будем относить к словам. Например, это знаки препинания, цифры, переводы строк и т.д. В зависимости от того, какие символы назначены быть разделителями, программа будет давать разные результаты. Программа достаточно просто реализуется как в символьном, так и в строчном вариантах.

4. *Вывести номера строк исходного файла и номер позиции в строке, где встречается заданное слово.*

Это задача построения аналога "индексного указателя", который присутствует во многих книгах. Фактически задача использует те же известные подходы к поиску подстроки, но дополнительно отслеживает номер строки и номер позиции в строке.

5. *Определить максимальную, минимальную и среднюю длину слов из данного файла а также частоту (процент) появления каждого символа.*

Очень простая задача. Можно завести таблицу на 256 элементов — счетчики количества появлений каждого байта во входной последовательности символов (это для посчета частот). Кроме этого нужно определить символы-разделители слов и отслеживать переходы последовательности с разделительного символа на неразделительные (т.е. моменты появления нового слова или окончания текущего слова).

6. *Вывести все слова из данного файла в алфавитном порядке.*

По поводу выделения слов в файле см. задачу выше, но теперь эти слова надо еще сохранять в массиве для последующей сортировки. Этот массив можно реализовать как массив указателей char*, где каждый элемент массива указывает на массив байтов отдельного слова. Такая конструкция аналогична представлению матрицы байтов через двойной указатель, только теперь каждая строка этой матрицы соответствует слову и будет иметь различную длину соответственно длине слова с завершающим нулевым символом. Возникает вопрос о длине этого массива указателей (по количеству слов в файле). Для начального варианта можно задать это количество константой (и проверять не превысило ли фактическое количество слов эту константу), а в окончательном варианте можно написать отдельную процедуру, подсчитывающую количество слов в файле (по аналогии с вводом числового массива из файла). Для сортировки можно использовать библиотечную функцию qsort, снабдив ее процедурой сравнения двух строк на основе функции strcmp (нужно только привести к правильному виду типы аргументов этих двух функции).

Интересно, что в больших литературных текстах большинство слов встречается всего несколько раз, а слов, которые встречаются больше, чем 3-4 раза всего около 10–15% от общего количества. Можно проверить это утверждение на реальных текстах.

7. *Сравнить два файла на добавленные/удаленные строк.*

Файл B получен из файла A добавлением и/или удалением некоторого малого количества строк (например, не больше $k = 10$). Требуется выписать "команды", с помощью которых файл A преобразуется в файл B. Эти команды — удаление и вставка строки — должны иметь примерно такой вид:

```
insert N1 текст вставленной строки
delete N2
```

и.т.д. Где $N1, N2, ...$ номера строк файла A, где новая строка вставляется или старая строка удаляется. При указании номеров строк надо иметь в виду, что вставка или удаление одной строки меняет нумерацию других последующих строк, поэтому надо точно определиться к какому состоянию файла относятся эти номера. Метод решения может описаться на алгоритм наибольшей общей подпоследовательности, но может быть реализован и "по-простому", если считать, что новые вставленные строки не совпадают с уже существующими. В это последнем случае можно просто определить какие строки в файлах совпадают, и потом естетсвенным образом получить множество удаленных строк (из файла A) и додавленных строе (из файла B).

8. *Ввод данных из таблицы.*

Предполагается, что в файле записана некоторая таблица (матрица) данных, снабженная заголовками столбцоа и строк, так что эта таблица легко "распознается и читается взглядом". Как правило, размеры таблицы (число строк и столбцов) не задано и должно определяться программой при чтении. Считается, что элементы таблицы и заголовки отделены друг от друга пробелами и все данные таблицы имеют один и тот же тип (либо числа, либо строки). Заголовки строк и столбцов всегда рассматриваются как строковые. При этом в таблице могут быть "отсутствующие элементы", значение которых в программе должно устанавливаться заданным значением по умолчанию (например, 0).

	A	B	C	D
one	1	2	33	4
two	45		7	12
three	11	5		
four	2	2	2	2

Здесь элемент (two C) равен 7, а элемент (three D) есть 0. Программа должна составить списки имен строк и столбцов и матрицу значений таблицы, и при запросе (i, j) выдавать имя i-й строки, j-го столбца и значение элемента с индексами (i, j). Для корректного определения позиции отсутсвующих элементов можно считать, что запись значения в столбце всегла имеет пересечение по горизонтальной позиции с именем этого столбца.

9. *Разрезать "длинные" строки в файле по пробелам на более короткие (не более заданной длины).*

Требуется переформатировать текст так, чтобы его строки не превышали указанную длину (параметр этой программы). Нужно принять решение что делать, если такое разбиение формально невозможно, например, если есть слово, превышающее указанную длину. Вариантов по сути три: выдать диагностику о неправильном тексте и прекратить работу, оставить слово как есть (с превышением длины), все равно разрезать слово по требуемой длине, но уже не по пробелам. В любом случае должна быть диагностика о такой ситуации. Алгоритм достаточно простой — выделять из файла очередное слово (см. задачи выше), выводить его и подсчитывать оставшийся ресурс размера строки, если очередное слово уже не помещается в остаток текущей строки, то выводить перевод строки и начинать вывод новой строки.

10. *Удалить из файла часть текста между двумя "скобками", где скобка — это заданный набор символов. Например, убрать из файла комментарии в стиле C.*

Здесь надо определиться с тем как интерпретировать эти "скобки", а именно, учитывать ли их вложенность друг в друга, или удалять текст от открывающей скобки до первой же закрывающей, игнорируя возможные открывающие скобки внутри этого удаляемого блока. Оба решения имеют свою специфику и примерно одинаковы по трудоемкости реализации. Другой момент — это как реагировать на ситуацию, когда открытая скобка "не закрылась" до конца файла, т.е. открывающая была, а закрывающей для нее нет. Еще один вопрос — убирать сами скобки или нет. В качестве теста можно взять текст C программы и задать скобки (), [], {} и т.п.

11. *Реализовать инструкцию типа* `#include`*, т.е. вставить содержимое файла* `filename` *в то место файла, где встречается строка* `#include filename`*.*

Как известно, инструкция `#include filename` вставляет содержимое указанного файла в данном месте данного файла. Инструкции могут быть вложенными, т.е. во вставленном файле могут быть свои `#include`. Для ребализации вложенности удобно использовать рекурсивную процедуру вставки. Т.е. процедура получает на вход имя исходного файла и начинает его обработку просто копируя на выход строки файла. Если ей встречается `#include filename`, то выделяется имя нового файла, и процедура рекурсивно вызывает сама себя уже с этим новым именем. Нужно отслеживать возможное зацикливание, когда в цепочке рекурсивных вызовов встречается имя уже открытого ранее файла. Для этого процедура может формировать список имен открытых в текущий момент файлов, добавляя в него новое имя при обработке `#include filename` и удаляя это имя, когда текущий файл заканчивается. Таким образом, при добавлении нового имени файла в список проверяется не присутствет ли уже это имя в списке. Для простоты можно считать, что каждая инструкция `#include filename` располагается в отдельной строке и больше ничего в этой строке нет.

Для тестов нужно подготовить некоторое количество файлов с инструкциями `#include` для проверки правильности работы, возможности вкладывать инструкции и отслеживать циклы.

12. *Реализовать инструкции типа* `#define` *и* `#undef`*, т.е. выполнить указанные подстановки в области их задания.*

Задача напоминает замену одного образца на другой, только появляется несколько образцов и возможность отменить режим замены. Для простоты считаем, что инструкция применяется только в режиме замены одного набора символов на другой (без реализации "псевдо-функций").

Для решения задачи можно сделать список определенных на данный момент имен и соответствующих им замен. Тогда инструкции `#define` и `#undef` добавляют или удаляют элементы в этом списке. Очередная строка обрабатывается последовательным поиском имен, зарегистрированных в списке, с выполнением требуемой подстановки. Для простоты можно считать, что инструкции `#define` и `#undef` расположены по одной в отдельных строках (не смешиваются с остальным текстом). Если результат подстановки неоднозначен (например, зависит от порядка просмотра элементов списка), то выберете самостоятельно какую-то определенную логику подстановки для всей обработки текста (и опишите эту логику в отчете).

13. *Реализовать инструкции типа* `#ifdef - #else - #endif`*, т.е. оставить в файле требуемый текст в зависимости от условия.*

Так же как и в предыдущих задачах считаем, что инструкции расположены по одной в отдельных строках (назовем эти строки служебными, а остальные, в том числе и пустые, — содержательными). В данной задаче инструкции `#define` и `#undef` имеют вид простого определения имени без подстановки.

Читаем строки файла и для каждой содержательной строки принимаем решение выдавать ее на выход или нет. Решение можно принять на основе анализа трех списков:

— список определенных на данный момент имен;

— список имен, которые должны быть определены для вывода текущей строки;

— список имен, которые не должны быть определены для вывода текущей строки;

Инструкции `#define` и `#undef` модифицируют первый список. Инструкции `#ifdef` добавляет имя во второй список, `#else` переносит имя из второго списка в третий, `#endif` удаляет имя из первого и второго списков.

Таким образом, строка печатается, если все элементы второго списка присутствуют в первом, а элементы третьего списка там не присутствуют.

Строго говоря, инструкции определения имен `#define` и `#undef` могут находится внутри условных конструкций `#ifdef`. Таким образом, сам факт определения имени может зависеть от определенности других имен. Полный и аккуратный анализ логических конструкций может оказаться достаточно сложным. Поэтому можно реализовывать программу с разными допущениями о "разнообразии" синтаксических конструкций, описывая в отчете какие конкретно формы допускаются в данном варианте программы, при необходимости добавляя более сложные сочетания конструкций.

14. *Отформатировать абзацы текста в заданных границах и с красной строкой (без переноса слов). Абзац — фрагмент текста между пустыми строками.*

Задача напоминает задачу о разрезании строк на более короткие (см. выше), но здесь накладывается более жесткое требование уместить все слова данного абзаца наиболее компактно и "красиво" в заданных границах. Границ предполагется три: позиция левой границы, позиция правой границы (либо длина строки), и величина дополнительного отступа первой строки.

Выделяем слова из файла и "укладываем их" последовательно в имеющихся границах, оставляя пробел между соседними словами. Если очередное слово не помещается в доступных границах, то начинаем новую строку абзаца, старые переводы строк уже не принимаем во внимание. Надо принять волевое решение как поступать, если очередное слово принципиально не помещается в абзац (оставить, разрезать, от-

казаться и т.п.).

При сборке абзаца нужно предусмотреть два режима: с выравниваеним по левому краю (это выравнивание получается автоматически при описанной сборке строки) и с выравниванием по обоим границам. В последнем случае после первоначальной сборки строки надо равномерно вставить дополнительные пробелы между словами, чтобы последний символ последнего слова вышел точно на правую границу абзаца.