**Министерство образования и науки РФ**

**Севастопольский государственный университет**

# Методические указания

**к выполнению лабораторных работ**
**по дисциплине «Операционные системы»**
**для студентов очной и заочной форм обучения**
**направлений подготовки**
**09.03.01 «Информатика и вычислительная техника»**
**09.03.02 «Информационные системы и технологии»**
**27.03.04 «Управление в технических системах»**

**Севастополь**

**2018**

УДК 681.3

Методические указания к выполнению лабораторных работ по дисциплине «Операционные системы» / Е.М. Шалимова – Севастополь: Изд–во СевГУ, 2018. –26 с.

Цель методических указаний – оказание помощи студентам в выполнении лабораторных работ по дисциплине. Методические указания содержат краткое изложение основных теоретических положений, задания на лабораторные работы, порядок их выполнения и требования к отчетам, а также список рекомендованной литературы.

# СОДЕРЖАНИЕ

# 1. ОБЩИЕ ПОЛОЖЕНИЯ, ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНЫХ РАБОТ И ТРЕБОВАНИЯ К ОТЧЕТАМ

В соответствии с программой дисциплины «Операционные системы» студент должен выполнить ряд лабораторных работ.

По результатам выполнения каждой лабораторной работы оформляется и защищается отчет, который должен содержать следующие основные элементы:

- титульный лист;
- цель работы;
- постановку задачи с указанием варианта задания;
- описание используемых структур данных;
- описание алгоритма решения задачи;
- спецификации подпрограмм;
- тестовые примеры;
- текст программы;
- результаты и выводы.

Отчёты по лабораторной работе оформляются и защищаются каждым студентом индивидуально. При защите лабораторной работы студент должен продемонстрировать работу программы.

Разработанные программы должны обеспечивать ввод исходных данных, проверку их корректности, вывод исходных данных и результатов. Логически законченные фрагменты должны быть оформлены в виде подпрограмм, все необходимые данные которым передаются через список параметров.

# 2. ЛАБОРАТОРНАЯ РАБОТА №1. ОРГАНИЗАЦИЯ ВВОДА-ВЫВОДА В ЯЗЫКЕ С.

**Цель работы:** изучение принципов организации и программирование ввода-вывода в языке С.

## 2.1. Основные теоретические положения

Функции ввода-вывода объявлены в заголовочном файле <stdio.h>.

При запуске С-программы операционная система всегда открывает три стандартных файла (или потока): входной (файловый указатель stdin), выходной (stdout) и файл ошибок (stderr). Обычно stdin соотнесен с клавиатурой, stdout и stderr –с экраном.

Любой другой файл предварительно должен быть открыт. Для этого используется функция **fopen**:

**FILE* fopen(char* name, char* mode);**

Функция получает в качестве аргументов внешнее имя файла (**name**) и режим доступа (**mode**), а возвращает файловый указатель, используемый в дальнейшем для работы с файлом. Функция возвращает нулевой указатель, если файл не может быть открыт по каким либо причинам.

Файловый указатель ссылается на специальную структуру типа **FILE**, содержащую информацию о файле.

Режим доступа может принимать значения, указанные в таблице 2.1.

Таблица 2.1- Режимы доступа к файлу

| | |
|---|---|
| “r” | - файл открывается только для чтения; |
| “w” | - файл создается только для записи, при этом, если он уже существовал, его содержимое теряется; |
| “a” | - файл создается или открывается для записи в конец файла; |
| “r+” | - файл открывается для чтения и для записи; |
| “w+” | - файл создается для чтения и для записи, при этом, если он уже существовал, его содержимое теряется; |
| “a+” | - файл создается или открывается для чтения и для записи в конец файла |

После окончания работы файл должен быть закрыт с помощью функции **fclose:**.

**int fclose(FILE*);**

Если программа завершается нормально, все открытые файлы автоматически закрываются системой.

При работе с файлами существуют различия, определяемые набором функций, используемых для ввода/вывода данных. В частности, рассмотрим форматированный ввод, для этого используется функция

**int fscanf(FILE *f,const char *format [, p1,p2,. . .]);**

Функция читает текстовые данные из файла f , преобразует их в соответствии со спецификациями, содержащимися в формате и присваивает по порядку аргументам p1,p2,…, каждый из которых должен быть указателем.

Для чтения из стандартного файла используется функция

**int scanf(const char *format [, p1,p2,. . .]);**

Обратные действия производит функция форматированного вывода

**int fprintf(FILE *f,const char *format, . . .) ;**

Функция преобразует аргументы (список которых обозначен многоточием) в текстовое представление в соответствии с форматом (format) и пишет в выходной файл f. Для вывода в стандартный файл используется функция

**int printf(const char *format, . . .) ;**

Спецификации форматов для функций семейств **scan** и **print** даны в приложении А.

Пример1. Рассмотрим пример обработки строки символов. Для заданной строки вывести все символы, расположенные между первой и второй запятыми. Считать, что слова разделены пробелами; строка заканчивается точкой и не может содержать больше 20 символов.

При решении данной задачи необходимо:

1. Определить адрес первой ’,’.
2. Определить адрес второй ’,’.
3. Если в строке нет двух ’,’ , напечатать соответствующее сообщение, иначе определить длину искомой подстроки и напечатать ее.

Текст программы:

```
#include <stdio.h>
#include <string.h>

 main()
{ int i=0,j;
 char x[20];  char*u1,*u2;

   scanf("%c",&x[0]);
   while (x[i]!='.')
```

```
   { i++;
 scanf("%c",&x[i]); }
u1= strchr(x,',') ; *u1=' ' ; u2= strchr(x,',') ;
if((u1==NULL) ||(u2==NULL) )
     printf("The string hasn't got two ,") ;
else {
        i=u2-u1;
     for(j=0;j<i;j++)
    printf("%c",u1[j]);
         }
}
```

Рассмотрим работу с файлами, содержащими текстовую информацию, разбитую на строки. Для записи в файл текстовой информации используются функции fputc и fputs, а для чтения –fgetc и fgets. Строка в текстовом файле заканчивается специальным символом –'\n'.

Пример2. Переписать начальные строки (не более 10-ти) из текстового файла "aaa" в конец файла "bbb". Длина строки не превышает 80 символов (включая '\n').

```
main()
{ FILE *f1,*f2;
  char s[81];
  int i=0;
  if((f1=fopen("aaa","r"))==NULL) exit(1);
  if((f2=fopen("bbb","a"))==NULL) exit(1);
  while(fgets(s,81,f1)!=NULL && i<10) {
    i++;
    fputs(s,f2);
  }
  fclose(f1); fclose(f2);
}
```

Следует отметить, что

**char *fgets(char *s,int n,FILE *f);**

читает не более n-1 литер в массив s, прекращая чтение, если встретился '\n', который включается в массив. В любом случае массив дополняется нулевым байтом. Если в решении задачи заменить второй параметр при вызове функции fgets на 80, то строки будут переписываться без искажения, но их количество может оказаться меньше десяти, если в исходном файле окажутся строки максимальной длины.

Для текстовых файлов использование функции

**int feof(FILE *f)**

имеет некоторые особенности. Если при чтении из файлового потока f достигнут конец файла, то возвращается ненулевое значение, в противном случае возвращается ноль. При этом, если не предпринималась попытка прочитать информацию за концом файла, то функция feof не будет сигнализировать о том, что достигнут конец файла.

Функции для работы со строками находятся в файле <string.h>. В приложении Б приведены объявления некоторых функций и их назначение.

## 2.2. Задание на лабораторную работу

Разработать программу обработки строки символов в соответствии с заданным вариантом. Считать, что строка оканчивается точкой, слова разделены пробелами. В программе предусмотреть ввод и вывод исходных данных и результатов Варианты заданий приведены в таблице 2.1.

Таблица 2.1- Варианты заданий

| Номер варианта | Задание |
|---|---|
| **1** | Во всех словах удалить первые буквы. |
| **2** | Во всех словах оставить только первые буквы. |
| **3** | Подсчитать количество слов "char" . Заменить их на "int" . |
| **4** | Подсчитать количество слов, в которых первый и последний символы одинаковые. Однобуквенные слова не учитывать. |
| **5** | Поменять местами первое и последнее слова. |
| **6** | Удалить все цифры. |
| **7** | Найти самое длинное слово. |
| **8** | Подсчитать число слов, число символов в словах и общее число символов. |
| **9** | Подсчитать количество слов, в которых четное количество символов. |
| **10** | Подсчитать количество слов, в которых первый и последний символы одинаковы. Однобуквенные слова учитывать. |

# 3. ЛАБОРАТОРНАЯ РАБОТА №2. ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ ПАМЯТИ В ЯЗЫКЕ С.

**Цель работы:** изучение принципов организации и программирование основных операций, выполняемых с линейными списковыми структурами.

## 3.1. Линейные структуры данных. Основные теоретические положения

Линейные структуры, в соответствии с [1,2], - это множество, состоящее из $n \geq 0$ записей X1, X2, ... , Xn, структурные свойства которого определяются одномерным относительным положением записей: X1 является первой записью; если $1 < k < n$, то k-ой записи Xk предшествует (k-1)-ая Xk-1, а за ней следует (k+1)-ая Xk+1; Xn является последней записью.

Одной из самых распространенных линейных статических структур является одномерный массив. Для хранения массива в памяти, как правило, используется ее последовательное распределение, подразумевающее размещение элементов массива в последовательных ячейках памяти. Местоположение массива в памяти определяется адресом его первого элемента.

Для хранения динамической линейной структуры используется более гибкая схема, называемая связанным распределением. Каждая запись динамической линейной структуры, называемой списком, содержит указатель на следующую запись. Такой список называется однонаправленным (односвязным). Указатель последней записи равен коду признака конца списка, а указатель на первую запись задается дополнительным параметром, называемым переменной связи. Если в каждый элемент списка добавить еще один указатель – на предыдущий элемент, то получится двунаправленный список (двусвязный). Тип списка принципиально не влияет на реализацию операций, поэтому далее ограничимся рассмотрением односвязных списков, а в Приложении А приведем пример программы обработки двунаправленного списка.

Пусть каждый элемент списка содержит два поля: INFO, содержащее значение элемента данных Xk, и LINK, содержащее адрес следующего элемента Xk+1. Переменную связи назовем FIRST. Введенные обозначения будем использовать в программных иллюстрациях операций над списками.

Связанное распределение требует дополнительное пространство в памяти для связей. Но при его использовании существенно упрощаются такие операции над линейной структурой, как удаление ее элемента и включение в нее нового элемента, которые сводятся, как показано на рисунке1а и 1б, к изменению соответствующих полей LINK элементов структуры.

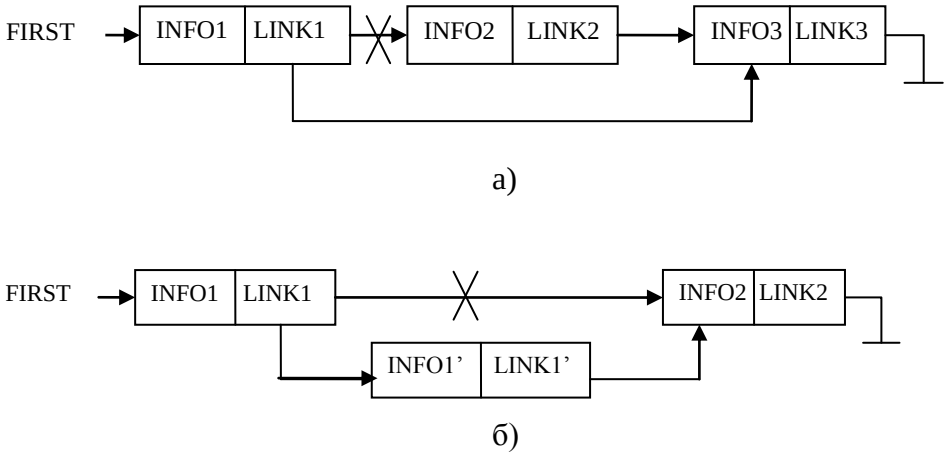


Рисунок 1 - Преобразование линейного списка
а) удаление элемента 2;
б) включение элемента 1'.

Особенно полезны списковые структуры в тех случаях, когда в памяти ЭВМ требуется разместить несколько динамически изменяющихся структур данных (массивов, строк, таблиц и т.д.) с заранее неизвестным числом элементов. Кроме того, использование списков целесообразно, когда требуется упорядочить элементы данных без их физического перемещения или связать воедино элементы данных, размещенные в разных местах памяти [1-4].

Для описания динамических структур данных удобным средством многих языков программирования являются переменные - указатели.

Приведем примеры реализации прохода списковой структуры, включения в нее нового элемента и удаления из нее элемента средствами языка Паскаль. При этом будем полагать, что список задается переменной связи FIRST и описание списковой структуры имеет вид:

```
type
  NLINK=^LIST;
  LIST=record
       INFO:integer;
       LINK:NLINK
      end;
var FIRST:NLINK;
```

Процедура прохода списка LISTPRINT выполняет вывод всех его элементов, начиная с элемента, указанного FIRST, и заканчивая элементом, содержащим nil в поле LINK.

```
procedure LISTPRINT(FIRST:NLINK);
 var
   BASE:NLINK;
 begin
   BASE:=FIRST;
   while BASE<>nil do
    begin
     writeln(BASE^.INFO);
     BASE:=BASE^.LINK
    end;        end;
```

Процедура включения элемента LISTADD добавляет в список новый элемент после элемента с заданным номером K, если в списке не менее K элементов. В противном случае элемент добавляется к списку в качестве последнего. Поле данных нового элемента при этом заполняется значением, заданным переменной NEWINFO.

```
procedure LISTADD(K, NEWINFO:integer; var FIRST:NLINK);
    var
      I:integer;
      NEWNODE,BASE:NLINK;
   begin
          new(NEWNODE);
          NEWNODE^.INFO:=NEWINFO;
     if FIRST<>nil then
       begin
             BASE:=FIRST;
             for I:=1 to K-1 do
             if BASE^.LINK<>nil then BASE:=BASE^.LINK;
             NEWNODE^.LINK:=BASE^.LINK;
             BASE^.LINK:=NEWNODE;
       end
    else begin FIRST:=NEWNODE; NEWNODE^.LINK:=nil;
```

```
    end;
end;
```

Процедура удаления элемента LISTDEL удаляет из списка элемент с заданным номером K, если в списке не менее K элементов. В противном случае процедура не изменяет структуру списка.

```
procedure LISTDEL(K:integer; var FIRST:NLINK);
 var
  I:integer;
  DELNODE,BASE:NLINK;
 begin
  if FIRST<>nil then
   begin
    BASE:=FIRST;
    for I:=1 to K-2 do
    if BASE<>nil then BASE:=BASE^.LINK;
    DELNODE:=BASE^.LINK;
    BASE^.LINK:=DELNODE^.LINK;
   end;
 end;
```

В программировании часто встречаются линейные структуры, в которых включение и исключение или доступ к значениям производятся только в первой или последней записи. Эти структуры имеют специальные названия: стек, очередь, дек [1,2].

При работе с этими структурами будем полагать, что процедура извлечения элемента предполагает его удаление.

СТЕК - линейная структура, в которой все включения и исключения и всякий доступ делаются в одном конце структуры.

Из стека всегда извлекается "младший" элемент из имеющихся в списке, т.е. тот, который был включен позже других. Этот элемент называют вершиной стека.

ОЧЕРЕДЬ - линейная структура, в которой все включения производятся на одном конце структуры, а все исключения и доступ на другом ее конце. Очередь задается двумя указателями: на первую запись (указатель начала очереди) и на последнюю запись (указатель конца очереди).

ДЕК - это линейная структура, в которой включение и исключение элементов, а также и доступ к элементам возможны на обоих концах структуры. Очевидно, что для реализации этих действий необходимо задавать, с какого конца требуется выполнить операцию.

## Динамические структуры данных

Структуры данных делятся на статические и динамические [1-3]. Отличительным признаком статических структур является фиксированный размер выделяемой им на этапе компиляции памяти. Для данных динамического типа память выделяется по мере необходимости на этапе выполнения программы. Рассмотрим наиболее часто используемые динамические структуры: списки и бинарные деревья.

Каждый элемент списка включает, по меньшей мере, два поля: информационное, содержащее значение элемента данных, и адресное, содержащее адрес следующего элемента (информационных полей может быть несколько). Такой список называется однонаправленным (односвязным). Указатель последней записи равен коду признака конца списка, а указатель на первую запись задается дополнительным параметром, называемым переменной связи. Если в каждый элемент списка добавить еще один указатель – на предыдущий элемент, то получится двунаправленный список.

Каждый узел бинарного дерева может быть представлен совокупностью трех полей: информационным, содержащим значение элемента данных, и двумя адресными, содержащими соответственно адреса левого и правого поддеревьев данного узла.

В языке С для представления элемента динамических данных используют структуру ***struct.***

Структура - это одна или несколько переменных (возможно, различных типов), которые сгруппированы под одним именем. Эти переменные называются полями структуры. Доступ к отдельному полю структуры осуществляется посредством конструкции вида:

***имя-структуры . поле-структуры***

Для доступа к полям структуры могут использоваться указатели. Пусть p - указатель на структуру, тогда к отдельному элементу структуры можно обратиться следующим образом:

***p-> поле -структуры***

Приведем примеры определения элементов однонаправленного и двунаправленного списков, а также бинарного дерева.

Пример 1.

```
struct Node1        /* определение элемента однонаправленного списка */
{ double data;      /* data –информационное поле со значением вещественного типа*/
struct Node1 *next;};  /* next–адресное поле, содержащее указатель на следующий элемент списка */
```

Пример 2.

```
struct Node2            /* определение элемента двунаправленного списка */
{ int num;              /* num–информационное поле со значением целого типа*/
struct Node2 *next;     /* next–адресное поле, содержащее указатель на следующий элемент списка */
struct Node2 *prev; };  /* prev –адресное поле, содержащее указатель на предыдущий элемент списка */
```

Пример 3.

```
struct NodeTree     /* определение элемента бинарного дерева*/
{ char name[10];    /* name –информационное поле, содержащее 10 символов*/
struct Node2 *llink;  /* llink –адресное поле, содержащее указатель на левое поддерево */
struct Node2 *rlink; };  /* rlink –адресное поле, содержащее указатель на правое поддерево */
```

Рассмотрим стандартные библиотечные функции для динамического выделения и освобождения памяти:

```
#include <stdlib.h>
void *malloc (size_t size);
void *calloc (size_t nelem, size_t elsize) ;
void free (void *ptr) ;
```

Функция **malloc()** выделяет указанное аргументом **size** число байтов.
Функция **calloc()** выделяет память для указанного аргументом **nelem** числа объектов, размер которых **elsize**. Выделенная память инициализируется нулями. Если память не выделена, обе функции возвращают **NULL**.
Функция **free()** освобождает ранее выделенную память, указатель на которую передается через аргумент **ptr.**

Программа формирует двунаправленный список из 5 элементов (со значениями 1,2,3,4,5) и выводит его на экран.

Указатель на начало списка обозначен pbeg, на конец списка — pend.

```
#include <iostream.h>
struct Node
{ int d;
Node *next;
Node *prev; };

Node * first(int d);
void add(Node **pend, int d);

int main()
{
// Формирование первого элемента списка
Node *pbeg = first(1);
Node *pend = pbeg;

// Добавление в конец списка четырех элементов 2. 3. 4. и 5;
for (int i = 2; i<6; i++)   add(&pend, i);

// Вывод списка на экран
Node * pv= pbeg;
while (pv) {
cout<< pv->d << '  ';
pv = pv->next;  }

return 0;
}
// Функция формирования первого элемента
Node * first(int d)
{
Node *pv = new Node;
pv->d = d;  pv->next = 0;    pv->prev = 0;
```

```
    return pv;
}

// Функция добавления элемента в конец списка
    void add(Node **pend, int d)
    {
    Node *pv = new Node;
    pv->d = d;  pv->next = 0;  pv->prev = *pend;
    (*pend)->next = pv;
    *pend = pv;
    }
```

## 3.2. Задание на лабораторную работу

Разработать программу формирования списка и выполнения заданной операции с ним. В программе предусмотреть вывод элементов исходного списка и результирующего списка.

Коды вариантов задания приведены в таблице 3.1. Первая компонента кода варианта задания задает операцию, вторая- тип списка.

Таблица 3.1- Варианты задания

| **№ варианта** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| **код** | **1_1** | **2_1** | **3_1** | **4_1** | **5_1** | **6_1** |
| **№ варианта** | **7** | **8** | **9** | **10** | **11** | **12** |
| **код** | **7_1** | **8_1** | **9_1** | **10_1** | **11_1** | **12_1** |
| **№ варианта** | **13** | **14** | **15** | **16** | **17** | **18** |
| **код** | **1_2** | **2_2** | **3_2** | **4_2** | **5_2** | **6_2** |
| **№ варианта** | **19** | **20** | **21** | **22** | **23** | **24** |
| **код** | **7_2** | **8_2** | **9_2** | **10_2** | **11_2** | **12_2** |

**Заданная операция**:

1. Заменить элемент с номером ***k***.
2. Добавить элемент после элемента с номером ***k***.
3. Вставить элемент перед элементом с номером ***k***.
4. Удалить элемент с номером ***k***.
5. Удалить элементы, информационное поле которых равно ***x***.
6. Удалить элемент, предшествующий элементу с номером ***k***.
7. Переставить ***k*** -ый и ***k +1***-ый элементы списка.
8. Удалить элемент, следующий за элементом с номером ***k*** .
9. Переставить первый и ***k***-ый элементы списка.
10. Найти сумму положительных элементов списка и записать ее последним элементом.
11. Добавить элемент после элемента, информационное поле которого равно ***x***.
12. Заменить элемент, информационное поле которого равно ***x***.

**Тип списка:**

1. – однонаправленный;
2. – двунаправленный.

# 4. ЛАБОРАТОРНАЯ РАБОТА №3. ИНФОРМАЦИОННЫЕ СТРУКТУРЫ ОС. ТАБЛИЦА ДЕСКРИПТОРОВ ФАЙЛОВ

**Цель работы**: изучение информационных структур ОС, получение навыков обработки массивов данных и отладки программ циклической структуры.

## 3.1. Основные теоретические положения

Для хранения информации о файлах в ОС используется таблица дескрипторов файлов.

Дескриптор файла (ДФ)– это специальная структура, связанная с каждым файлом и содержащая описательную информацию. Эта информация используется файловой системой для выполнения соответствующих операций с файлом. ДФ формируется тогда, когда файл создается и обновляется тогда, когда файл перемещается, редактируется или когда к нему обращаются.

Конкретное содержание ДФ зависит от ОС, но фактически во всех файловых системах в дескрипторе содержатся три основных объекта данных: идентификатор файла, сведения о том, где файл хранится, и информация, управляющая доступом.

При программировании таблица ДФ может быть представлена массивом структур. Структура – это одна или несколько переменных (возможно, различных типов), которые сгруппированы под одним именем. Эти переменные называются полями структуры. Структуры могут быть вложенными.

Описание структуры начинается с ключевого слова ***struct*** и содержит список деклараций, заключенный в фигурные скобки. За словом ***struct*** может следовать имя, называемое ***тегом структуры***. Тег дает название структуре данного вида и далее может служить кратким обозначением той части декларации, которая заключена в фигурные скобки, т. е. по сути является именем типа. Декларация структуры, не содержащей списка переменных, не резервирует памяти: она просто определяет шаблон, или образец структуры. Однако если структура имеет тег, то им далее можно пользоваться при определении структурных объектов.

Доступ к отдельному полю структуры осуществляется посредством конструкции вида: ***имя-структуры . поле-структуры*** .

Для доступа к членам структуры могут использоваться указатели. Пусть p - указатель на структуру, тогда к отдельному элементу структуры можно обратиться следующим образом: ***p-> поле –структуры*** .

Рассмотрим пример определения структуры. Пусть анкета студента представлена структурой (***struct stud***) следующего вида:

```
struct stud {
            char fio[15];            /* фамилия студента*/
            struct data { int year;
                          int mon;
                          int day;
                        } d;              /* дата рождения */
             int m[3];                  /* оценки в сессию */
      };
```

Приведем функцию, которая печатает фамилии отличников и даты рождения. Параметрами функции являются массив анкет студентов g и их количество n.

```
void f(struct stud g[],int n)
{ int i;
  for(i=0;i<n;i++) {
  if(g[i].m[0]==5 && g[i].m[1]==5 && g[i].m[2]==5)
        printf("%s %d.%d.%d\n",
              g[i].fio,g[i].d.day,g[i].d.mon,g[i].d.year)
  }
}
```

Одна из функций файловой системы – отображение информации о файлах в упорядоченном виде.

Сортировка – упорядочение по возрастанию (или убыванию) заданного множества элементов.

Основное условие сортировки – экономное использование памяти, т.е. перестановки элементов должны выполняться на том же месте.

Рассмотрим простые методы сортировки на примере сортировки массива A, состоящего из n элементов .

Сортировка прямыми включениями: пусть элементы $a_1$, $a_2$,…, $a_{i-1}$ уже упорядочены. Очередной элемент $a_i$ вставляется на нужное место в отсортированную часть массива. Приведем фрагмент программного кода, реализующий рассмотренный метод.

```
for (int i = 1; i< n; i++)
    {
        x=A[ i ]; j=i;
        while (x< A[ j-1 ]) &&(j>=1)
        { A[ j ]= A[ j-1];
          j=j-1; }
        A[j]= x; }
```

Сортировка прямым выбором: выбирается наименьший элемент массива и меняется местами с первым элементом, затем просматриваются элементы, начиная со второго, и наименьший из них меняется местами (если надо) со вторым элементом, и так далее n-1 раз. На последнем, n-ом проходе цикла при необходимости меняются местами предпоследний и последний элементы массива. Фрагмент машинного кода приведен ниже.

```
/* просмотр массива n - 1 раз */
for (int i = 0; i< n-1; i++)
  {
     nmin= i;  min=a[i];
/* поиск наименьшего элемента */
      for (int i = i + 1; i< n; i++)
            if (A[j] < min )
              { nmin = j; min=A[nmin]; }
    a[nmin]=A[i];
    A[i] = min;
  }
```

Сортировка прямым обменом (пузырьковая сортировка): сравниваются пары соседних элементов, начиная с последней, и, при необходимости, элементы меняются местами. Таким образом, наименьший элемент продвигается в первую позицию. Фрагмент кода представлен ниже.

```
for (int i = 1; i< n; i++)
  for (int j = n; j>= i; j--)
      if (A[ j-1]> A[ j ] )
          {    x= A[ j-1];
               A[ j-1]= A[ j] ;
               A[ j] = x;    }
```

## 3.2. Задание на лабораторную работу

Разработать функцию сортировки таблицы дескрипторов файлов по заданному ключу. Таблицу дескрипторов представить массивом структур. Дескриптор должен содержать имя файла (не более 8 символов), тип файла (не более 3 символов), дату создания (в формате чч.мм.гг), количество обращений (целое число), размер файла (целое число), время последней модификации(в формате час.мин). В главной программе предусмотреть ввод и вывод исходных данных и результатов, а так же обращение к функции сортировки, глобальные переменные не использовать. Варианты задания приведены в таблице 3.1.

Первый символ кода варианта задания задает ключ, второй – метод сортировки.

Таблица 3.1- Варианты задания

| **№ варианта** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| **код** | **11** | **12** | **13** | **21** | **22** | **23** |
| **№ варианта** | **7** | **8** | **9** | **10** | **11** | **12** |
| **код** | **31** | **32** | **33** | **41** | **42** | **43** |
| **№ варианта** | **13** | **14** | **15** | **16** | **17** | **18** |
| **код** | **51** | **52** | **53** | **61** | **62** | **63** |

Тип ключа:

1. имя файла;
2. тип файла;
3. дата создания;
4. размер;
5. время последней модификации;
6. количество обращений.

Метод сортировки:

1. сортировка включениями;
2. сортировка выбором;
3. сортировка обменом.

## Библиографический список

1. Робачевский А.М. Операционная система UNIX/ А.М. Робачевский. – СПб.: BHV- Петербург, 2001. –528с.
2. Керниган Б. Язык программирования Си/ Б. Керниган, Д. Ритчи. – СПб.: Невский Диалект, 2001. –228с.
3. Таненбаум Э. Современные операционные системы. / Э. Таненбаум. – СПб.: Питер, 2011. –1120 с.
4. Вирт Н. Алгоритмы и структуры данных/ Н. Вирт– М.: Мир, 1989. – 360с.

# ПРИЛОЖЕНИЕ А.
# ОСНОВНЫЕ ПРЕОБРАЗОВАНИЯ *PR1NTF* и *SCANF*

Таблица А.1.– Основные преобразования printf

| Литера | Тип аргумента; вид печати |
|---|---|
| d,i | int; десятичное целое. |
| o | int; беззнаковое восьмеричное (octal) целое (без ведущегонуля). |
| X,x | int; беззнаковое шестнадцатиричное целое (без ведущих 0x и 0X), для 10...15 используются abcdef или ABCDEF |
| u | int; беззнаковое десятичное целое |
| c | int; одиночная литера. |
| s | char *; печатает литеры, расположенные до знака \0, или в количестве, заданном точностью. |
| f | double; [- ] от. *dddddd*, где количество цифр *d* задается точностью (по умолчанию равно 6). |
| E,e | double; [-1] *m. ddddddetxx* или [ - ] m. *ddddddEtxx*, где количество цифр *d* задается точностью (по умолчанию равно 6). |
| G,g | double; использует %e или %E, если экспонента меньше, чем -4, или больше или равна точности; в противном случае использует %f. «Хвостовые» нули и «хвостовая» десятичная точка не печатаются. |
| p | void *; указатель (представление зависит от реализации). |
| % | аргумент не преобразуется; печатается знак % |

Таблица А.2.– Основные преобразования scanf

| Литера | Вводимые данные; тип аргумента |
|---|---|
| d | десятичное целое; int *. |
| i | целое; int *-.- Целое может быть восьмеричным(с ведущим 0) или шестнадцатиричным (с ведущими 0x или 0X). |
| o | восьмеричное целое (с ведущим нулем или без него); int * |
| u | беззнаковое десятичное целое; unsigned int * |
| x | шестнадцатиричное целое (с ведущими 0x или 0X или без них);int*. |
| c | литеры; char *. Следующие литеры ввода (по умолчанию одна) размещаются в указанном месте. Обычный пропуск пробельных литер подавляется; чтобы прочесть очередную литеру, отличную от пробельной, используйте %1s. |
| s | стринг литер (без обрамляющих кавычек); char*, указывающий на массив литер, достаточный для стринга и завершающей литеры '\0', которая будет добавлена. |
| e,f,g | число с плавающей точкой, возможно, со знаком; обязательно присутствие либо десятичной точки, либо экспоненциальной части, а возможно, и обеих вместе; float *. |
| % | сам знак %, никакое присваивание не выполняется |

Таблица А.3.– Приоритеты и порядок вычисления операторов.

| Обозначение | Операция | Порядок выполнения |
|---|---|---|
| **( )**<br>**[ ]**<br>**->  .** | вызов функции<br>индексное выражение<br>операции доступа к полям структуры | Слева направо |
| **!**<br>**~**<br>**++  --**<br>**-**<br>**(тип)**<br>*****<br>**&**<br>**sizeof** | логическое “НЕ”<br>поразрядное “НЕ”<br>инкрементная и декрементная операции<br>унарный минус<br>приведение к типу<br>косвенная адресация<br>взятие адреса<br>получение размера объекта | **Справа налево** |
| ***  /**<br>**%** | умножение, деление,<br>остаток от деления | Слева направо |
| **+  -** | сложение, вычитание | Слева направо |
| **<<  >>** | поразрядные сдвиги | Слева направо |
| **<  <=**<br>**>  >=** | операции отношения | Слева направо |
| **==  !=** | операции отношения | Слева направо |
| **&** | поразрядное “И” | Слева направо |
| **^** | поразрядное исключающее “ИЛИ” | Слева направо |
| **\|** | поразрядное “ИЛИ” | Слева направо |
| **&&** | логическое “И” | Слева направо |
| **\|\|** | логическое “ИЛИ” | Слева направо |
| **? :** | тернарный оператор | **Справа налево** |
| **=**<br>**+=  -=**<br>***=  /=  %=**<br>**<<=  >>=**<br>**&=  ^=  \|=** | операции присваивания | **Справа налево** |
| **,** | операция запятая | Слева направо |

Приоритет операций убывает сверху вниз.

Операции из одной ячейки таблицы имеют одинаковый приоритет.

# ПРИЛОЖЕНИЕ Б.
# ФУНКЦИИ ДЛЯ РАБОТЫ СО СТРОКАМИ

В объявлениях приняты следующие обозначения:
dest – строка – результат,
source – строка – источник.

char *strcat (char * dest[1], const char * source[2]}. Функция дописывает строку dest в конец строки source;

char *strncat(char * dest, const char *source, unsigned 11), Функция дописывает не более n символов строки dest в конец строки source;

char *strchr (const char *source, int c) - поиск в строке source первого слева вхождения символа c, возвращает указатель на найденный символ или NULL;

char *strrchr (const char * source, int c) -~ поиск в строке source первого справа вхождения символа c, возвращает указатель на найденный символ или NULL; .

int strcmp (const char *sl, const char *s2) - сравнивает строки посимвольно, слева направо. Возвращает 0, если строки s1 и s2 равны, возвращает отрицательное число, если s1 меньше s2, возвращает положительное число, если s1 больше s2.

int strncmp (const char *sl, const char *s2, unsigned n) - сравнивает строки по первым n символам;

char *strcpy (char * dest, const char * source)-копирование строки source в строку dest;

char *strncpy(char * dest, const char *source, unsigned n) - копирование не более n первых символов в строку dest;

int strlen (const char *s) -возвращает длину строки s;

char *strset (char *s, int c)-заполняет вою строку s символом c;
char *strnset(char *s, int c, unsigned n) - заменяет первые n символов строки s на символ c;

char *strrev (char *s) -~ располагает все символы строки s, за исключением ноль-терминатора, в обратном порядке.