

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

**Лабораторная работа №8
курса «Алгоритмы и структуры данных»**

Выполнил студент
Пугачев Вячеслав Эдуардович
Группа: КБ-231
Проверил:
Акиньшин Даниил Иванович

Белгород, 2024

Цель работы: изучить СД типа «таблица», научиться их программно реализовывать и использовать.

Задания:

1. Для СД типа «таблица» определить:
 - (а) Абстрактный уровень представления СД:
 - i. Характер организованности и изменчивости,
 - ii. Набор допустимых операций.
 - (б) Физический уровень представления СД:
 - i. Схему хранения;
 - ii. Объем памяти, занимаемый экземпляром СД;
 - iii. Формат внутреннего представления СД и способ его интерпретации;
 - iv. Характеристику допустимых значений;
 - v. Тип доступа к элементам.
 - (с) Логический уровень представления СД. Способ описания СД и экземпляра СД на языке программирования.
2. Реализовать СД типа «таблица» в соответствии с вариантом индивидуального задания в виде модуля;
3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания с использованием модуля, полученного в результате выполнения пункта 2 задания.

Содержание

Задание 1:	3
Абстрактный уровень:	3
Задание 1.1.1:	3
Задание 1.1.2:	3
Физический уровень:	3
Задание 1.2.1:	3
Задание 1.2.2:	4
Задание 1.2.3:	4
Задание 1.2.4:	4
Задание 1.2.5:	5
Логический уровень:	5
Задание 2:	7
Задание 3:	20
Вывод:	28

Задание 1:

Абстрактный уровень:

Задание 1.1.1:

СД типа «таблица» — еще одна СД в ряду нелинейных структур. Также она не является и иерархической. Она представляет собой множество пар $\langle K, V \rangle$, где K — ключ текущей пары, а V — ее значение. Все пары в таблице имеют одинаковую организацию, то есть как ключи, так и значения имеют некоторый общий формат и тип. Все ключи в таблице обязательно уникальны: они и используются для идентификации и доступа к элементам при работе с таблицами — в то время как значения могут повторяться. Конечно, для типа значения, используемого для ключей (число или строка), чаще всего определена операция сравнения, и ключи можно упорядочить (что иногда и делается в целях ускорения поиска по ключам путем применения алгоритма бинарного поиска). Но на абстрактном уровне пары «ключ-значение» все же не всегда можно назвать упорядоченными: между элементами может не существовать смысловой последовательной связи. Скорее, таблица представляет собой пары «идентификатор объекта — описание некоторых его свойств», в то время как сами объекты могут быть между собой не связаны. Поэтому таблица и относится к особому, табличному типу СД.

«Таблица» является динамической СД: пары «ключ-значение» в ходе работы могут добавляться и удаляться, а также могут перезаписываться значения по некоторому ключу.

Задание 1.1.2:

Для СД типа «таблица» определены следующие операции, обязательные для любого варианта реализации этой структуры:

1. Инициализация,
2. Включение элемента,
3. Исключение элемента с заданным ключом,
4. Чтение элемента с заданным ключом,
5. Изменение элемента с заданным ключом,
6. Проверка пустоты таблицы,
7. Уничтожение таблицы

Физический уровень:

Задание 1.2.1:

Для хранения СД типа «таблица» может использоваться как последовательная, так и связанная схема хранения — в зависимости от того, какая вспомогательная СД была выбрана для реализации. Если элементы включаются и исключаются без сохранения упорядоченности ключей в

этой вспомогательной структуре, то используется последовательный или односвязный линейный список. Если упорядоченность сохраняется (для использования потом в поиске), то может использоваться как последовательный или односвязный список, так и бинарное дерево. Если же реализуется хэш-таблица, то есть множество возможных ключей K переводится в множество возможных адресов A с помощью некоторой хэш-функции $H(k) = a$, то для хранения элементов обычно выбирается массив.

Для реализации на ПЛС схема хранения последовательная, для массива в хэш-таблицах — последовательная с некоторыми оговорками (см. задание 1.2.3), для реализации на ОЛС или бинарном дереве — связная.

Задание 1.2.2:

Количество памяти, занимаемой СД типа «таблица», зависит от того, какое количество памяти занимает его ключ и базовый тип, выбрана ли последовательная или связная схема хранения (нужно ли отводить место для хранения указателей), сколько пар «ключ-значение» содержит таблица. Пусть базовый тип занимает T байт, ключ — S байт, таблица содержит K пар. Если таблица хранится на массиве или последовательном линейном списке (соответственно, дополнительная информация о паре не хранится), то количество памяти, занимаемой экземпляром СД «дерево», будет равно $(T + S) * K$ байт. Если таблица хранится на односвязном линейном списке, то каждую пару необходимо сопровождать указателем на следующую; один указатель типа `unsigned long` занимает 8 байт, и вся таблица будет занимать $(T + S + 8) * K$ байт. Если таблица хранится на бинарном дереве, то каждая пара будет сопровождаться двумя указателями, на левое и правое поддерево, и вся таблица будет занимать $(T + S + 16) * K$ байт. Наконец, если мы говорим о хэш-таблице, то хранить ключи пар необязательно: на ключ уже указывает адрес, по которому хранится таблица. Значит, таблица будет занимать $T * K$ байт.

Задание 1.2.3:

Формат внутреннего представления СД «таблица» также зависит от выбора вспомогательной структуры для реализации. Если таблица реализуется на ПЛС, то двоичные коды элементов (правило перевода ключей и значений в двоичный код диктуется типом ключа и значения) хранятся последовательно друг за другом. В случае с хэш-таблицами двоичные коды значений таблицы хранятся в массиве, но уже не друг за другом; они разъединены пустыми пространствами, зарезервированными для хранения значений по ключам, которые на данный момент в таблицу не включены. Если таблица реализуется на ОЛС или бинарном дереве, то каждая пара «ключ-значение» включается в структуру «запись» с одним (для ОЛС) или двумя (для бинарного дерева) указателями на следующий или на дочерние элементы. Тогда элементы хранятся в памяти компьютера независимо друг от друга, а поля структуры «запись» также могут храниться не подряд или в отличном от заданного порядке. Указатели кодируются как беззнаковые целые числа — переводятся в двоичную систему счисления и дополняются нулями слева до 64 разрядов (размера в 8 байт).

Задание 1.2.4:

Количество и диапазон допустимых значений для СД типа «таблица» определяется количеством и диапазоном допустимых значений ее базового типа (типа значений в парах «ключ-

значение»). Если таблица имеет фиксированное количество K пар «ключ-значение», то при расчете кардинального числа не учитывается диапазон значений ключей. Условно говоря, считается, что определен некоторый неизменный набор из K ключей, для которых выбираются значения, причем таблицы считаются разными, если в них один и тот же набор значений по-разному распределен между ключами. Тогда для каждого из K ключей выбирается значение из множества допустимых значений мощностью $CAR(BaseType)$, и кардинальное число для такой таблицы будет равно $CAR(BaseType)^K$. Кардинальное число СД «таблица» в целом определяется в конечном итоге как сумма по всем возможным значениям количества пар $CAR(Table) = \sum_{i=0}^{max} CAR(BaseType)^i$, где max — максимальное количество элементов (определяется объемом памяти, отведенным для хранения таблицы).

Задание 1.2.5:

Тип доступа к элементам СД типа «таблица» зависит от того, является ли таблица неупорядоченной, упорядоченной или хэш-таблицей. В неупорядоченных таблицах пары «ключ-значение» хранятся вне зависимости от значения ключа; для поиска нужного элемента применяется линейный поиск, и нужно перебрать все элементы, хранящиеся перед искомым. В упорядоченных таблицах пары хранятся по возрастанию/убыванию ключей, и для поиска элемента с нужным ключом в таких таблицах может применяться бинарный поиск или поиск по дереву — однако такой доступ также относится к последовательным. А вот в хэш-таблицах от ключа зависит, в какую область памяти будет записана пара «ключ-значение». Потому, зная ключ, можно напрямую обратиться к этой области памяти — для таких таблиц доступ будет прямым.

Логический уровень:

СД типа «таблица» является производной СД, и описать ее на логическом уровне (представить на языке программирования) можно только после самостоятельной ее реализации тем или иным способом. Приведем здесь описания для той реализации, которая будет выполнена в задании 2 этой лабораторной работы (неупорядоченная таблица на односвязном линейном списке).

```
1 #include "../VoidSingleList/VoidSingleList.h" // мотретьС лаб. №5
2
3 typedef List Table;
4 typedef char T_Key[16]; // Определить тип ключа
5
6 int main() {
7     Table T;
8 }
```

../АСД 8 си/Table/Table_example.c

Однако для такой таблицы определен только тип ключа, но не задано ничего более — ни тип базового элемента, ни его размер, ни одна из пар «ключ-значение» для этой таблицы. Все это будет возможно после написания модуля для работы с таблицами.

Индивидуальное задание; вариант 21

Модуль 2: Неупорядоченная таблица на односвязном линейном списке.

Реализация на языке C:

```
#if !defined(__TABLE1_H)
#define __TABLE1_H
#include "list3.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef List Table;
typedef ... T_Key; // Определить тип ключа
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы, адреса кото-
рых находятся в параметрах a и b. Возвращает -1, если ключ элемента по адресу a меньше
ключа элемента по адресу b, 0 — если ключи равны и +1 — если ключ элемента по адресу a
больше ключа элемента по адресу b */
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeMem, unsigned SizeEl); inline int EmptyTable(Table *T);
/* Возвращает 1, если таблица пуста, иначе — 0 */
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу. Возвращает 1
, если элемент включен в таблицу, иначе — 0 (если в таблице уже есть элемент с заданным
ключом или нехватает памяти) */
int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемента. Возвращает
1, если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента. Возвращает 1
, если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемента. Возвращает
1, если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); // Удаление таблицы из динамической памяти
#endif
```

Задача 3: Текст программы на некотором алгоритмическом языке может содержать символы-разделители, служебные слова, числовые константы и идентификаторы (слова, начинающиеся не с цифры и не являющиеся служебными). Проверить ошибки в записи идентификаторов и констант, парность служебных слов: «BEGIN» и «END», «IF» и «THEN», «FOR» и «DO», и скобок: «(» и «)», «[» и «]». Проверку констант выполнить с помощью стандартной процедуры VAL. Для проверки парности служебных слов и символов-разделителей использовать динамический массив из КР целочисленных элементов, где КР — количество парных служебных слов и символов-разделителей. Сначала все элементы массива обнуляются. Если встречается первое слово i -й пары, то i -й элемент массива увеличивается на единицу, а если второе слово — уменьшается. После обработки текста программы все элементы массива должны быть нулевыми. Парные служебные слова и символы-разделители хранить в таблице. Ключ элемента таблицы — парное служебное слово, информационная часть содержит $+i$ для первого слова i -й пары и $-i$ для второго слова. Информацию о символах-разделителях и парных служебных словах прочитать из текстовых файлов.

Задание 2:

Если перед нами стоит задача реализовать неупорядоченную таблицу на односвязном линейном списке, необходимо сначала реализовать сам односвязный линейный список. В рамках лабораторной работы №5 мы реализовывали односвязный линейный список, однако он имел базовый типа `int`, а в нашем случае базовым типом должен быть пустой указатель, что требует изменения кода некоторых функций. Таким образом, реализуем сначала модуль для работы с односвязными линейными списками.

Не будем углубляться в объяснения, поскольку, как уже было сказано, похожий код мы уже писали в рамках работы №5. Следующим образом будет выглядеть заголовочный файл:

```
1 #ifndef __LIST3_H
2 #define __LIST3_H
3
4 const short ListOk = 0;
5 const short ListNotMem = 1;
6 const short ListUnder = 2;
7 const short ListEnd = 3;
8 typedef void* BaseType;
9 typedef struct Element {
10     BaseType data;
11     struct Element* next;
12 } element;
13 typedef element *ptrel;
14 typedef struct {
15     ptrel Start;
16     ptrel ptr;
17     unsigned int N; //размер списка
18     unsigned int size; //размер информационной части элемента
19 } List;
20 short ListError;
21 //Инициализация списка из элементов, занимающих size байт, который
    создается по адресу L
22 void InitList(List *L, int size);
23 //Включение элемента со значением E в список по адресу L
24 void PutList(List *L, BaseType E);
25 //Исключение элемента из списка по адресу L и сохранение его в
    переменной по адресу E
26 void GetList(List *L, BaseType *E);
27 //Чтение элемента списка по адресу L и сохранение его в переменной по
    адресу E
28 void ReadList(List *L, BaseType *E);
29 //Возвращает 1, если список по адресу L не пуст, И 0 в противном случае
30 int FullList(List *L);
31 //Проверка: является ли элемент списка по адресу L последним
32 int EndList(List *L);
33 //Возвращает количество элементов в списке по адресу L
```

```

34 unsigned int Count(List *L);
35 //Установка в начало списка по адресу L
36 void BeginPtr(List *L);
37 //Установка в конец списка по адресу L
38 void EndPtr(List *L);
39 //Переход к следующему элементу в списке по адресу L
40 void MovePtr(List *L);
41 //Переход к nму- элементу в списке по адресу L
42 void MoveTo(List *L, unsigned int n);
43 //Удаление списка по адресу L
44 void DoneList(List *L);
45 //Копирование списка по адресу L1 в список по адресу L2
46 void CopyList(List *L1, List *L2);
47 #endif

```

../АСД 8 си/VoidSingleList/VoidSingleList.h

...А следующим — файл реализации.

```

1 #ifndef __LIST3_C
2 #define __LIST3_C
3 #include "VoidSingleList.h"
4 #include <malloc.h>
5 #include <memory.h>
6 #include <stdio.h>
7
8
9 void InitList(List *L, int size) {
10     L->Start = NULL;
11     L->ptr = L->Start;
12     L->N = 0;
13     L->size = size;
14 }
15
16 void PutList(List *L, BaseType E) {
17     ptrl new_ptr = malloc(sizeof(element));
18     (new_ptr)->data = malloc(L->size);
19     if (new_ptr == NULL || new_ptr->data == NULL) {
20         ListError = ListNotMem;
21     } else {
22         memcpy(new_ptr->data, E, L->size);
23         if (L->Start == NULL) {
24             new_ptr->next = NULL;
25             L->Start = new_ptr;
26         } else if (L->ptr == NULL) {
27             new_ptr->next = L->Start;
28             L->Start = new_ptr;
29         } else {
30             new_ptr->next = L->ptr->next;

```



```

31         L->ptr->next = new_ptr;
32     }
33     L->ptr = new_ptr;
34     L->N++;
35     ListError = ListOk;
36 }
37 }
38
39 void GetList(List *L, BaseType *E) {
40     if (L->Start == NULL) {
41         ListError = ListUnder;
42     } else if (L->ptr == NULL) {
43         ListError = ListEnd;
44     } else {
45         memcpy(E, L->ptr->data, L->size);
46         ptrrel pntr = L->ptr;
47
48         if (pntr == L->Start) {
49             L->Start = pntr->next;
50         } else {
51             BeginPtr(L);
52             while (L->ptr->next != pntr) {
53                 MovePtr(L);
54             }
55             L->ptr->next = pntr->next;
56         }
57         L->ptr = pntr->next;
58         free(pntr->data);
59         free(pntr);
60         L->N--;
61         ListError = ListOk;
62     }
63 }
64
65 void ReadList(List *L, BaseType *E) {
66     if (L->ptr == NULL) {
67         ListError = ListUnder;
68     } else {
69         memcpy(E, L->ptr->data, L->size);
70         ListError = ListOk;
71     }
72 }
73
74 int FullList(List *L) {
75     return L->N != 0;
76 }
77

```

```

78 int EndList(List *L) {
79     return (L->ptr)->next == NULL;
80 }
81
82 unsigned int Count(List *L) {
83     return L->N;
84 }
85
86 void BeginPtr(List *L) {
87     L->ptr = L->Start;
88 }
89
90 void EndPtr(List *L) {
91     BeginPtr(L);
92     for (int i = 1; i < L->N; i++) {
93         MovePtr(L);
94     }
95 }
96
97 void MovePtr(List *L) {
98     if (L->ptr == NULL) {
99         ListError = ListEnd;
100     } else {
101         L->ptr = L->ptr->next;
102         ListError = ListOk;
103     }
104 }
105
106 void MoveTo(List *L, unsigned int n) {
107     BeginPtr(L);
108     if (n > L->N) {
109         ListError = ListEnd;
110     } else {
111         for (int i = 0; i < n; i++) {
112             MovePtr(L);
113         }
114         ListError = ListOk;
115     }
116 }
117
118 void DoneList(List *L) {
119     BeginPtr(L);
120     while (FullList(L)) {
121         BaseType trash = malloc(L->size);
122         GetList(L, trash);
123     }
124     L->Start = NULL;

```

```

125     L->ptr = NULL;
126     L->size = 0;
127     ListError = ListOk;
128 }
129
130 void CopyList(List *L1, List *L2) {
131     ptr_t ptr_value = L1->ptr;
132     BeginPtr(L1);
133     BeginPtr(L2);
134     while (L2->Start != 0) {
135         BaseType trash;
136         GetList(L2, &trash);
137     }
138     for (int i = 0; i < L1->N; i++) {
139         BaseType cur_data;
140         ReadList(L1, &cur_data);
141         PutList(L2, cur_data);
142         MovePtr(L1);
143     }
144     L1->ptr = ptr_value;
145     ListError = ListOk;
146 }
147
148 #endif

```

../ACД 8 си/VoidSingleList/VoidSingleList.c

Код модуля для таблиц также будет разделен на файлы заголовка и реализации. В заголовочном файле зададим константы с кодами двух основных ошибок, которые могут возникнуть в ходе работы с таблицами: TableNotMem — не удалось выделить место под хранение нового элемента таблицы; TableUnder — попытка прочесть элемент из пустой таблицы. Под хранение кода ошибки отводится переменная.

После этого дадим название используемым типам данных. СД «таблица» объявляется как переименование СД «односвязный линейный список», у которого базовый тип — пустой указатель. Затем объявляется тип ключа, T_Key, как массив символов (строка) длиной 16; на практике же максимальная длина строки, которая может использоваться как ключ — 15, а шестнадцатая позиция отводится под ноль-символ (такой тип ключа был выбран, опираясь на условия задания 3). Немного подкорректировав данный для варианта 21 шаблон, получим такой заголовочный файл:

```

1 #ifndef __TABLE1_H
2 #define __TABLE1_H
3 #include "../VoidSingleList/VoidSingleList.h" // мотретьС лабраб.. №5
4
5 const short TableOk = 0;
6 const short TableNotMem = 1;
7 const short TableUnder = 2;
8 int TableError; // 0..2

```

```

9
10 typedef List Table;
11 typedef char T_Key[16]; // Определить тип ключа
12
13 //Сравнивает ключи элементов таблицы, адреса которых находятся в
    параметрах a и b.
14 //Возвращает -1, если ключ элемента по адресу a меньше ключа элемента
    по адресу b,
15 // 0 — если ключи равны и +1 — если ключ элемента по адресу a больше
    ключа элемента по адресу b
16 typedef int (* func)(void*, void*);
17
18 void InitTable(Table *T, unsigned SizeEl);
19 //Возвращает 1, если таблица пуста, иначе — 0
20 int EmptyTable(Table *T);
21 //Включение элемента в таблицу. Возвращает 1 , если элемент включен в
    таблицу, иначе — 0
22 //Если( в таблице уже есть элемент с заданным ключем или не хватает
    памяти)
23 int PutTable(Table *T, void *E, func f);
24 //Исключение элемента. Возвращает 1 , если элемент с ключем Key был в
    таблице, иначе — 0
25 int GetTable(Table *T, void *E, T_Key Key, func f);
26 //Чтение элемента. Возвращает 1 , если элемент с ключем Key есть в
    таблице, иначе — 0
27 int ReadTable(Table *T, void *E, T_Key Key, func f);
28 //Изменение элемента. Возвращает 1 , если элемент с ключем Key есть в
    таблице, иначе — 0
29 int WriteTable(Table *T, void *E, T_Key Key, func f);
30 //Удаление таблицы из динамической памяти
31 void DoneTable(Table *T);
32
33 #endif

```

../АСД 8 си/Table/Table.h

Реализация функций будет вынесена в отдельный файл со следующим содержанием.

Примечание: будем считать базовым типом не самой таблицы, но списка, хранящего элементы таблицы, будем считать пустой указатель на область памяти, где хранятся последовательно ключ и значение пары. Причем ключ отделен от значения одним (если его длина достигает 15) или несколькими (если эта длина меньше) нуль-символами, что позволяет нам сравнивать по ключу и два значения типа char[16], и ключ с парой «ключ-значение», и две пары. При записи, извлечении, чтении и перезаписи значения будем считать, что включается, исключается, считывается и перезаписывается вся пара, представленная в вышеописанном формате, и как аргумент должен передаваться указатель на такую пару. Такой подход потребует от нас дополнительных функций, которые будут описаны позднее.

На мысль о таком подходе наводит то, что прототип функции PutTable имеет аргумент

типа пустой указатель, но не имеет аргумента типа T_Key — значит, пустой указатель должен одновременно хранить и ключ, и значение. Логично распространить такой подход и на остальные функции.

```
1 #ifndef __TABLE1_C
2 #define __TABLE1_C
3 #include "../VoidSingleList/VoidSingleList.c" // мотретьС лабраб.. №5
4 #include "Table.h"
5
6 void InitTable(Table *T, unsigned SizeEl) {
7     InitList(T, SizeEl + sizeof(T_Key));
8 }
9
10 int EmptyTable(Table *T) {
11     return !FullList(T);
12 }
13
14 int PutTable(Table *T, void *E, func f) {
15     BeginPtr(T);
16     for (int cur_ind = 1; cur_ind <= T->N; cur_ind++) {
17         if (f(T->ptr->data, E) == 0) {
18             return 0;
19         }
20         if (cur_ind < T->N) {
21             MovePtr(T);
22         }
23     }
24     PutList(T, E);
25     //TableError = ListError;
26     //if (TableError == 1) {
27         // exit (TableError);
28     // }
29     return 1;
30 }
31
32 int GetTable(Table *T, void *E, T_Key Key, func f) {
33     BeginPtr(T);
34     for (int cur_ind = 1; cur_ind <= T->N; cur_ind++) {
35         if (f(T->ptr->data, Key) == 0) {
36             GetList(T, E);
37             return 1;
38         }
39         MovePtr(T);
40     }
41     return 0;
42 }
43
```

```

44 int ReadTable(Table *T, void *E, T_Key Key, func f) {
45     BeginPtr(T);
46     for (int cur_ind = 1; cur_ind <= T->N; cur_ind++) {
47         if (f(T->ptr->data, Key) == 0) {
48             ReadList(T, E);
49             return 1;
50         }
51         MovePtr(T);
52     }
53     return 0;
54 }
55
56 int WriteTable(Table *T, void *E, T_Key Key, func f) {
57     BeginPtr(T);
58     for (int cur_ind = 1; cur_ind <= T->N; cur_ind++) {
59         if (f(T->ptr->data, Key) == 0) {
60             memcpy(T->ptr->data, E, T->size);
61             return 1;
62         }
63         MovePtr(T);
64     }
65     return 0;
66 }
67
68 void DoneTable(Table *T) {
69     DoneList(T);
70 }
71
72 #endif

```

../АСД 8 си/Table/Table.c

Примечательно, что в итоге сценарий чтения или извлечения из пустой таблицы не приводит к ошибке — извлечение и чтение производится по ключу, и для программы равнозначны сценарии, когда ключ не найден в пустой или непустой таблице.

Для нормальных (не вызывающих ошибки и аварийного завершения) сценариев этих функций можно составить автоматизированные тесты и вынести их в отдельный файл тестирования. Порядок функций при тестировании может быть немного изменен, поскольку для заполнения таблиц значениями необходима функция PutTable, ее следует отладить раньше. Для тестирования объявляется базовый тип таблицы — целочисленный тип int.

Примечание: здесь также определяются как функция сравнения ключей, так и функции для записи по пустому указателю одновременно и ключа, и значения некоторой строки таблицы, и извлечения значения из нее. Последние две функции не могут быть объявлены ранее, потому что привязаны к базовому типу таблицы, который объявляется для каждого файла отдельно, а не в заголовочном файле модуля.

```

1 #include "Table.c"
2 #include <assert.h>

```

```

3 #include <stdio.h>
4
5 typedef int T_Value;
6
7 int compareKeys(void *E1, void *E2) {
8     char *lptr = E1;
9     char *rptr = E2;
10    while (*lptr == *rptr && *lptr != '\0') {
11        lptr++;
12        rptr++;
13    }
14    if (*lptr == *rptr) {
15        return 0;
16    } else {
17        return (*lptr > *rptr) ? 1 : -1;
18    }
19 }
20
21 void constructKeyValuePair(void *E, T_Key key, int value) {
22     char *kptr = key;
23     char *eptr = E;
24     for (int i = 0; i < sizeof(T_Key); i++) {
25         *eptr = *kptr;
26         eptr++;
27         if (*kptr != '\0') {
28             kptr++;
29         }
30     }
31     memcpy(eptr, &value, sizeof(T_Value));
32 }
33
34 T_Value getTablePairValue(void *E) {
35     return *(T_Value*)(E+sizeof(T_Key));
36 }
37
38 void Test_InitTable() {
39     Table T;
40     InitTable(&T, sizeof(T_Value));
41     assert(T.Start == NULL && T.ptr == NULL && T.N == 0 && T.size ==
sizeof(T_Key) + sizeof(T_Value));
42 }
43
44 void Test_PutTable_InEmpty() {
45     Table T;
46     InitTable(&T, sizeof(T_Value));
47     void *E = malloc(T.size);
48     constructKeyValuePair(E, "Peter", 3);

```

```

49     assert(PutTable(&T, E, compareKeys));
50     assert(compareKeys(T.ptr->data, "Peter") == 0 &&
    getTablePairValue(T.ptr->data) == 3);
51 }
52
53 void Test_PutTable_InNotEmpty() {
54     Table T;
55     InitTable(&T, sizeof(T_Value));
56     void *E = malloc(T.size);
57     constructKeyValuePair(E, "Peter", 3);
58     constructKeyValuePair(E, "Jane", 8);
59     assert(PutTable(&T, E, compareKeys));
60     assert(compareKeys(T.ptr->data, "Jane") == 0 &&
    getTablePairValue(T.ptr->data) == 8);
61 }
62
63 void Test_PutTable_failed() {
64     Table T;
65     InitTable(&T, sizeof(T_Value));
66     assert(EmptyTable(&T));
67     void *E = malloc(T.size);
68     constructKeyValuePair(E, "Peter", 3);
69     PutTable(&T, E, compareKeys);
70     constructKeyValuePair(E, "Peter", 8);
71     assert(!PutTable(&T, E, compareKeys));
72 }
73
74 void Test_PutTable() {
75     Test_PutTable_InEmpty();
76     Test_PutTable_InNotEmpty();
77     Test_PutTable_failed();
78 }
79
80 void Test_EmptyTable_IsEmpty() {
81     Table T;
82     InitTable(&T, sizeof(T_Value));
83     assert(EmptyTable(&T));
84 }
85
86 void Test_EmptyTable_IsNotEmpty() {
87     Table T;
88     InitTable(&T, sizeof(T_Value));
89     void *E = malloc(T.size);
90     constructKeyValuePair(E, "Peter", 3);
91     PutTable(&T, E, compareKeys);
92     assert(!EmptyTable(&T));
93 }

```



```

94
95 void Test_EmptyTable() {
96     Test_EmptyTable_IsEmpty();
97     Test_EmptyTable_IsNotEmpty();
98 }
99
100 void Test_GetTable_failedEmpty() {
101     Table T;
102     InitTable(&T, sizeof(T_Value));
103     void *E = malloc(T.size);
104     assert(!GetTable(&T, E, "Peter", compareKeys));
105 }
106
107 void Test_GetTable_failedNotFound() {
108     Table T;
109     InitTable(&T, sizeof(T_Value));
110     void *E = malloc(T.size);
111     constructKeyValuePair(E, "Peter", 3);
112     PutTable(&T, E, compareKeys);
113     assert(!GetTable(&T, E, "Jane", compareKeys));
114 }
115
116 void Test_GetTable_success() {
117     Table T;
118     InitTable(&T, sizeof(T_Value));
119     void *E = malloc(T.size);
120     constructKeyValuePair(E, "Peter", 3);
121     PutTable(&T, E, compareKeys);
122     constructKeyValuePair(E, "Jane", 8);
123     PutTable(&T, E, compareKeys);
124     void *E2 = malloc(T.size);
125     assert(GetTable(&T, E2, "Peter", compareKeys));
126     assert(compareKeys(E2, "Peter") == 0 && getTablePairValue(E2) ==
127         3);
128
129 void Test_GetTable() {
130     Test_GetTable_failedEmpty();
131     Test_GetTable_failedNotFound();
132     Test_GetTable_success();
133 }
134
135 void Test_ReadTable_failedEmpty() {
136     Table T;
137     InitTable(&T, sizeof(T_Value));
138     void *E = malloc(T.size);
139     assert(!ReadTable(&T, E, "Peter", compareKeys));

```

```

140 }
141
142 void Test_ReadTable_failedNotFound() {
143     Table T;
144     InitTable(&T, sizeof(T_Value));
145     void *E = malloc(T.size);
146     constructKeyValuePair(E, "Peter", 3);
147     PutTable(&T, E, compareKeys);
148     assert(!ReadTable(&T, E, "Jane", compareKeys));
149 }
150
151 void Test_ReadTable_success() {
152     Table T;
153     InitTable(&T, sizeof(T_Value));
154     void *E = malloc(T.size);
155     constructKeyValuePair(E, "Peter", 3);
156     PutTable(&T, E, compareKeys);
157     constructKeyValuePair(E, "Jane", 8);
158     PutTable(&T, E, compareKeys);
159     void *E2 = malloc(T.size);
160     assert(ReadTable(&T, E2, "Peter", compareKeys));
161     assert(compareKeys(E2, "Peter") == 0 && getTablePairValue(E2) ==
162     3);
163     assert(compareKeys(T.ptr->data, "Peter") == 0 &&
164     getTablePairValue(T.ptr->data) == 3);
165 }
166
167 void Test_ReadTable() {
168     Test_ReadTable_failedEmpty();
169     Test_ReadTable_failedNotFound();
170     Test_ReadTable_success();
171 }
172
173 void Test_WriteTable_failedEmpty() {
174     Table T;
175     InitTable(&T, sizeof(T_Value));
176     void *E = malloc(T.size);
177     constructKeyValuePair(E, "Peter", 3);
178     assert(!WriteTable(&T, E, "Peter", compareKeys));
179 }
180
181 void Test_WriteTable_failedNotFound() {
182     Table T;
183     InitTable(&T, sizeof(T_Value));
184     void *E = malloc(T.size);
185     constructKeyValuePair(E, "Peter", 3);
186     PutTable(&T, E, compareKeys);

```

```

185     void *E2 = malloc(T.size);
186     constructKeyValuePair(E2, "Jane", 8);
187     assert(!WriteTable(&T, E2, "Jane", compareKeys));
188 }
189
190 void Test_WriteTable_success() {
191     Table T;
192     InitTable(&T, sizeof(T_Value));
193     void *E = malloc(T.size);
194     constructKeyValuePair(E, "Peter", 3);
195     PutTable(&T, E, compareKeys);
196     void *E2 = malloc(T.size);
197     constructKeyValuePair(E2, "Peter", 8);
198     assert(WriteTable(&T, E2, "Peter", compareKeys));
199     assert(compareKeys(T.ptr->data, "Peter") == 0 &&
200     getTablePairValue(T.ptr->data) == 8);
201 }
202
203 void Test_WriteTable() {
204     Test_WriteTable_failedEmpty();
205     Test_WriteTable_failedNotFound();
206     Test_WriteTable_success();
207 }
208
209 void Test_DoneTable() {
210     Table T;
211     InitTable(&T, sizeof(T_Value));
212     void *E = malloc(T.size);
213     constructKeyValuePair(E, "Peter", 3);
214     PutTable(&T, E, compareKeys);
215     constructKeyValuePair(E, "Jane", 8);
216     PutTable(&T, E, compareKeys);
217     DoneTable(&T);
218     assert(T.size == 0 && T.N == 0 && T.Start == NULL && T.ptr ==
219     NULL);
220 }
221
222 void Test_Table() {
223     Test_InitTable();
224     Test_EmptyTable();
225     Test_PutTable();
226     Test_GetTable();
227     Test_ReadTable();
228     Test_WriteTable();
229     Test_DoneTable();
230 }

```

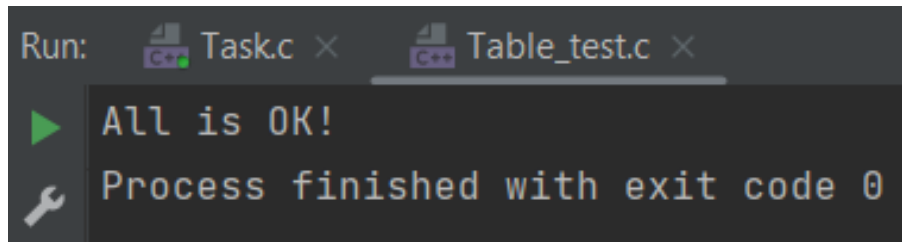
```

230 int main() {
231     Test_Table();
232     printf("All is OK!");
233 }

```

../АСД 8 си/Table/Table_test.c

Запустив программу, можем самостоятельно убедиться, что все тесты прошли успешно:



```

Run: Task.c x Table_test.c x
All is OK!
Process finished with exit code 0

```

Задание 3:

Теперь, когда модуль для СД типа «таблица» реализован, можем перейти к решению задачи для варианта 21, описанной выше. Как указано в условии, ключами таблицы, хранящей информацию о парных служебных словах и разделителях, должны быть сами эти слова, а значениями — числа, отражающие номер пары; поэтому в качестве типа ключа мы оставляем строку не больше 15 символов, а в качестве значения — целое число.

По сути, наша программа должна будет решать следующие подзадачи: 1) формировать таблицу для хранения информации о том, как служебные слова и разделители связаны в пары; 2) Создать пустой массив для хранения информации о том, сохраняется ли баланс открывающих и закрывающих слов для каждой пары, 3) Принимать имя файла, где записан текст кода, с клавиатуры, и записывать информацию из него в строку-буфер, 4) проходить по словам строки-буфера и для каждого делать выводы, является ли оно служебным словом или разделителем (и изменять массив «баланса» в таком случае), либо идентификатором, оператором или числовой константой, 5) по окончании обработки выводить информацию о найденных несоответствиях или ошибках или их отсутствии.

Определим ряд вспомогательных функций. Для создания таблицы (и последующей с ней работы) нам понадобятся те же функции создания пары «ключ-значение» и извлечения значения из пары.

```

1 #include "Table.c"
2 #include <assert.h>
3 #include <stdio.h>
4
5 typedef int T_Value;
6
7 int compareKeys(void *E1, void *E2) {
8     char *lptr = E1;
9     char *rptr = E2;
10    while (*lptr == *rptr && *lptr != '\0') {
11        lptr++;

```

```

12     rptr++;
13 }
14 if (*lptr == *rptr) {
15     return 0;
16 } else {
17     return (*lptr > *rptr) ? 1 : -1;
18 }
19 }
20
21 void constructKeyValuePair(void *E, T_Key key, int value) {
22     char *kptr = key;
23     char *eptr = E;
24     for (int i = 0; i < sizeof(T_Key); i++) {
25         *eptr = *kptr;
26         eptr++;
27         if (*kptr != '\\0') {
28             kptr++;
29         }
30     }
31     memcpy(eptr, &value, sizeof(T_Value));
32 }
33
34 T_Value getTablePairValue(void *E) {
35     return *(T_Value*)(E+sizeof(T_Key));
36 }

```

../АСД 8 си/Table/Table_test.c

Для копирования информации из файла в буфер хватает стандартных функций библиотеки stdio. Для разделения строки на слова воспользуемся набором функций из лабораторной работы №17 в рамках курса ОП (2-ой семестр). Точнее говоря, вводятся функции для поиска первого пробельного и непробельного символа, начиная с некоторого указателя. Объявляется также структура WordDescriptor, хранящая указатель на первый символ слова и символ, следующий за последним. Таким образом, можно ввести функцию, сохраняющую в структуру WordDescriptor указатели на начало и конец первого слова, встреченного после некоторого указателя. Также можно ввести функцию, перезаписывающую последовательность символов между указателями на начало и конец, в другую строку, со вставкой ноль-символа в конце.

```

1 typedef struct {
2     char *begin;
3     char *end;
4 } WordDescriptor;
5
6 char* findNonSpace(char *begin) {
7     while (*begin != '\\0' && isspace(*begin)) {
8         begin++;
9     }
10    return begin;
11 }

```

```

12
13 char* findSpace(char *begin) {
14     while (*begin != '\0' && !isspace(*begin)) {
15         begin++;
16     }
17     return begin;
18 }
19
20 bool getWord(char *beginSearch, WordDescriptor *word) {
21     word->begin = findNonSpace(beginSearch);
22     if (*word->begin == '\0')
23         return 0;
24     word->end = findSpace(word->begin);
25     return 1;
26 }
27
28 void writeWordAsString(WordDescriptor word, char *dest) {
29     char *pointer = word.begin;
30     while (pointer != word.end) {
31         *(dest + (pointer - word.begin)) = *pointer;
32         pointer++;
33     }
34     *(dest + (pointer - word.begin)) = '\0';
35 }

```

../АСД 8 си/Table/Task.c

Когда задача разбиения строки на слова решена, можно ввести функцию, определяющую, является ли переданное ей слово ошибочно введенной переменной или числовой константой. Здесь требуются некоторые пояснения. Мы считаем словом последовательность символов, отделенную по краям пробельными символами, однако даже в языке программирования с самым лаконичным синтаксисом должны использоваться также операторы (обозначения арифметических операций, запятые), которые не всегда отделяются от идентификаторов и констант пробелами. Мы будем идти по переданному функции слову, задав изначально ложные флаги «начато число» и «начат идентификатор». При считывании символа, если он является цифрой и флаг «начат идентификатор» является ложным, флаг «начато число» становится истинным. Если же был встречен символ алфавита или символ «_», при этом истинен флаг «начато число», функция возвращает ложный результат; если же число не было начато, считается, что начат идентификатор. Если встреченный символ не является ни буквой, ни цифрой, проверяется, не относится ли он к списку «запрещенных», то есть парных разделителей, зарезервированных для других целей. Если это не так, устанавливаются в значение false флаги «начато число» и «начат идентификатор»; считается, что текущая последовательность символов рассмотрена, и далее идет некий оператор, отделяющий ее от следующей. ну а если встреченный символ все-таки относится к списку разделителей, функция возвращает false.

Получается, такая реализация требует, чтобы для корректной работы программы все парные разделители, как и парные служебные слова, были отделены пробелами и обособлены от остального кода.

```

1 bool isForbiddenSymbol(char c) {
2     return c == '(' || c == ')' || c == '[' || c == ']';
3 }
4
5 bool isWordValuable(WordDescriptor word) {
6     char *pointer = word.begin;
7     bool result = true;
8     bool was_num_started = false;
9     bool was_ident_started = false;
10
11     while (pointer != word.end && result) {
12         if (isalpha(*pointer) || *pointer == "_") {
13             if (was_num_started) {
14                 result = false;
15             } else {
16                 was_ident_started = true;
17             }
18         } else if (isdigit(*pointer)) {
19             if (!was_ident_started) {
20                 was_num_started = true;
21             }
22         } else if (!isForbiddenSymbol(*pointer)) {
23             was_num_started = false;
24             was_ident_started = false;
25         } else {
26             result = false;
27         }
28         pointer++;
29     }
30     return result;
31 }

```

../АСД 8 си/Table/Task.c

Наконец, можно задать массив типа `char` как буфер перейти к телу функции `main`. Инициализируется таблица для хранения парных слов и разделителей, данные парные слова и разделители задаются в массиве литералов, и проходом по этому массиву формируются строки таблицы.

Также инициализируется нулями массив для подсчета того, сколько раз встречались открывающие и закрывающие слова этих пар, и создается еще одна пустая таблица. Ее назначение поясним ниже.

Затем запрашивается название файла с клавиатуры, файл связывается с файловой переменной, и его содержимое посимвольно копируется в буфер, пока не будет достигнут конец файла. После этого в буфере после содержимого файла добавляется нуль-символ, а файл закрывается.

Далее вводится указатель для пробега по содержимому буфера, и запускается цикл, выполняющийся до тех пор, пока по этому указателю можно считать слово. В конце каждой

итерации указатель сдвигается на позицию конца текущего слова. В теле же цикла содержимое слова копируется в отдельную строку, и первым делом проверяется: можно ли считать в заранее созданную переменную строку таблицы, если использовать текущее слово как ключ? Если можно, значит, найденное слово принадлежит к списку ключевых слов или разделителей. Из переменной, куда производилось копирование строки таблицы, считывается значение, по нему находится нужный индекс массива подсчета открывающих и закрывающих слов, и его значение увеличивается или уменьшается, в зависимости от того, положительно или отрицательно значение.

Если считать строку таблицы по данному ключу нельзя, проверяется: является ли данное слово допустимой константой, переменной или их последовательностью, разделенной операторами? Если да, то никаких действий предпринимать не надо. Но если нет? Возможно, было бы логично просто вывести сообщение об ошибке и прекратить работу. С другой стороны, было бы удобно сразу получить как можно более полную информацию об имеющихся ошибках, а не перезапускать программу после каждого исправления имен переменных, только чтобы получить сообщение о новой ошибке. Но если выполнение программы будет продолжаться после найденной ошибки, а переменная с недопустимым именем используется снова и снова, то вывод будет состоять из множества однотипных сообщения. Значит, надо хранить информацию о том, какие недопустимые имена встречались и — что тоже было бы полезно — сколько раз.

Для этого нам и пригодится вторая таблица — для хранения имен недопустимых переменных. Если найденная последовательность признана ошибочной, конструируется переменная со строкой таблицы, где ключ — данная последовательность, значение — единица, и производится попытка прочитать строку по этому ключу в эту же переменную. Если она была успешной, значит, такая переменная уже встречалась: из строки считывается значение, конструируется новая строка, где это значение увеличивается на 1, и перезаписывается в таблицу. если же попытка чтения была неудачной, значит, такое ошибочное имя встречается впервые, и нужно занести строку в таблицу.

В конце сначала производится проход по элементам массива-счетчика: если один из элементов не равен 0, то выводится сообщение, что соответствующая конструкция не завершена верно. Потом по очереди извлекаются строки из таблицы ошибочных имен, и выводится информация о том, какое из них сколько раз было встречено в программе. Если же ни одного вывода не было совершено, выводится сообщение о корректности программы.

```
1 char buffer[1000];
2 int main() {
3     Table table_of_paired;
4     InitTable(&table_of_paired, sizeof(T_Value));
5     char array_of_paired[10][sizeof(T_Key)] = {"BEGIN", "END", "IF",
6 "THEN", "FOR", "DO", "(", ")", "[", ""]};
7     for (int i = 0; i < 5; i++) {
8         void *cur_el = malloc(table_of_paired.size);
9         constructKeyValuePair(cur_el, array_of_paired[2*i], i+1);
10        PutTable(&table_of_paired, cur_el, compareKeys);
11        constructKeyValuePair(cur_el, array_of_paired[2*i+1], -i-1);
12        PutTable(&table_of_paired, cur_el, compareKeys);
13    }
14    int counter_of_paired[5] = {0, 0, 0, 0, 0};
15    Table table_of_unknown;
```



```

15 InitTable(&table_of_unknown, sizeof(T_Value));
16
17 char filename[30];
18 scanf("%s", filename);
19 FILE *file = fopen(filename, "r");
20 int cur = 0;
21 while ((buffer[cur] = fgetc(file)) != EOF) {
22     cur++;
23 }
24 buffer[cur] = '\0';
25 fclose(file);
26
27 char *cursor = buffer;
28 WordDescriptor cur_word;
29 bool is_word_found = getWord(cursor, &cur_word);
30 while (is_word_found) {
31     char *string_word = malloc(cur_word.end - cur_word.begin + 1);
32     writeWordAsString(cur_word, string_word);
33     void *container = malloc (table_of_paired.size);
34     if (ReadTable(&table_of_paired, container, string_word,
compareKeys)) {
35         int value_of_pair = getTablePairValue(container);
36         if (value_of_pair > 0) {
37             counter_of_paired[value_of_pair-1]++;
38         } else {
39             counter_of_paired[-value_of_pair-1]--;
40         }
41     } else if (!isWordValuable(cur_word)) {
42         if (ReadTable(&table_of_unknown, container, string_word,
compareKeys)) {
43             int frequency = getTablePairValue(container);
44             constructKeyValuePair(container, string_word,
frequency+1);
45             WriteTable(&table_of_unknown, container, string_word,
compareKeys);
46         } else {
47             constructKeyValuePair(container, string_word, 1);
48             PutTable(&table_of_unknown, container, compareKeys);
49         }
50     }
51     cursor = cur_word.end;
52     is_word_found = getWord(cursor, &cur_word);
53 }
54
55 bool has_errors = false;
56 for (int i = 0; i < 5; i++) {
57     if (counter_of_paired[i] != 0) {

```

```

58         printf("ERROR! Some constructions <%s - %s> are not closed
properly\n", array_of_paired[2*i], array_of_paired[2*i+1]);
59         has_errors = true;
60     }
61 }
62 while (!EmptyTable(&table_of_unknown)) {
63     void *container_of_unknown = malloc(table_of_unknown.size);
64     void *first_string = table_of_unknown.Start->data;
65     GetTable(&table_of_unknown, container_of_unknown,
first_string, compareKeys);
66     int frequency = getTablePairValue(container_of_unknown);
67     printf("ERROR! Sequence <%s> is not service word, correct
variable name, operator, or number (repeats %d times)\n",
first_string, frequency);
68     has_errors = true;
69 }
70 if (!has_errors) {
71     printf("Program is correct\n");
72 }
73 }

```

../АСД 8 си/Table/Task.c

Для того, чтобы протестировать полученную программу, будем использовать следующие демонстрационные тексты программ:

```

BEGIN
IF var1 > var2 THEN
    BEGIN
        real_sum := var1+var2
    END
FOR i := 1 TO 10 DO
    BEGIN
        square_sum += i*i
    END
END

```

В данном тексте программы ошибок нет, при этом используется несколько разных пар служебных слов, есть переменные, содержащие в своих именах цифры и символ «_», и переменные, соединенные операторами без пробелов.

```

BEGIN
array := [ 3, 1, 7, 5
len := 5
res := 0
FOR i := 1 TO len DO
    res += ( A [ i ] + 2 ) * 3
END
END

```

В данном тексте программы имена переменных не содержат ошибок, но есть ошибки в использовании парных служебных слов и разделителей: отсутствует закрывающая скобка «]» при наличии открывающей «[», а также отсутствует служебное слово BEGIN, обозначающее начало блока для цикла FOR, тогда как END присутствует.

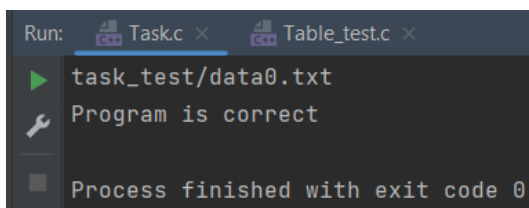
```
BEGIN
1comp, co2mp, comp3
z := ( 1comp + co2mp + comp3 ) * 30
p := 1comp * co2mp * comp3 + 30
END
```

В данном тексте конструкции из парных слов не содержат ошибок, но есть ошибки в именовании переменных: имя 1comp недопустимо по правилам именования.

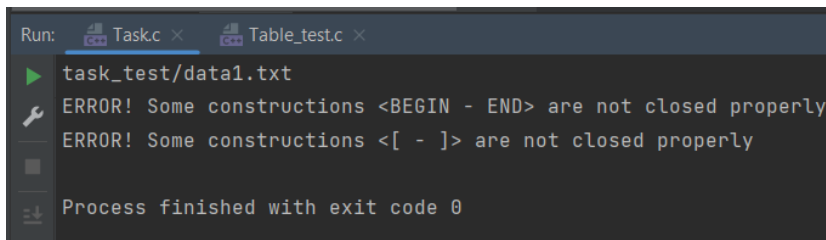
```
BEGIN
P := 1
FOR i( <= 5
  BEGIN
    P := P*i
    i := i+1
  END
END
```

В данном тексте есть как неправильно закрытые конструкции из парных слов (слово FOR не имеет парного ему слова DO), так и ошибки в именовании переменных: имя i(не будет правильно считываться, потому что название переменной не отделено от разделителя.

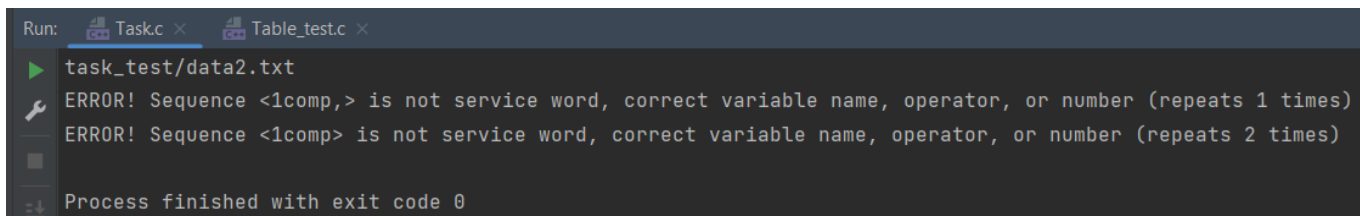
Запустим полученную программу для каждого из этих файлов:



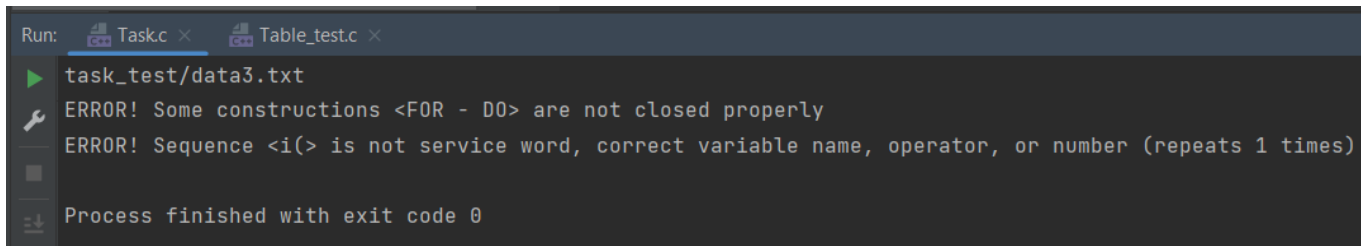
```
Run: Task.c x Table_test.c x
task_test/data0.txt
Program is correct
Process finished with exit code 0
```



```
Run: Task.c x Table_test.c x
task_test/data1.txt
ERROR! Some constructions <BEGIN - END> are not closed properly
ERROR! Some constructions <[ - ]> are not closed properly
Process finished with exit code 0
```



```
Run: Task.c x Table_test.c x
task_test/data2.txt
ERROR! Sequence <1comp,> is not service word, correct variable name, operator, or number (repeats 1 times)
ERROR! Sequence <1comp> is not service word, correct variable name, operator, or number (repeats 2 times)
Process finished with exit code 0
```

A screenshot of a C++ IDE's console window. The window has a dark background with light-colored text. At the top, there are two tabs: 'Taskc' and 'Table_test.c'. The main area of the console displays the following text: 'task_test/data3.txt' followed by two error messages. The first error message is 'ERROR! Some constructions <FOR - DO> are not closed properly'. The second error message is 'ERROR! Sequence <i(> is not service word, correct variable name, operator, or number (repeats 1 times)'. At the bottom of the console, it says 'Process finished with exit code 0'. On the left side of the console, there are several small icons: a green play button, a wrench, a square, and a download arrow.

Легко заметить, что в конечном итоге вывод программы совпадает с нашими собственными размышлениями.

Примечание: на примере обработки файла data2.txt, однако, видно, что хотя программа способна подсчитывать встреченные ошибочные последовательности, она не может отличить, является ли такая последовательность отдельным идентификатором или последовательностью из нескольких идентификаторов или операторов. Впрочем, по условию задачи этого и не требовалось.

Вывод:

В ходе лабораторной работы дали характеристику СД типа «таблица», форматам ее представления, реализовали один из них в соответствии с вариантом (Неупорядоченная таблица на односвязном линейном списке с базовым типом «пустой указатель»), написали ряд базовых функций для работы с таблицами в этом формате, а также решили задачу проверки на синтаксические ошибки программы на алгоритмическом языке.