МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «ЭВМ и системы»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по вычислительной практике

«Основы программирования на языке Си»

для студентов специальности 1-40 02 01

«Вычислительные машины, системы и сети»

часть 2

Брест 2007

УДК 681.3

Методические указания содержат краткий теоретический курс программирования на языке Си, включающий темы: структуры и функции пользователя, файловый ввод-вывод, организация сортировки и поиска данных, консольный ввод-вывод.

Изложены цель и порядок выполнения каждой работы, приведены контрольные вопросы.

Методические указания предназначены для использования студентами специальности 1-40 02 01 "Вычислительные машины, системы и сети" при выполнении работ по вычислительной практике.

Требования к содержанию отчетов студентов изложены в 1-й части методических указаний

Издаётся в 2-х частях. Часть 2.

Список лит. 4 назв.

Составители: Костюк Д.А., к.т.н., доцент
Клементьева Л.А., ст. преподаватель
Петров Д.О., ассистент

РАБОТА № 1

СТРУКТУРЫ И ФУНКЦИИ

1. Цель работы

Изучить способы создания структур и функций пользователя в языке Си; научиться составлять программы обработки структурированных данных и разделять сложные алгоритмы на самостоятельные функциональные элементы.

2. Теоретические сведения

Структурированное представление данных в Си

Си позволяет объединять несколько разнотипных объектов (в т.ч. массивов и указателей) в рамках одного типа. Такой тип данных называется структурой, а переменные этого типа – структурными переменными.

Как и массив, структурная переменная представляет собой совокупность данных. Отличие структур от массивов в том, что к элементам структуры необходимо обращаться по имени, а не по номеру, и что различные элементы структуры не обязательно должны принадлежать одному типу.

Объявление структуры осуществляется с помощью ключевого слова struct, за которым идет имя типа, придуманное программистом для данной структуры, и далее список элементов, заключенных в фигурные скобки:

```
struct имя {тип элемента_1 имя элемента_1;
                  .........
                 тип элемента_n имя элемента_n;};
```

Элементы структуры называются полями. Именем поля может быть любой идентификатор. Как и при объявлении переменных, в одной строке можно записывать через запятую несколько идентификаторов одного типа:

```
sruct date {char day, month;
                int year;
               };
```

Следом за фигурной скобкой, заканчивающей список элементов, могут записываться структурные переменные данного типа, например:

```
sruct date {char day, month;
        int year;
       } a, b, c;
```

(при этом выделяется соответствующая память). Описание без последующего списка не выделяет никакой памяти; оно просто задает форму структуры. Введенное при описании структуры имя типа позже можно использовать для объявления переменных, не забывая указывать ключевое слово struct:

```
struct date days;
```

(объявлена структурная переменная days, которая имеет тип date).

При необходимости структуры можно инициализировать, помещая за описанием список начальных значений элементов.

Разрешается вкладывать структуры одна в другую, например:

```
struct man {char name[20], fam[20];
            struct date bd;
            int age; };
```

Определенный выше тип date включает три элемента: day, month, year, содержащий целые значения. Структура man включает элементы name, fam, bd и voz. Первые два name[20] и fam[20] - это символьные массивы из 20 элементов каждый. Переменная bd представлена составным элементом (вложенной структурой) типа data. Элемент age содержит значения целого типа int). Теперь можно определить переменные, значения которых принадлежат введенному типу:

```
struct man man_[100];
```

Здесь определен массив man_, состоящий из 100 структур типа man.

Чтобы обратиться к отдельному элементу структуры, необходимо указать его имя, поставить точку и сразу за ней написать имя нужного элемента, например:

```
man_[i].age = 19;
man_[j].bd.day = 22;
man_[j].bd.year = 1987;
```

Унарная операция & позволяет взять адрес структурной переменной. Предположим, что определена переменная day:

```
struct date day;
```

Другое определение

```
struct date *db;
```

объявляет db - указатель на структуру типа date.

Запишем выражение:

```
db = &day;
```

В этом случае для выбора элементов d, m, y структуры необходимо использовать конструкции:

```
(*db).day; (*db).month; (*db).year;
```

Действительно, поскольку db это адрес структурной переменной, то *db - сама структурная переменная. Круглые скобки здесь необходимы, так как точка имеет более высокий, чем звездочка, приоритет. Для аналогичных целей в языке Си предусмотрена специальная операция ->. Эта операция выбирает элемент структуры по указателю и позволяет представить рассмотренные выше конструкции в более простом виде:

```
db -> day; db -> month; db -> year;
```

Функции пользователя в Си

Программы на языке Си обычно состоят из большого числа отдельных функций (подпрограмм). Как правило, эти функции имеют небольшие размеры и могут находиться как в одном, так и в нескольких файлах. Все функции являются глобальными. В языке запрещено определять одну функцию внутри другой. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные

В любой программе на Си присутствует как минимум одна функция – функция main(), с которой начинается выполнение программы. Аналогичным образом могут быть описаны любые другие пользовательские функции.

В общем случае функции в языке Си необходимо объявлять в самом начале программы. Такое объявление, часто называемое прототипом функции, выглядит как отдельно описанный заголовок, в конце которого стоит знак «;». Прототип должен предшествовать любому использованию функции в программе, а полное описание функции может быть помещено как после тела программы (т.е. функции main( )), так и до него.

Функция объявляется следующим образом:

```
        тип имя_функции(тип имя_аргумента_1, тип
имя_аргумента_2, ...);
```

Тип перед именем функции определяет тип значения, которое возвращает функция. Если тип не указан, то предполагается, что функция возвращает целое значение int.

В прототипе функции для каждого ее параметра можно указать только его тип (например: тип функция (int, float, ...), а можно дать и его имя (например: тип функция (int a, float b, ...)). В полном описании функции имена параметров необходимы (для того, чтобы функция могла к ним обращаться).

ПРИМЕЧАНИЕ: в языке Си разрешается создавать функции с переменным числом аргументов. Тогда при задании прототипа вместо последнего аргумента указывается троеточие. Например, так объявлены библиотечные функции scanf и printf. Доступ к дополнительным параметрам при таком объявлении функции осуществляется средствами адресной арифметики, причем программист несет ответственность за правильное определение количества параметров и их типов.

Передача значения из вызванной функции в вызвавшую происходит с помощью оператора возврата return в следующей форме:

```
        return значение;
```

Таких операторов в функции может быть несколько, и тогда они фиксируют соответствующие точки выхода. Например:

```
int f(int a, int b)
{
    if (a > b)
    {
       printf("max = %d\n", a);
       return a;
    }
    printf("max = %d\n", b);
    return b;
}
```

Вызвать пользовательскую функцию можно теми же способами, что и стандартную:

```
c = f(15, 5);
c = f(d, g);
f(d, g);
```

Вызвавшая функция может при необходимости игнорировать возвращаемое значение. После слова return можно ничего не записывать; в этом случае вызвавшей функции никакого значения не передается. Управление передается вызвавшей функции и в случае выхода "по концу" (последняя закрывающая фигурная скобка тела функции).

В языке Си аргументы функции передаются по значению, т.е. вызванная функция получает временную копию каждого аргумента, а не его адрес. Это означает, что вызванная функция не может изменить значение переменной вызвавшей ее программы. Однако это легко сделать, если передавать в функцию не переменные, а их адреса. Например:

```
void swap(int *a, int *b)
{
      int *tmp = *a;
      *a = *b;
      *b = *tmp;
}
```

Вызов swap(&b, &c) (здесь подпрограмме передаются адреса переменных b и c) приведет к тому, что значения переменных b и c поменяются местами.

ПРИМЕЧАНИЕ: если в качестве аргумента функции используется имя массива, то передается только адрес начала массива, а сами элементы не копируются. Это происходит потому, что имя массива само по себе является указателем.

3. Порядок выполнения работы

3.1. Изучить теоретические сведения, изложенные в пункте 2. Получить у преподавателя вариант задания.
3.2. Разработать алгоритм решения задачи.
3.3. Разработать тестовые примеры для алгоритма, созданного в п. 3.2.
3.3. Написать программу, выполняющую вычисления в соответствии с разработанным алгоритмом, и реализовать ее на ЭВМ.
3.4. Протестировать программу.
3.5. Оформить отчет по работе в соответствии с правилами, приведенными в разделе "Цели и задачи проведения вычислительной практики".

4. Контрольные вопросы

4.1. Что такое *структура* в языке Си, как она объявляется в тексте программы?
4.2. Какие типы данных могут быть полями структуры?
4.2. Как осуществляется доступ к полям структуры?
4.3.Что такое *функция*, и как она объявляется в тексте программы?
4.4. В чем различие между прототипом функции и полным ее описанием?
4.5. Если тип значения, возвращаемого функцией, явно не задан, то какой тип будет указан по умолчанию?
4.6. Как вызвать фунцию?
4.7. Каким образом передается значение из вызванной функции?
4.8. Как происходит передача аргументов в функцию?
4.9. Как можно передать в функцию в качестве аргумента массив?

РАБОТА № 2

ФАЙЛОВЫЙ ВВОД/ВЫВОД В ТЕКСТОВОМ РЕЖИМЕ

1. Цель работы

Изучить средства организации буферизованного файлового ввода/вывода в языке Си, научиться разрабатывать программы, работающие с информацией на дисковых накопителях в текстовом режиме.

2. Теоретические сведения

Для файлового ввода/вывода в Си предусмотрены две основные группы функций:

- функции низкоуровневого ввода/вывода, использующие для доступа к файлам целочисленные файловые дескрипторы;
- функции более высокого уровня, осуществляющие буферизованный ввод/вывод с применением потоков.

Применение потоков позволяет использовать дополнительные удобства в виде буферизации и форматированного ввода/вывода. Поток в Си – это объект, служащий для доступа к файлам как к упорядоченной последовательности символов.

Поток представляется структурой типа FILE, с которой ассоциирован некоторый открытый файл. При необходимости несколько потоков могут ссылаться на один и тот же файл.

Любым операциям ввода/вывода должен предшествовать вызов функции открытия файла. Исключением являются три системных потока, которые открыты заранее: стандартный ввод, стандартный вывод и поток ошибок.

Эти потоки открываются системой еще перед началом работы программы. Для обращения к ним служат указатели на объекты типа FILE с именами, соответственно, stdin, stdout и stderr.

При открытии файлов указывается вид последующих операций ввода/вывода: чтение, запись, модификация (чтение и запись), добавление (запись в конец).

После открытия файла индикатор текущей позиции устанавливается в начало (на нулевой байт) при условии, что файл не открывали на добавление; в этом случае индикатор должен указывать на конец файла.

В дальнейшем индикатор текущей позиции смещается под воздействием операций чтения, записи и позиционирования, чтобы упростить последовательное продвижение по файлу.

Для открытия файла средствами высокоуровневого ввода/вывода служит функция fopen():

```
FILE *fopen(const char *path, const char *mode);
```

В качестве результата она возвращает указатель на объект, служащий для управления сформированным потоком (в случае неудачи возвращает пустой указатель, равный NULL).

Первый аргумент функции представляет собой указатель на строку с именем файла. Второй аргумент, mode, определяет режим открытия файла. Эта строка должна начинаться с одного из следующих символов:

- "r" - файл открывается для чтения, индикатор текущей позиции

устанавливается на начало файла;

- "w" - файл открывается для записи (или создается, если не существовал прежде), индикатор текущей позиции устанавливается на начало файла, размер файла усекается до нуля;
- "a" - файл открывается для дозаписи в конец (или создается, если не существовал прежде), индикатор текущей позиции устанавливается в конец файла,
- "r+" - файл открывается для чтения и записи, индикатор текущей позиции устанавливается на конец файла;
- "w+" - файл открывается для чтения и записи (или создается, если не существовал прежде), индикатор текущей позиции устанавливается на начало файла, размер файла усекается до нуля;
- "a+" - файл открывается для чтения и дозаписи в конец (или создается, если не существовал прежде), индикатор текущей позиции устанавливается в конец файла.

Строка mode может также включать в себя символ 'b', который должен находиться не на первой позиции. Это означает, что файл открывается не в текстовом, а в бинарном режиме. При программировании для операционных систем DOS и Windows разница между текстовым и двоичным режимами открытия файла заключается в интерпретации разделителей строк. В двоичном режиме разделитель строки «видится» Си как последовательность байт 0xD 0xA, а в текстовом – как служебный символ '\n'. При работе с текстовыми файлами целесообразно не использовать двоичный режим

Для закрытия открытого таким образом файла применяется функция

```
int fclose(FILE *stream);
```

Аргументом функции fclose() является указатель, который прежде вернула функция fopen(). При успешном завершении операции fclose() возвращает 0, в противном случае – константу EOF.

Существует также функция fcloseall(), которая вызывается без аргументов и закрывает все открытые в программе файловые потоки

Наиболее широко известными функциями форматированного ввода/вывода являются fprintf() и fscanf():

```
fprintf(FILE * поток, "управляющая строка", аргумент1,
аргумент2 ...);
fscanf(FILE * поток, "управляющая строка", аргумент1,
аргумент2 ...);
```

Данные функции аналогичны функциям printf() и scanf() за исключением дополнительного первого аргумента, в качестве которого указывается ранее открытый файловый поток.

Открытие файла: функция fopen

Для доступа к файлу применяется тип данных FILE. Это структурный тип, имя которого задано с помощью оператора typedef в стандартном заголовочном файле "stdio.h". Программисту не нужно знать, как устроена структура типа файл: ее устройство может быть системно зависимым, поэтому в целях переносимости программ обращаться явно к полям струтуры FILE запрещено. Тип данных "указатель на структуру FILE" используется в

программах как черный ящик: функция открытия файла возвращает этот указатель в случае успеха, и в дальнейшем все файловые функции применяют его для доступа к файлу.

Прототип функции открытия файла выглядит следующим образом:

```
FILE *fopen(const char *path, const char *mode);
```

Здесь path - путь к файлу (например, имя файла или абсолютный путь к файлу), mode - режим открытия файла. Строка mode может содержать несколько букв. Буква "r" (от слова read) означает, что файл открывается для чтения (файл должен существовать). Буква "w" (от слова write) означает запись в файл, при этом старое содержимое файла теряется, а в случае отсутствия файла он создается. Буква "a" (от слова append) означает запись в конец существующего файла или создание нового файла, если файл не существует.

В некоторых операционных системах имеются различия в работе с текстовыми и бинарными файлами (к таким системам относятся MS DOS и MS Windows, в системе Unix различий между текстовыми и бинарными файлами нет). В таких системах при открытии бинарного файла к строке mode следует добавлять букву "b" (от слова binary), а при открытии текстового файла – букву "t" (от слова text). Кроме того, при открытии можно разрешить выполнять как операции чтения, так и записи; для этого используется символ + (плюс). Порядок букв в строке mode следующий: сначала идет одна из букв "r", "w", "a", затем в произвольном порядке могут идти символы "b", "t", "+". Буквы "b" и "t" можно использовать, даже если в операционной системе нет различий между бинарными и текстовыми файлами, в этом случае они просто игнорируются.

Значения символов в строке mode сведены в следующую таблицу:

r	Открыть существующий файл на чтение
w	Открыть файл на запись. Старое содержимое файла теряется, в случае отсутствия файла он создаётся.
a	Открыть файл на запись. Если файл существует, то запись производится в его конец.
t	Открыть текстовый файл.
b	Открыть бинарный файл.
+	Разрешить и чтение, и запись.

Несколько примеров открытия файлов:

```
FILE *f, *g, *h;
. . .
/* 1. Открыть текстовый файл "abcd.txt" для чтения */
f = fopen("abcd.txt", "rt");

/* 2. Открыть бинарный файл "c:\Windows\Temp\tmp.dat"
 для чтения и записи */
g = fopen("c:/Windows/Temp/tmp.dat", "wb+");
```

```
/* 3. Открыть текстовый файл "c:\Windows\Temp\abcd.log"
 для дописывания в конец файла */
h = fopen("c:\\Windows\\Temp\\abcd.log", "at");
```

Обратите внимание, что во втором случае мы используем обычную косую черту / для разделения директорий, хотя в системах MS DOS и MS Windows для этого принято использовать обратную косую черту. Дело в том, что в операционной системе Unix и в языке Си, который является для нее родным, символ используется в качестве экранирующего символа, т.е. для защиты следующего за ним символа от интерпретации как специального. Поэтому во всех строковых константах Си обратную косую черту надо повторять дважды, как это и сделано в третьем примере. Впрочем, стандартная библиотека Си позволяет в именах файлов использовать нормальную косую черту вместо обратной; эта возможность была использована во втором примере.

В случае удачи функция fopen открытия файла возвращает ненулевой указатель на структуру типа FILE, описывающую параметры открытого файла. Этот указатель надо затем использовать во всех файловых операциях. В случае неудачи (например, при попытке открыть на чтение несуществующий файл) возвращается нулевой указатель. При этом глобальная системная переменная errno, описанная в стандартном заголовочном файле "errno.h, содержит численный код ошибки. В случае неудачи при открытии файла этот код можно распечатать, чтобы получить дополнительную информацию:

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

FILE *f = fopen("filnam.txt", "rt");
if (f == NULL) {
    printf("Ошибка открытия файла с кодом %d",errno);
    . . .
}
```

Закрытие файла: функция fclose

По окончании работы с файлом его надо обязательно закрыть. Система обычно запрещает полный доступ к файлу до тех пор, пока он не закрыт. (Например, в нормальном режиме система запрещает одновременную запись в файл для двух разных программ.) Кроме того, информация реально записывается полностью в файл лишь в момент его закрытия. До этого она может содержаться в оперативной памяти (в так называемой файловой кеш-памяти), что при выполнении многочисленных операций записи и чтения значительно ускоряет работу программы.

Для закрытия файла используется функция fclose с прототипом

```
int fclose(FILE *f);
```

В случае успеха функция fclose возвращает ноль, при ошибке – отрицательное значение (точнее, константу конец файла EOF, определенную в системных заголовочных файлах как минус единица). Отметим, что ошибка при закрытии файла - явление очень редкое (чего не скажешь в отношении открытия файла), так что анализировать значение, возвращаемое функцией fclose, в общем-то не обязательно. Пример использования функции fclose:

```
FILE *f;

f = fopen("tmp.res", "wb"); /* Открываем файл "tmp.res" */
if (f == 0) { /* При ошибке открытия файла
      Напечатать сообщение об ошибке */
    puts("Не могу открыть файл для записи");
    exit(1);  /* завершить работу программы с кодом 1 */
}

. . .

// Закрываем файл
if (fclose(f) < 0) {
    // Напечатать сообщение об ошибке
    puts("Ошибка при закрытии файла");
}
```

Пример: подсчет числа символов и строк в текстовом файле

В качестве содержательного примера использования рассмотренных выше функций файлового ввода приведем программу, которая подсчитывает число символов и строк в текстовом файле. Программа сначала вводит имя файла с клавиатуры. Для этого используется функция scanf ввода по формату из входного потока, для ввода строки применяется формат "%s. Затем файл открывается на чтение как бинарный (это означает, что при чтении не будет происходить никакого преобразования разделителей строк). Используя в цикле функцию чтения fread, мы считываем содержимое файла порциями по 512 байтов, каждый раз увеличивая суммарное число прочитанных символов. После чтения очередной порции сканируется массив прочитанных символов и подсчитывается число символов "\n" перевода строки, которые записаны в концах строк текстовых файлов как в системе Unix, так и в MS DOS или MS Windows. В конце закрывается файл и печатается результат.

```
/*
 Файл "wc.c"
 Подсчет числа символов и строк в текстовом файле
*/

#include <stdio.h>  /* Описания функций ввода-вывода */
#include <stdlib.h> /* Описание функции exit */

int main() {
    char fileName[256]; /* Путь к файлу */
    FILE *f;            /* Структура, описывающая файл */
    char buff[512];     /* Массив для ввода символов */
    size_t num;         /* Число прочитанных символов */
    int numChars = 0;   /* Суммарное число символов = 0 */
    int numLines = 0;   /* Суммарное число строк:= 0 */
    int i;              /* Переменная цикла */

    printf("Введите имя файла: ");
    scanf("%s", fileName);
```

```
    f = fopen(fileName, "rb"); /* Открываем файл на чтение */
    if (f == 0) {
        /*
           При ошибке открытия файла
           Напечатать сообщение об ошибке
        */
        puts("Не могу открыть файл для чтения");
        exit(1);
        /* закончить работу программы с кодом 1
           ошибочного завершения */
    }

    while ((num = fread(buff, 1, 512, f)) > 0) {
         /*
            Читаем
            блок из 512 символов. num - число реально
            прочитанных символов. Цикл продолжается, пока
            num > 0
         */

        numChars += num; /* Увеличиваем число символов */

        /* Подсчитываем число символов перевода строки */
        for (i = 0; i < num; ++i) {
            if (buff[i] == '/n') {
                ++numLines; /* Увеличиваем число строк */
            }
        }
    }

    fclose(f);

    /* Печатаем результат */
    printf("Число символов в файле = %d", numChars);
    printf("Число строк в файле = %d", numLines);

    return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы: она применяется к собственному тексту, записанному в файле "wc.c.

```
Введите имя файла: wc.c
Число символов в файле = 1635
Число строк в файле = 50
```

Форматный ввод-вывод: функции fscanf и fprintf

В отличие от функции бинарного ввода fread, которая вводит байты из файла без всякого преобразования непосредственно в память компьютера, функция форматного ввода fscanf предназначена для ввода информации с преобразованием ее из текстового представления в бинарное. Пусть информация записана в текстовом файле в привычном для человека виде (т.е. так, что ее можно прочитать или ввести в файл, используя текстовый редактор). Функция fscanf читает информацию из текстового файла и

преобразует ее во внутреннее представление данных в памяти компьютера. Информация о количестве читаемых элементов, их типах и особенностях представления задается с помощью формата. В случае функции ввода формат – это строка, содержащая описания одного или нескольких вводимых элементов. Форматы, используемые функцией fscanf, аналогичны применяемым функцией scanf, они уже неоднократно рассматривались. Каждый элемент формата начинается с символа процента "%". Наиболее часто используемые при вводе форматы приведены в таблице:

%d	целое десятичное число типа int (d - от decimal)
%lf	вещ. число типа double (lf - от long float)
%c	один символ типа char
%s	ввод строки. Из входного потока выделяется слово, ограниченое пробелами или символами перевода строки '\n'. Слово помещается в массив символов. Конец слова отмечается нулевым байтом.

Прототип функции fscanf выглядит следующим образом:

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

Многоточие здесь означает, что функция имеет переменное число аргументов, больше двух, и что количество и типы аргументов, начиная с третьего, произвольны. На самом деле, фактические агрументы, начиная с третьего, должны быть указателями на вводимые переменные. Несколько примеров использования функции fscanf:

```
int n, m; double a; char c; char str[256];
    FILE *f;
    . . .
    fscanf(f, "%d", &n);   /* Ввод целого числа */
    fscanf(f, "%lf", &a);  /* Ввод вещественного числа */
    fscanf(f, "%c", &c);   /* Ввод одного символа */
    fscanf(f, "%s", str);  /* Ввод строки (выделяется очеред-
                              ное слово из входного потока)
                           */
    fscanf(f, "%d%d", &n, &m); /* Ввод двух целых чисел */
```

Функция fscanf возвращает число успешно введенных элементов. Таким образом, возвращаемое значение всегда меньше или равно количеству процентов внутри форматной строки (которое равно числу фактических аргументов минус 2).

Функция fprintf используется для форматного вывода в файл. Данные при выводе преобразуются в их текстовое представление в соответствии с форматной строкой. Ее отличие от форматной строки, используемой в функции ввода fscanf, заключается в том, что она может содержать не только форматы для преобразования данных, но и обычные символы, которые записываются без преобразования в файл. Форматы, как и в случае функции fscanf, начинаются с символа процента "%". Они аналогичны форматам, используемым функцией fscanf. Небольшое отличие заключается в том, что форматы функции fprintf позволяют также управлять представлением данных, например, указывать количество позиций, отводимых под запись числа, или количество цифр после десятичной точки при выводе вещественного числа. Некоторые типичные примеры форматов для вывода приведены в следующей таблице

%d	вывод целого десятичного числа
%10d	вывод целого десятичного числа, для записи числа отводится 10 позиций, запись при необходимости дополняется пробелами слева
%lf	вывод вещественного числа типа double в форме с фиксированной десятичной точкой
%.3lf	вывод вещественного числа типа double с печатью трёх знаков после десятичной точки
%12.3lf	вывод вещественного числа типа double с тремя знаками после десятичной точки, под число отводится 12 позиций
%c	вывод одного символа
%s	конец строки, т.е. массива символов. Конец строки задается нулевым байтом

Прототип функции fprintf выглядит следующим образом:

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

Многоточие, как и в случае функции fscanf, означает, что функция имеет переменное число аргументов. Количество и типы аргументов, начиная с третьего, должны соответствовать форматной строке. В отличие от функции fscanf, фактические аргументы, начиная с третьего, представляют собой выводимые значения, а не указатели на переменные. Для примера рассмотрим небольшую программу, выводящую данные в файл "tmp.dat":

```
#include <stdio.h>  /* Описания функций ввода-вывода */
#include <math.h>   /* Описания математических функций */
#include <string.h> /* Описания функций работы со строками
*/

int main() {
    int n = 4, m = 6; double x = 2.;
    char str[256] = "Print test";
    FILE *f = fopen("tmp.dat","wt"); /*Открыть файл для записи*/
    if (f == 0) {
        puts("Не могу открыть файл для записи");
        return 1; /* Завершить программу с кодом ошибки */
    }
    fprintf(f, "n=%d, m=%d", m, n);
    fprintf(f, "x=%.4lf, sqrt(x)=%.4lf", x, sqrt(x));
    fprintf(f, "Строка \"%s\" содержит %d символов.",str,
strlen(str));
    fclose(f);  /* Закрыть файл */
    return 0;   /* Успешное завершение программы */
}
```

В результате выполнения этой программы в файл "tmp.dat" будет записан следующий текст:

```
n=6, m=4
x=2.0000, sqrt(x)=1.4142
Строка "Print test" содержит 10 символов.
```

В последнем примере форматная строка содержит внутри себя двойные

кавычки. Это специальные символы, выполняющие роль ограничителей строки, поэтому внутри строки их надо экранировать (т.е. защищать от интерпретации как специальных символов) с помощью обратной косой черты , которая, напомним, в системе Unix и в языке Си выполняет роль защитного символа. Отметим также, что мы воспользовались стандартной функцией sqrt, вычисляющей квадратный корень числа, и стандартной функцией strlen, вычисляющей длину строки.

Понятие потока ввода или вывода

В операционной системе Unix и в других системах, использующих идеи системы Unix (например, MS DOS и MS Windows), применяется понятие потока ввода или вывода. Поток представляет собой последовательность байтов. Различают потоки ввода и вывода. Программа может читать данные из потока ввода и выводить данные в поток вывода. Программы можно запускать в конвейере, когда поток вывода первой программы является потоком ввода второй программы и т.д. Для запуска двух программ в конвейере используется символ вертикальной черты | между именами программ в командной строке. Например, командная строка

```
ab | cd | ef
```

означает, что поток вывода программы ab направляется на вход программе cd, а поток вывода программы cd - на вход программе ef. По умолчанию, потоком ввода для программы является клавиатура, поток вывода назначен на терминал (или, как говорят программисты, на консоль). Потоки можно перенаправлять в файл или из файла, используя символы больше > и меньше <, которые можно представлять как воронки. Например, командная строка

```
abcd > tmp.res
```

перенаправляет выходной поток программы abcd в файл "tmp.res", т.е. данные будут выводиться в файл вместо печати на экране терминала. Соответственно, командная строка

```
abcd < tmp.dat
```

заставляет программу abcd читать исходные данные из файла "tmp.dat" вместо ввода с клавиатуры. Командная строка

```
abcd < tmp.dat > tmp.res
```

перенаправляет как входной, так и выходной потоки: входной назначается на файл "tmp.dat", выходной – на файл "tmp.res".

В Си работа с потоком не отличается от работы с файлом. Доступ к потоку осуществляется с помощью переменной типа FILE *. В момент начала работы Си-программы открыты три потока:

- stdin – стандартный входной поток. По умолчанию он назначен на клавиатуру;
- stdout – стандартный выходной поток. По умолчанию он назначен на экран терминала;
- stderr – выходной поток для печати информации об ошибках. Он также назначен по умолчанию на экран терминала.

Переменные stdin, stdout, stderr являются глобальными, они описаны в стандартном заголовочном файле "stdio.h. Операции файлового ввода-вывода

могут использовать эти потоки, например, строка

```
fscanf(stdin, "%d", &n);
```

вводит значение целочисленной переменной n из входного потока.

Строка

```
fprintf(stdout, "n = %d", n);
```

выводит значение переменой n в выходной поток.

Строка

```
fprintf(stderr, "Ошибка при открытии файла");
```

выводит указанный текст в поток stderr, используемый обычно для печати сообщений об ошибках. Функция perror также выводит сообщения об ошибках в поток stderr.

По умолчанию, стандартный выходной поток и выходной поток для печати ошибок назначены на экран терминала. Однако операция перенаправления вывода в файл > действует только на стандартный выходной поток. Например, в результате выполнения командной строки

```
abcd > tmp.res
```

обычный вывод программы abcd будет записываться в файл "tmp.res", а сообщения об ошибках по-прежнему будут печататься на экране терминала. Для того чтобы перенаправить в файл "tmp.log" стандартный поток печати ошибок, следует использовать командную строку

```
abcd 2> tmp.log
```

(между двойкой и символом > не должно быть пробелов!). Двойка здесь означает номер перенаправляемого потока. Стандартный входной поток имеет номер 0, стандартный выходной поток - номер 1, стандартный поток печати ошибок - номер 2. Данная команда перенаправляет только поток stderr, поток stdout по-прежнему будет выводиться на терминал. Можно перенаправить потоки в разные файлы:

```
abcd 2> tmp.log > tmp.res
```

Таким образом, существование двух разных потоков вывода позволяет при необходимости отделить мух от котлет, т.е. направить нормальный вывод и вывод информации об ошибках в разные файлы.

Функции scanf и printf ввода и вывода в стандартные потоки

Поскольку ввод из стандартного входного потока, по умолчанию назначенного на клавиатуру, и вывод в стандартный выходной поток, по умолчанию назначенный на экран терминала, используются особенно часто, библиотека функций ввода-вывода Си предоставляет для работы с этими потоками функции scanf и printf. Они отличаются от функций fscanf и fprintf только тем, что у них отсутствует первый аргумент, означающий поток ввода или вывода.

Строка

```
scanf(format, ...); /* Ввод из станд. входного потока */
```

эквивалентна строке

```
fscanf(stdin, format, ...); /* Ввод из потока stdin */
```

Аналогично, строка

```
printf(format, ...); /* Вывод в станд. выходной поток */
```

эквивалентна строке

```
fprintf(stdout, format, ...); /* Вывод в поток stdout */
```

Функции текстового преобразования sscanf и sprintf

Стандартная библиотека ввода-вывода Си предоставляет также две замечательные функции sscanf и sprintf ввода и вывода не в файл или поток, а в строку символов (т.е. массив байтов), расположенную в памяти компьютера. Мнемоника названий функций следующая: в названии функции fscanf первая буква f означает файл (file), т.е. ввод производится из файла; соответственно, в названии функции sscanf первая буква s означает строку (string), т.е. ввод производится из текстовой строки. (Последняя буква f в названиях этих функций означает форматный). Первым аргументом функций sscanf и sprintf является строка (т.е. массив символов, ограниченный нулевым байтом), из которой производится ввод или в которую производится вывод. Эта строка как бы стоит на месте файла в функциях fscanf и fprintf.

Фунции sscanf и sprintf удобны для преобразования данных из текстового представления во внутреннее и обратно. Например, в результате выполнения фрагмента

```
char txt[256] = "-135.76"; double x;
sscanf(txt, "%lf", &x);
```

текстовая запись вещественного числа, содержащаяся в строке txt, преобразуется во внутреннее представление вещественного числа, результат записывается в переменную x. Обратно, при выполнения фрагмента

```
char txt[256]; int x = 12345;
sprintf(txt, "%d", x);
```

значение целочисленной переменной x будет преобразовано в текстовую форму и записано в строку txt, в результате строка будет содержать текст "12345", ограниченный нулевым байтом.

Для преобразования данных из текстового представления во внутреннее в стандартной библиотеке Си имеются также функции atoi и atof с прототипами

```
int atoi(const char *txt);     /* текст => int */
double atof(const char *txt); /* текст => double */
```

Функция atoi преобразует текстовое представление целого числа типа int во внутреннее. Соответственно, функция atof преобразует текстовое представление вещественного числа типа double. Мнемоника имен следующая:

- atoi означает "character to integer";
- atof означает "character to float".

(В последнем случае float следует понимать как плавающее, т.е. вещественное, число, имеющее тип double, а вовсе не float!)

Прототипы функций atoi и atof описаны в стандартном заголовочном файле "stdlib.h", а не "stdio.h", поэтому при их использовании надо подключать этот файл:

```
#include <stdlib.h>
```

(вообще-то, это можно делать всегда, поскольку "stdlib.h" содержит описания многих полезных функций, например, функции завершения программы exit, генератора случайных чисел rand и др.).

Отметим, что аналогов функции sprintf для обратного преобразования из внутреннего в текстовое представление в стандартной библиотеке Си нет. Компилятор Си фирмы Borland предоставляет функции itoa и ftoa, однако эти функции не входят в стандарт и другими компиляторами не поддерживаются, поэтому пользоваться ими не следует.

Другие полезные функции ввода-вывода

Стандартная библиотека ввода-вывода Си содержит ряд других полезных функций ввода-вывода. Отметим некоторые из них.

Посимвольный ввод-вывод

`int fgetc(FILE *f);`	ввести символ из потока f
`int fputc(int c, FILE *f);`	вывести символ в поток f

Построковый ввод-вывод

`char *fgets(char *line,int size, FILE *f);`	ввести строку из потока f
`char *fputs(char *line, FILE *f);`	вывести строку в поток f

Позиционирование в файле

`int feof(FILE *f);`	проверить, достигнут ли конец файла f

Функция fgetc возвращает код введенного символа или константу EOF (определенную как минус единицу) в случае конца файла или ошибки чтения. Функция fputc записывает один символ в файл. При ошибке fputc возвращает константу EOF (т.е. отрицательное значение), в случае удачи - код выведенного символа c (неотрицательное значение).

В качестве примера использования функции fgetc перепишем рассмотренную ранее программу wc, подсчитывающую число символов и строк в текстовом файле:

```
/*
 Файл "wc1.c"
 Подсчет числа символов и строк в текстовом файле
 с использованием функции чтения символа fgetc
*/
```

```
#include <stdio.h>  /* Описания функций ввода-вывода */

int main() {
    char fileName[256]; /* Путь к файлу */
    FILE *f;            /* Структура, описывающая файл */
    int c;              /* Код введенного символа */
    int numChars = 0;   /* Суммарное число символов:= 0 */
    int numLines = 0;   /* Суммарное число строк:= 0 */

    printf("Введите имя файла: ");
    scanf("%s", fileName);

    f = fopen(fileName, "rb"); /* Открываем файл */
    if (f == 0) {
        /* При ошибке открытия файла
           Напечатать сообщение об ошибке
        */
        fprintf(stderr,"Не могу открыть файл для чтения");
        return 1; /* закончить работу программы с кодом 1 */
    }

    while ((c = fgetc(f)) != EOF) { /* Читаем символ */
        /* Цикл продолжается, пока c != -1 (конец файла) */

        ++numChars; /* Увеличиваем число символов */

        /* Подсчитываем число символов перевода строки */
        if (c == '\n') {
            ++numLines; /* Увеличиваем число строк */
        }
    }

    fclose(f);

    /* Печатаем результат
    printf("Число символов в файле = %d", numChars);
    printf("Число строк в файле = %d", numLines);

    return 0; /* Возвращаем код успешного завершения */
}
```

Пример выполнения программы wc1 в применении к собственному тексту, записанному в файле "wc1.c:

```
Введите имя файла: wc1.c
Число символов в файле = 1334
Число строк в файле = 44
```

Функция fgets с прототипом

```
char *fgets(char *line, int size, FILE *f);
```

выделяет из файла или входного потока f очередную строку и записывает ее в массив символов line. Второй аргумент size указывает размер массива для

записи строки. Максимальная длина строки на единицу меньше, чем size, поскольку всегда в конец считанной строки добавляется нулевой байт Функция сканирует входной поток до тех пор, пока не встретит симол перевода строки "\n", или пока число введенных символов не станет равным size - 1 Символ перевода строки "\n " также записывается в массив непосредственно перед терминирующим нулевым байтом. Функция возвращает указатель line в случае успеха или нулевой указатель при ошибке или конце файла.

Построчный вывод выполняется функцией fputs(char_pointer, fp), которая передает данные из массива символов с адресом char_pointer в файл, соответствующий указателю fp, до тех пор, пока не будет обнаружен завершающий строку нулевой байт.

Функция fputs возвращает отрицательное значение, обычно EOF, если обнаружена ошибка при выводе данных. Значение, которое возвращается после успешного вызова функции, зависит от реализации, но обычно отличается от EOF.

Следующая программа выполняет слияние файлов с помощью построчного ввода-вывода:

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

int main()
{
  FILE *sp, *tp;
  char line[81];

  if((sp = fopen("1.dat","r")) == NULL)
  {
    fprintf(stderr,"Ошибка при открытии файла 1.dat\n");
    exit(1);
  }

  if((tp = fopen("2.txt","a")) == NULL)
  {
    fprintf(stderr,"Ошибка при открытии файла 2.txt\n");
    exit(1);
  }

  while(fgets(line,sizeof(line)-1,sp))
  {
    fputs(line,tp);
  }

  fclose(sp);
  fclose(tp);

  return 0;
}
```

Вместо считывания по одному символу за прием эта программа считывает полную строку (до 80 символов). Имейте в виду, что функция fputs записывает в файл символы из строки line до тех пор, пока не обнаружит нулевой байт (помещенный в нее функцией fgets).

Функция feof с прототипом

```
int feof(FILE *f)
```

возвращает ненулевое значение при попытке произвести чтение за последним элементом файла f.

Например, если файл f содержит 10 байт и вы прочитали из него эти 10 байт, feof возвратит 0. И только после того, как вы попытаетесь прочесть 11-й байт из файла f, функция feof вернет ненулевое значение

3. Порядок выполнения работы

3.1. Изучить теоретические сведения, изложенные в пункте 2. Получить у преподавателя вариант задания.
3.2. Разработать алгоритм решения задачи.
3.3. Разработать тестовые примеры для алгоритма, созданного в п. 3.2.
3.3. Написать программу, выполняющую вычисления в соответствии с разработанным алгоритмом и реализовать ее на ЭВМ.
3.4. Протестировать программу.
3.5. Оформить отчет по работе в соответствии с правилами, приведенными в разделе "Цели и задачи проведения вычислительной практики".

4. Контрольные вопросы

4.1. Что такое поток и какие системные потоки вы знаете?
4.2. Как в программе на языке Си представляется поток?
4.3. Как открыть файл с функцией fopen?
4.4. В чем разница между открытием файла в, текстовом и в бинарном режиме?
4.4. В чем смысл операции закрытия файла. Как закрыть файл с помощью функции fclose?
4.5. Как осуществляется форматный ввод и вывод в поток? Как использовать функции fscanf и fprintf?
4.6. Каким образом можно ввести или вывести в файл единственный символ?
4.7. Как записывать или читать из файла строки символов?
4.8. Как использовать функции бинарного чтения/записи fread и fwrite?
4.9. Каким образом можно организовать произвольный доступ к компонентам файла?

РАБОТА № 3

ФАЙЛОВЫЙ ВВОД/ВЫВОД В БИНАРНОМ РЕЖИМЕ

1. Цель работы

Изучить средства организации буферизованного файлового ввода/вывода в языке Си, научиться разрабатывать программы, работающие с информацией на дисковых накопителях в бинарном режиме.

2. Теоретические сведения

Функции бинарного чтения и записи fread и fwrite

После того как файл открыт, можно читать информацию из файла или записывать информацию в файл. Рассмотрим сначала функции бинарного чтения и записи fread и fwrite. Они называются бинарными потому, что не выполняют никакого преобразования информации при вводе или выводе (с одним небольшим исключением при работе с текстовыми файлами, которое будет рассмотрено ниже): информация хранится в файле как последовательность байтов ровно в том виде, в котором она хранится в памяти компьютера.

Функция чтения fread имеет следующий прототип:

```
size_t fread(
    char *buffer,      // Массив для чтения данных
    size_t elemSize,   // Размер одного элемента
    size_t numElems,   // Число элементов для чтения
    FILE *f            // Указатель на структуру FILE
);
```

Здесь size_t определен как беззнаковый целый тип в системных заголовочных файлах. Функция пытается прочесть numElems элементов из файла, который задается указателем f на структуру FILE, размер каждого элемента равен elemSize. Функция возвращает реальное число прочитанных элементов, которое может быть меньше, чем numElems, в случае конца файла или ошибки чтения. Указатель f должен быть возвращен функцией fopen в результате успешного открытия файла.

Пример использования функции fread:

```
FILE *f;
double buff[100];
size_t res;

f = fopen("tmp.dat", "rb"); /* Открываем файл */
if (f == 0) { /* При ошибке открытия файла
                 Напечатать сообщение об ошибке */
    fprintf(stderr,"Не могу открыть файл для чтения");
    exit(1);  /* завершить работу с кодом 1 */
}

/* Пытаемся прочесть 100 вещественных чисел из файла */
res = fread(buff, sizeof(double), 100, f);
/* res равно реальному количеству прочитанных чисел */
```

В этом примере файл "tmp dat" открывается на чтение как бинарный, из него читается 100 вещественных чисел размером 8 байт каждое Функция fread возвращает реальное количество прочитанных чисел, которое меньше или равно 100.

Функция fread читает информацию в виде потока байтов и в неизменном виде помещает ее в память. Следует различать текстовое представление чисел и их бинарное представление! В приведенном выше примере числа в файле должны быть записаны в бинарном виде, а не в виде текста Для текстового ввода чисел надо использовать функции ввода по формату, которые будут рассмотрены ниже.

Внимание! Открытие файла как текстового с помощью функции fopen, например,

```
FILE *f = fopen("tmp.dat", "rt");
```

вовсе не означает, что числа при вводе с помощью функции fopen будут преобразовываться из текстовой формы в бинарную! Из этого следует только то, что в операционных системах, в которых строки текстовых файлов разделяются парами символами "/r/n" (они имеют названия CR и LF - возврат каретки и продергивание бумаги, Carriage Return и Line Feed), при вводе такие пары символов заменяются на один символ "/n" (продергивание бумаги). Обратно, при выводе символ "/n" заменяется на пару "/r/n". Такими операционными системами являются MS DOS и MS Windows В системе Unix строки разделяются одним символом "/n" (отсюда проистекает обозначение "/n", которое расшифровывается как new line). Таким образом, внутреннее представление текста всегда соответствует системе Unix, а внешнее - реально используемой операционной системе. Отметим также, что создатели операционной системы компьютеров Apple Macintosh выбрали, чтобы жизнь не казалась скучной, третий, отличный от двух предыдущих, вариант: текстовые строки разделяются одним символом "/r" – возврат каретки!

Такое представление текстовых файлов восходит к тем уже далеким временам, когда еще не было компьютерных мониторов и для просмотра текста использовались электрифицированные пишущие машинки или посимвольные принтеры. Текстовый файл фактически представлял собой программу печати на пишущей машинке и, таким образом, содержал команды возврата каретки и продергивания бумаги в конце каждой строки.

Функция бинарной записи в файл fwrite аналогична функции чтения fread. Она имеет следующий прототип:

```
size_t fwrite(
    char *buffer,      /* Массив записываемых данных */
    size_t elemSize,  /* Размер одного элемента */
    size_t numElems,  /* Число записываемых элементов */
    FILE *f           /* Указатель на структуру FILE */
);
```

Функция возвращает число реально записанных элементов, которое может быть меньше, чем numElems, если при записи произошла ошибка - например, не хватило свободного пространства на диске. Пример использования функции fwrite:

```
FILE *f;
double buff[100];
size_t num;
. . .

f = fopen("tmp.res", "wb"); /* Открываем файл "tmp.res" */
if (f == 0) { /* При ошибке открытия файла
                Напечатать сообщение об ошибке */
    fprintf(stderr,"Не могу открыть файл для записи");
    exit(1);  /* завершить работу программы с кодом 1 */
}

/* Записываем 100 вещественных чисел в файл */
res = fwrite(buff, sizeof(double), 100, f);
/* В случае успеха res == 100 */
```

Другие полезные функции ввода-вывода

Стандартная библиотека ввода-вывода Си содержит ряд других полезных функций ввода-вывода. Отметим некоторые из них.

Позиционирование в файле

`int fseek(FILE *f, long offset, int whence);`	установить текущую позицию в файле f
`long ftell(FILE *f);`	получить текущую позицию в файле f

Для произвольного доступа к записям, хранящимся в двоичном файле в виде структур фиксированной длины, понадобится функция, способная перемещать указатель чтения/записи внутри этого файла. Этим требованиям удовлетворяет функция fseek.

int fseek(FILE *fp, long offset, int whence),
где fp – указатель открытого файла,
offset – смещение указателя чтения/записи,
whence – одно из значений (SEEK_SET, SEEK_CUR, SEEK_END) определяющих позицию, относительно которой происходит смещение указателя чтения/записи.

SEEK_SET – смещение от начала файла
SEEK_CUR – смещение от текущей позиции
SEEK_END – смещение от конца файла

Обычно в сочетании с функцией fseek используется функция ftell. Эта функция возвращает текущую позицию в файле, с которой должны начинаться очередные чтение или запись. Функция ftell вызывается следующим образом:

```
lpos = ftell(fp);
```

где lpos имеет тип long, а fp – указатель файла.

3. Порядок выполнения работы

3.1. Изучить теоретические сведения, изложенные в пункте 2. Получить у преподавателя вариант задания.
3.2. Разработать алгоритм решения задачи.
3.3. Разработать тестовые примеры для алгоритма, созданного в п. 3.2
3.3. Написать программу, выполняющую вычисления в соответствии с разработанным алгоритмом, и реализовать ее на ЭВМ
3.4 Протестировать программу.
3.5. Оформить отчет по работе в соответствии с правилами, приведенными в разделе 'Цели и задачи проведения вычислительной практики".

4. Контрольные вопросы

4.1.Как использовать функции бинарного чтения/записи fread и fwrite?
4.2.Каким образом можно организовать произвольный доступ к компонентам файла?

РАБОТА № 4

ПОИСК И СОРТИРОВКА

1. Цель работы

Изучить наиболее распространенные алгоритмы поиска и сортировки, научиться применять их при разработке программ на языке Си.

2. Теоретические сведения

Поиск

Одно из наиболее часто встречающихся в программировании действий - поиск.

Последовательный поиск

Теперь познакомимся с последовательным поиском в целях нахождения некоторого значения в списке. Предположим, что мы ищем значение в списке целых чисел с использованием этого алгоритма.

Код для алгоритма последовательного поиска следующий:

```
// поиск в массиве a из n элементов для нахождения
соответствия с ключом
// использовать последовательный поиск, возвращать индекс
// соответствующего элемента массива или -1, если нет
соответствия
int SeqSearch(int list[ ], int n, int key)
{
  for (int i=0; i < n; i++)
    if (list[i] == key)
      return i;
      /*возвращать индекс соответствующего элемента */
  return -1;    /* поиск неудачный, возвращать -1 */
}
```

При определении эффективности алгоритма последовательного поиска различают поведение наилучшего и наихудшего случаев. Наилучшему случаю соответствует нахождение ключа в первом элементе списка. Время выполнения алгоритма при этом составляет 0(1). Наихудший случай имеет место, когда этот ключ не находится в списке или обнаруживается в конце списка. Он требует проверки всех n элементов и имеет порядок 0(n). Средний случай требует небольшого количества вероятностных рассуждений. Для случайного списка: совпадение с ключом может с одинаковой вероятностью появиться в любой позиции списка. После выполнения проверок большого количества элементов, средняя позиция совпадения – это срединный элемент (midpoint) n/2. Эта промежуточная точка анализируется после n/2 сравнений, что определяет ожидаемую стоимость поиска. По этой причине мы говорим, что средняя эффективность последовательного поиска составляет 0(n).

Бинарный поиск

Последовательный поиск применим для любого списка. Если список является упорядоченным, алгоритм, называемый бинарный поиск (binary search), предоставляет улучшенный метод поиска. Ваш опыт по нахождению номера в большом телефонном справочнике — это модель такого алгоритма. Зная нужные имя и фамилию, вы открываете справочник ближе к началу,

середине или концу, в зависимости от первой буквы фамилии. Вам может повезти, и вы сразу попадете на нужную страницу. В противном случае вы переходите к более ранней или более поздней странице в справочнике в зависимости от относительного местоположения имени человека по алфавиту. Например, если имя человека начинается с буквы R, а вы находитесь на странице с именами на букву T, вы переходите на более раннюю страницу. Процесс продолжается до тех пор, пока вы не найдете соответствие или не обнаружите, что этого имени нет в справочнике. Соответствующая идея применима к поиску в упорядоченном списке. Мы идем к середине списка и ищем быстрое соответствие ключа значению срединного элемента. Если нам не удается найти соответствия, мы смотрим на относительный размер ключа и значение срединного элемента и затем перемещаемся в нижнюю или верхнюю половину списка. В общем, если мы знаем, как упорядочены данные, мы можем использовать эту информацию, чтобы сократить время поиска.

Следующие шаги описывают алгоритм. Предположим, что список упорядочен, как массив. Индексами в концах списка являются: low = 0 и high = n-1, где n – это количество элементов в массиве

1. Найти индекс срединного элемента массива:

```
mid =(low+high)/2.
```

2. Сравнить значение в срединном элементе с key (ключ). Если совпадение найдено, возвращать индекс mid для нахождения ключа

```
if (A[mid] == key)
  return(mid);
```

Если A[mid] < key, совпадение должно происходить в диапазоне индексов d+1 . . . high, в правой половине рассматриваемого списка. Это верно, потому что список упорядочен. Новыми границами являются low=mid+1 и high

Если key < A[mid], совпадение должно происходить в диапазоне индексов low . . . mid - 1, в левой половине списка. Новыми границами являются low и high=mid-1.

Алгоритм уточняет местоположение совпадающего с ключом элемента, деля пополам длину интервала, в котором может находиться этот элемент, и затем выполняя тот же алгоритм поиска в меньшем подсписке. В конце концов, если искомый элемент не находится в списке, low превысит high, алгоритм возвращает индикатор сбоя -1 (совпадение не произошло).

Рассмотрим массив целых A. Этот пример дает выборку алгоритма для заданного ключа 33.

	0	1	2	3	4	5	6	7	8
A	-7	3	5	8	12	16	23	33	55

A	-7	3	5	8	12	16	23	33	55	low = 0 high = 8
	0	1	2	3	4 mid	5	6	7	8	mid = (0+8)/2 = 4 33 > A[mid]

A	-7	3	5	8	12	16	23	33	55	low = 5 high = 8
	0	1	2	3	4	5	6 mid	7	8	mid = (5+8)/2 = 6 33 > A[mid]

A	-7	3	5	8	12	16	23	33	55	low = 7 high = 8
	0	1	2	3	4	5	6	7 mid	8	mid = (7+8)/2 = 7 33 = A[mid] Найдено!

Заметьте, что этот алгоритм требует трех (3) сравнений. При линейном поиске в списке требуется восемь (8) сравнений.

Анализ бинарного поиска

Наилучший случай имеет место, когда совпадающий с ключом элемент находится в середине списка. При этом порядок алгоритма составляет O(1), так как требуется только одно тестирующее сравнение равенства. При наихудшем случае, когда элемент не находится в списке или определяется в последнем сравнении, имеем порядок $O(\log_2 n)$.

Основные алгоритмы сортировки массивов

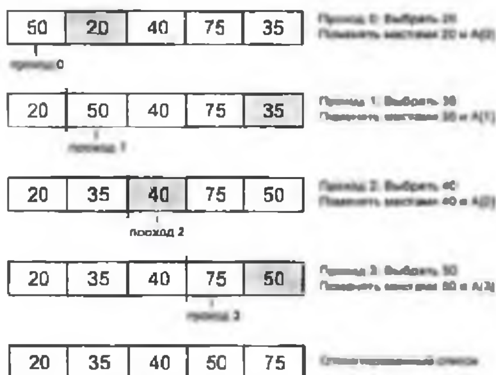
Начнем с трех алгоритмов, составляющих основу методики сортировки "на месте" по возрастанию. Для каждого алгоритма оценим его вычислительную эффективность.

Сортировка посредством выбора

Сортировка выбором моделирует наш повседневный опыт. Воспитататели детского сада часто используют эту методику, чтобы выстроить детей по росту. При этом самый маленький ребенок выбирается из неупорядоченной группы детей и перемещается в шеренгу, выстраиваемую по росту.

Для компьютерного алгоритма предполагается, что n элементов данных хранятся в массиве A и по этому массиву выполняется n-1 проход. В нулевом проходе выбирается наименьший элемент, который затем меняется местами с A[0]. После этого неупорядоченными остаются элементы A[1] ... A[n-1]. В следующем проходе просматривается неупорядоченная хвостовая часть списка, откуда выбирается наименьший элемент и запоминается в A[1]. В следующем проходе производится поиск наименьшего элемента в подсписке A[2] ... A[n-1]. Найденное значение меняется местами с A[2]. Таким образом, выполняется n-1 проход, после чего неупорядоченный хвост списка сокращается до одного элемента, который и является наибольшим.

Рассмотрим сортировку посредством выбора на массиве, содержащем пять целых чисел: 50, 20, 40, 75 и 35.



В i-ом проходе сканируется подсписок A[i] ... A[n-1] и переменной smallindex присваивается индекс наименьшего элемента в этом подсписке. Затем элементы A[i] и A[smallindex] меняются местами.

Вычислительная сложность сортировки посредством выбора

Сортировка посредством выбора требует фиксированного числа сравнений, зависящего только от размера массива, а не от начального распределения в нем данных. В i-м проходе число сравнений с элементами A[i+1]...A[n-1] равно

(n-1)-(i+1)+1=n-i-1

Общее число сравнений равно

$$\sum_{0}^{n-2}(n-1)-i=(n-1)^2-\sum_{0}^{n-2}i=(n-1)^2-\frac{(n-1)(n-2)}{2}=\frac{1}{2}n(n-1)$$

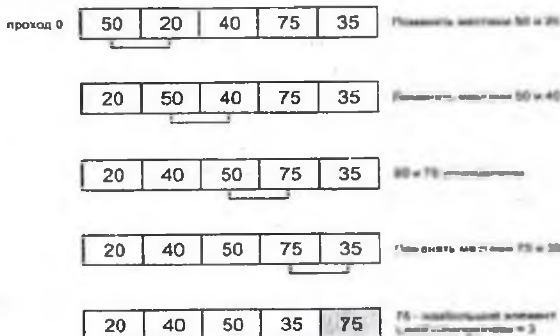
Сложность алгоритма, измеряемая числом сравнений, равна $O(n^2)$, а число обменов равно O(n). Наилучшего и наихудшего случаев не существует, так как алгоритм делает фиксированное число проходов, в каждом из которых сканируется определенное число элементов.

Сортировка методом пузырька

Для сортировки n-элементного массива A методом пузырька требуется до n-1 проходов. В каждом проходе сравниваются соседние элементы, и если первый из них больше второго, эти элементы меняются местами. К моменту окончания каждого прохода наименьший элемент поднимается к вершине текущего подсписка, подобно пузырьку воздуха в кипящей воде.

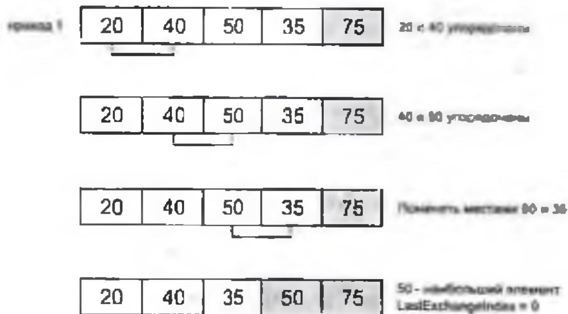
Рассмотрим эти проходы подробнее. Переменная lastExchangeIndex хранит последний участвующий в обмене индекс и приравнивается нулю в начале каждого прохода. В нулевом проходе сравниваются соседние элементы (A[0],A[1]), (A[1],A[2]), ... , (A[n-2],A[n-1]). В каждой паре (A[i],A[i+1]) элементы меняются местами при условии, что A[i+1] < A[i], а значение lastExchangeIndex становится равным i. В конце этого прохода наибольшее значение оказывается в элементе A[n-1], а значение lastExchangeIndex показывает, что все элементы в хвостовой части списка от A[lastExchangeIndex+1] до A[n-1] отсортированы. В очередном проходе сравниваются соседние элементы в подсписке A[0] – A[lastExchangeIndex]. Процесс прекращается при lastExchangeIndex=0. Алгоритм совершает максимум n-1 проход.

Рассмотрим пузырьковую сортировку на массиве, содержащем пять целых чисел: 50, 20, 40, 75 и 35.

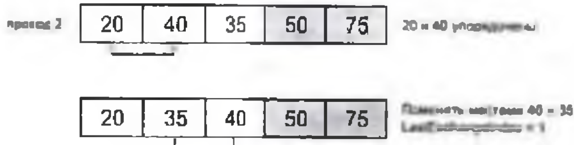


Поскольку lastExchangeIndex не равен нулю, процесс продолжается. В

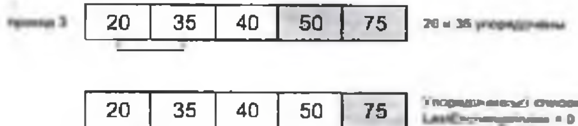
проходе 1 сканируется подсписок от A[0] до A[3]=lastExchangeIndex]. Новым значением lastExchangeIndex становится 2.



В проходе 2 сканируется подсписок от A[0] до A[2] и элементы 40 и 35 меняются местами. lastExchangeIndex становится равным 1.



В проходе 3 выполняется единственное сравнение (20 и 35). Обменов нет, lastExchangeIndex=0, и процесс прекращается.

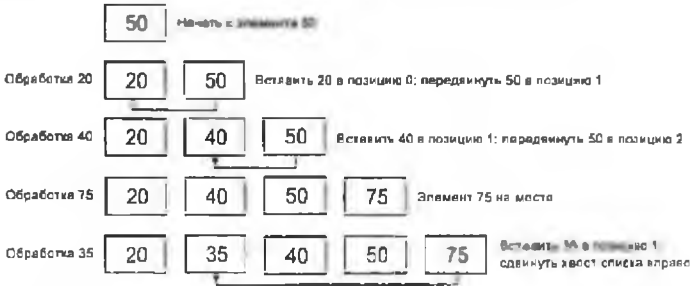


Вычислительная сложность сортировки методом пузырька

При пузырьковой сортировке сохраняется индекс последнего обмена во избежание избыточного просмотра. Это придает алгоритму заметную эффективность в некоторых особых случаях. Самое замечательное – это то, пузырьковая сортировка совершает всего один проход по списку, уже упорядоченному по возрастанию. Таким образом, в лучшем случае эффективность равна O(n). Худший случай для пузырьковой сортировки - когда список упорядочен по убыванию. Тогда необходимо выполнять все n-1 проходов В i-ом проходе производится (n - i - 1) сравнений и (n - i - 1) обменов. Сложность наихудшего случая составляет $O(n^2)$ сравнений и $O(n^2)$ обменов. Анализ общего случая затруднен из-за возможности пропуска некоторых проходов. Можно показать, что среднее количество проходов k равно O(n), следовательно, общее число сравнений равно $O(n^2)$. Даже если пузырьковая сортировка выполняется менее чем за n-1 проходов, это требует, как правило, большего числа обменов, чем при сортировке посредством выбора, и поэтому ее средняя производительность меньше. В общем случае сортировка посредством выбора превосходит пузырьковую за счет меньшего числа обменов.

Сортировка вставками

Сортировка вставками похожа на хорошо всем знакомый процесс тасования карточек с именами. Регистратор заносит каждое имя на карточку размером 127х76, а затем упорядочивает карточки по алфавиту, вставляя карточку в верхнюю часть стопки в подходящее место. Опишем этот процесс на примере нашего пятиэлементного списка A = 50, 20, 40, 75, 35.



Вычислительная эффективность сортировки вставками

Сортировка вставками требует фиксированного числа проходов. В n-1 проходах включаются элементы A[1]-A[n-1]. В i-ом проходе включения производятся в подсписок A[0]-A[i] и требуют в среднем i/2 сравнений. Общее число сравнений равно

$$1/2 + 2/2 + 3/2 + \ldots + (n-2)/2 + (n-1)/2 = n(n-1)/4$$

В отличие от других методов, сортировка вставками не использует обмены. Сложность алгоритма измеряется числом сравнений и равна $O(n^2)$. Наилучший случай - когда исходный список уже отсортирован. Тогда в i-ом проходе вставка производится в точке A[i], а общее число сравнений равно n-1, т.е. сложность составляет O(n). Наихудший случай возникает, когда список отсортирован по убыванию. Тогда каждая вставка происходит в точке A[0] и требует i сравнений. Общее число сравнений равно n(n-1)/2, т.е. сложность составляет $O(n^2)$.

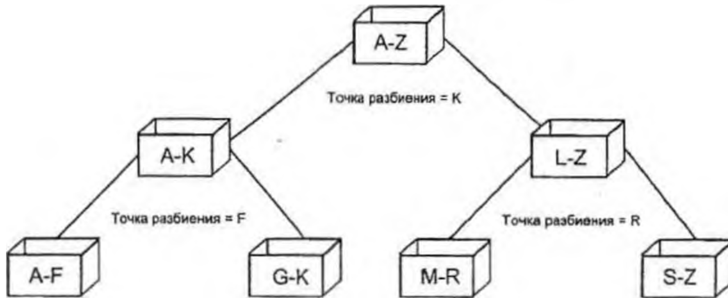
"Быстрая сортировка"

К этому моменту мы рассмотрели ряд $O(n^2)$-сложных алгоритмов сортировки массивов "на месте". "Быстрая сортировка", которую изобрел К. Хоар является самой быстрой из известных до сих пор.

Как и для большинства алгоритмов сортировки, методика "быстрой сортировки" взята из повседневного опыта. Чтобы отсортировать большую стопку алфавитных карточек по именам, можно разбить ее на две меньшие стопки относительно какой-нибудь буквы, например K. Все имена, меньшие или равные K, идут в одну стопку, а остальные - в другую. Затем каждая стопка снова делится на две. Например, на следующем рисунке точками разбиения являются буквы F и R. Процесс разбиения на все меньшие и меньшие стопки продолжается

В алгоритме "быстрой сортировки" применяется метод разбиения с определением центрального элемента. Так как мы не можем позволить себе удовольствие разбрасывать стопки по всему столу, как при сортировке алфавитных карточек, элементы разбиваются на группы внутри массива. Рассмотрим алгоритм "быстрой сортировки" на примере, а затем обсудим технические детали. Пусть дан массив, состоящий из 10 целых чисел:

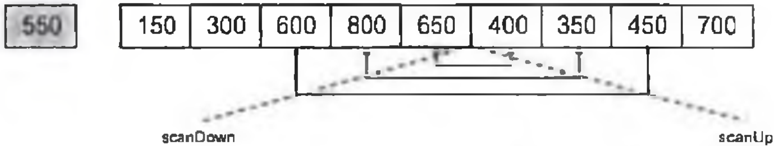
```
A = 800, 150, 300, 600, 550, 650, 400, 350, 450, 700
```



Фаза сканирования. Массив простирается от индекса low = 0 до индекса high = 9. Его середина приходится на индекс mid = 4. Первым центральным элементом является A[mid] = 550. Таким образом, все элементы массива A разбиваются на два подсписка: S_l и S_h. Меньший из них (S_l) будет содержать элементы, меньшие или равные центральному. Подсписок S_h будет содержать элементы большие, чем центральный. Поскольку заранее известно, что центральный элемент в конечном итоге будет последним в подсписке S_l, мы временно передвигаем его в начало массива, меняя местами с A[0] (A[low]). Это позволяет сканировать подсписок A[1]-A[9] с помощью двух индексов: scanUp и scanDown. Начальное значение scanUp соответствует индексу 1 (low+1). Эта переменная адресует элементы подсписка S_l. Переменная scanDown адресует элементы подсписка Sh и имеет начальное значение 9 (high). Целью прохода является определение элементов для каждого подсписка.

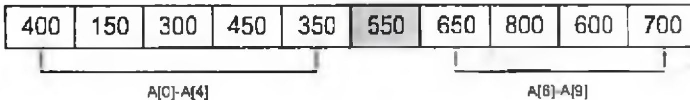
Оригинальность "быстрой сортировки" заключается во взаимодействии двух индексов в процессе сканирования списка. Индекс scanUp перемещается вверх по списку, а scanDown — вниз. Мы продвигаем scanUp вперед и ищем элемент A[scanUp] больший, чем центральный. В этом месте сканирование останавливается, и мы готовимся переместить найденный элемент в верхний подсписок. Перед тем как это перемещение произойдет, мы продвигаем индекс scanDown вниз по списку и находим элемент, меньший или равный центральному. Таким образом, у нас есть два элемента, которые находятся не в тех подсписках, и их можно менять местами (меняются местами A[scanUp] и A[scanDown]).

Этот процесс продолжается до тех пор, пока scanUp и scanDown не зайдут друг за друга (scanUp = 6, scanDown = 5). В этот момент scanDown оказывается в подсписке, элементы которого меньше или равны центральному. Мы попали в точку разбиения двух списков и указали окончательную позицию для центрального элемента. В нашем примере поменялись местами числа 600 и 450, 800 и 350, 650 и 400



Затем происходит обмен значениями центрального элемента A[0] с A[scan-Down].

В результате получился подсписок A[0] - A[4], элементы которого меньше элементов подсписка A[6] - A[9]. Центральный элемент (550) разделяет два подсписка, каждый из которых равен примерно половине исходного списка. Оба подсписка обрабатываются по одному и тому же алгоритму. Мы называем это рекурсивной фазой.



Рекурсивная фаза. Одним и тем же методом обрабатываются два подсписка: S_l (A[0]-A[4]) и S_h (A[5]-A[9]).

Подсписок S_l. Подсписок определяется диапазоном от индекса low = 0 до high = 4. Его середина приходится на индекс mid = 2. Центральным элементом является A[mid] = 300. Поменять местами центральный элемент A[low] и присвоить начальные значения индексам scanUp и scanDown:

```
scanUp = 1 = low +1
scanDown = 4 = high

Сканирование вверх останавливается на индексе 2
(A[2] > центрального элемента)
Сканирование вниз останавливается на индексе 1
(A[1] < центрального элемента)
```

Так как scanDown < scanUp, процесс останавливается. При этом scanDown является точкой разделения двух еще меньших подсписков: A[0] и A[2] – A[4]. Завершить обработку, поменяв местами A[scanDown] = 150 и

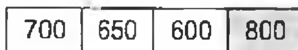
A[low] = 300. Заметьте, что положение центрального элемента дает нам одноэлементный и трехэлементный подсписки. Рекурсивный процесс прекращается на пустом или одноэлементном подсписках.

Подсписок S_h. Диапазон подсписка - от индекса low = 6 до high = 9. Его середина приходится на индекс mid = 7. Центральным элементом является A[mid] = 800. Поменять местами центральный элемент с A[low] и присвоить начальные значения индексам scanUp и scanDown:

```
scanUp = 7 = low + 1
scanDown = 9 = high

Сканирование вверх останавливается по достижении конца
списка (scanUp=10)
scanDown остается на начальной позиции
```

Так как scanDown < scanUp, процесс останавливается. При этом scanDown является точкой вставки центрального элемента. Завершить обработку, поменяв местами A[scanDown] = 700 и A[low] = 800. Обратите внимание, что положение центрального элемента дает нам трехэлементный и пустой подсписки. Рекурсивный процесс прекращается на пустом или одноэлементном подсписках.



Завершение сортировки. Обработать подсписок 400, 450, 350 (A[2] - A[4]). Центральным элементом является 450.

На фазе сканирования элементы выстраиваются в порядке 350, 400, 450. Для двухэлементного подсписка 350, 400 понадобится еще один рекурсивный вызов.

Обработать подсписок 700, 650, 600 (A[6] - A[8]). Центральным элементом является 650.

После сканирования элементы выстраиваются в порядке 600, 650, 700. Числа 600 и 700 образуют два одноэлементных подсписка.

"Быстрая сортировка" завершена. Результатом является следующий сортированный список:

```
A = 150, 300, 350, 400, 450, 550, 600, 650, 700, 800
```

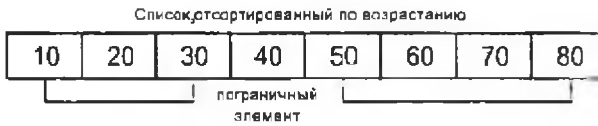
Вычислительная сложность "быстрой сортировки".

Общий анализ эффективности "быстрой сортировки" достаточно труден. Будет лучше показать ее вычислительную сложность, подсчитав число сравнений при некоторых идеальных допущениях. Допустим, что n — степень двойки, $n = 2^k$ ($k = \log_2 n$), а центральный элемент располагается точно посередине каждого списка и разбивает его на два подсписка примерно одинаковой длины.

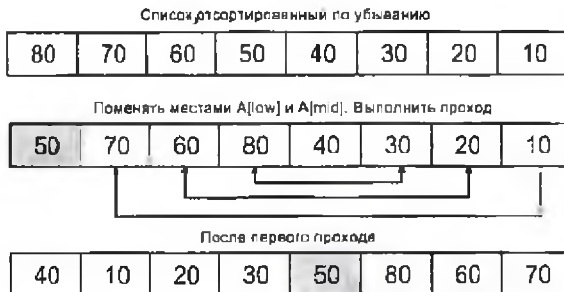
При первом сканировании производится n-1 сравнений. В результате создаются два подсписка размером n/2. На следующей фазе обработка каждого подсписка требует примерно n/2 сравнений. Общее число сравнений на этой фазе равно 2(n/2) = n. На следующей фазе обрабатываются четыре подсписка, что требует 4(n/4) сравнений, и т.д. В конце концов процесс разбиения прекращается после k проходов, когда получившиеся подсписки содержат по одному элементу. Общее число сравнений приблизительно равно

$$n + 2(n/2) + 4(n/4) + \ldots + n(n/n) = n + n + \ldots + n = n*k = n* \log_2 n$$

Для списка общего вида вычислительная сложность "быстрой сортировки" равна $O(n * \log_2 n)$. Идеальный случай, который мы только что рассмотрели, фактически возникает тогда, когда массив уже отсортирован по возрастанию. Тогда центральный элемент попадает точно в середину каждого подсписка.



Если массив отсортирован по убыванию, то на первом проходе центральный элемент находится в середине списка и меняется местами с каждым элементом как в нижнем, так и в правом подсписке. Результирующий список почти отсортирован, алгоритм имеет сложность порядка $O(n * \log_2 n)$.



Наихудшим сценарием для "быстрой сортировки" будет тот, при котором центральный элемент все время попадает в одноэлементный подсписок, а все прочие элементы остаются во втором подсписке. Это происходит тогда, когда центральным всегда является наименьший элемент. Рассмотрим последовательность 3, 8, 1, 5, 9. В первом проходе производится n сравнений, а больший подсписок содержит n-1 элементов. В следующем проходе этот

подсписок требует n - 1 сравнений и дает подсписок из n - 2 элементов и т.д. Общее число сравнений равно

$$n + n - 1 + n - 2 + \ldots + 2 = n(n+1)/2 - 1$$

Сложность худшего случая равна $O(n^2)$, т.е. не лучше, чем для сортировок вставками и выбором. Однако этот случай является патологическим и маловероятен на практике. В общем, средняя производительность "быстрой сортировки" выше, чем у всех рассмотренных нами сортировок.

3. Порядок выполнения работы

3.1. Изучить теоретические сведения, изложенные в пункте 2. Получить у преподавателя вариант задания.
3.2. Разработать алгоритм решения задачи.
3.3. Разработать тестовые примеры для алгоритма, созданного в п. 3.2.
3.3. Написать программу, выполняющую вычисления в соответствии с разработанным алгоритмом, и реализовать ее на ЭВМ.
3.4. Протестировать программу.
3.5. Оформить отчет по работе в соответствии с правилами, приведенными в разделе "Цели и задачи проведения вычислительной практики".

4. Контрольные вопросы

4.1. Какие методы поиска вы знаете? Объясните алгоритм бинарного поиска. Какова его эффективность?
4.2. Объясните алгоритм сортировки посредством выбора. Какова его эффективность?
4.3. Объясните алгоритм пузырьковой сортировки. Какова его эффективность?
4.4. Объясните алгоритм сортировки вставками. Какова его эффективность?
4.5. Объясните алгоритм «быстрой» сортировки (Хоара). Какова его эффективность?

РАБОТА № 5

КОНСОЛЬНЫЙ ВВОД/ВЫВОД

1. Цель работы

Изучить консольный ввод/вывод и научиться создавать простейший оконный интерфейс.

2. Теоретические сведения

Функции, прототипы которых помещены в файле <stdio.h>, осуществляют так называемый потоковый ввод/вывод. Используя их, невозможно выполнить, например, не отображаемый на экране ввод символов, "обойти" обработку реакции на нажатие комбинации клавиш Ctrl-Break, определить нажатие специальных клавиш.

Для выполнения таких действий используются функции Turbo C, прототипы которых помещены в файле <conio.h>. Они не являются частью ANSI-стандарта языка и входят в расширение Turbo C.

Функции, прежде всего, рассчитаны на построение простейшего оконного интерфейса и поэтому при любом выводе на экран (в том числе и "эха") дополнительно корректируются переменные, хранящие текущие координаты курсора активного окна (см. далее) Кроме того, имеется возможность управления цветом вывода.

Функции консольного ввода

char *cgets(char *str) - помещает в буфер, на начало которого указывает str, строку символов со стандартного ввода. Запись символов начинается с str[1]; str[0] должен содержать максимальное число символов, которое должно быть прочитано и записано в строку. Функция возвращает указатель на начало буфера str

Перечислим достоинства этой функции по сравнению с gets():

1) возможность определения при вводе длины строки;
2) защита при вводе от "лишних" символов, для которых компилятором не зарезервировано место;
3) возможность ввода за одно обращение к функции строк, длина которых превышает установленный по умолчанию буфер для стандартного ввода в 128 символов.

Приведем пример использования функции `cgets()` ввода строки до 254 символов за одно обращение:

```
/*
 Пример ввода строки 254 до символов за одно обращение.
*/

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

```
void main(void)
{
 char str[256];
 str[0] = 254; /* ограничитель максимальной длины строки */
 puts("Введите строку, длина которой может быть до 254
символов");
 cgets(str);
 printf("\nВведено %d символов строки:\n%s\n", str[1],
&str[2]);
}
```

int cscanf(char*format[, argument,...]) - выполняет форматированный ввод с клавиатуры. В отличие от функции scanf() не выполняет буферизацию ввода: все символы, вводимые с клавиатуры, доступны программе немедленно. Ввод пробела рассматривается как завершение ввода.

int getch(void) - выполняет ввод символа с клавиатуры. Turbo C не выполняет "эха" ввода. В этой связи полезна для организации интерфейса с пользователем, при котором нажатие той или иной клавиши вызывает немедленную реакцию программы без отображения введённого символа на экране.

int getche(void) - выполняет небуферизуемый ввод символа с клавиатуры. Turbo C "эхоирует" ввод на экране. Перевод строки происходит при достижении правой вертикальной границы текущего активного окна.

Приведем пример программы, иллюстрирующей применение функций getch() и getche() для определения нажатий не только ASCII-клавиш, но и специальных клавиш:

```
/*
  Демонстрирует ввод нажатий специальных клавиш.
*/

#include <conio.h>
#include <stdio.h>

void main(void)
{
  int ch;
  do
  {
    puts("Нажмите любую клавишу...");
    if(!(ch = getch()))
    {
      ch = getch();
      printf("Специальная клавиша. Расширенный скэн-код
%#u\n", ch);
    }
    else
      printf("Символьная клавиша %c (ASCII-код %#u)\n",ch,ch);
```

```
    puts("Продолжаете ? (y/n)");
  } while((ch=getch()) == 'y' || ch == 'Y');
}
```

char *getpass(const char *prompt) - выводит на экран ASCIIZ-строку, на начало которой указывает prompt, а затем принимает с клавиатуры без "эха" строку символов. Вводимые символы (не более 7) помещаются во внутреннюю статическую память. Функция возвращает указатель на внутреннюю статическую строку, переопределяемую каждым новым обращением к функции. Основное назначение данной функции - ввод паролей в программе без отображения их на экран.

Приведем пример программы, приглашающей задать пароль, принимающей его с клавиатуры, а затем сравнивающей его с заданным в программе эталоном.

```
/*
  Демонстрирует ввод и проверку санкционированности
доступа
  к программе по паролю.
  Прописные и строчные буквы в пароле и вводимой с
клавиатуры
  строке являются различными.
*/

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

int main(void)
{
  char * correct_string = "PaSsWoRd";
  char * input_string;
  input_string = getpass("Введите пароль:");
  if(strcmp(input_string,correct_string)!=0)
  {
    puts("\aНекорректный пароль. Завершение");
    return 1;
  }
  puts("Доступ разрешен.");
  return 0;
}
```

int kbhit(void) - проверяет, пуст ли буфер клавиатуры. Если в буфере есть символы, функция возвращает ненулевое значение, в противном случае она возвращает 0. Является удобным средством предотвращения "зацикливания" или "повисания" при ожидании невозможного в данный момент события. Кроме того, осуществляется проверка нажатия комбинации клавиш Ctrl-Break, что позволяет выполнить аварийное завершение программы.

int ungetch(int ch) - записывает непосредственно в буфер клавиатуры символ ch. Он будет доступен при выполнении следующей операции чтения с

консоли (функциями файла <conio.h>). Разрешает помещать только один символ, который не должен совпадать с константой EOF, описанной в файле <stdio.h>. В случае успеха функция возвращает ch; в противном случае возвращается -1.

Приведем пример программы, которая помещает строку символ за символом в буфер клавиатуры. Строка затем читается и выводится на экран.

```
/*
  Помещает строку символ за символом в буфер клавиатуры.
  Затем строка считывается и выводится на экран.
*/

#include <conio.h>
#include <stdio.h>

void main(void)
{
  char * string = "Тестовая строка для вывода на экран";
  while(*string)
  {
    ungetch(* string++);
    putchar(getch());
  }
}
```

Функции консольного вывода

В текстовом режиме экран можно представить как совокупность так называемых *текселов (texel - Text Element)*.

Каждому знакоместу экрана (текселу) в текстовом режиме соответствуют два байта. Первый байт хранит *ASCII-код символа*, а следующий за ним байт кодирует особенности отображения символа на экране: цвет *символа (Foreground Color)*, и *цвет фона символа (Background Color)*.

Turbo C предоставляет возможность изменять параметры текстового режима при помощи следующей функции, прототип которой находится в файле <conio.h> :

void textmode(int newmode) - изменяет текущий текстовый режим. Новый режим указывается единственным параметром newmode и может задаваться либо числом, либо с использованием символических констант, значения которых определяет перечислимый тип text_modes.

```
enum text_modes
{
  LASTMODE=-1, /* предыдущий текстовый режим */
  BW40=0,    /* 40 столбцов x 25 строк, черно-белое изображение */
  C40,       /* 40 столбцов x 25 строк, цветное изображение */
  BW80,      /* 80 столбцов x 25 строк, черно-белое изображение */
  C80,       /* 80 столбцов x 25 строк, цветное изображение */
  MONO=7,    /* 40 столбцов x 25 строк, монохромное изображение */
  C4350=64   /* 80 столбцов x 50 строк, цветное изображение */
};
```

Функции консольного вывода используют понятие активного окна экрана. *Активное окно* – это прямоугольная область экрана, в границах которой в данный момент работают функции. Описание активного окна (или, как часто говорят, *фрейм)* хранится во внутренней структурной переменной Turbo C. Установку параметров активного текстового окна выполняет функция window ().

void window(int l_t_col,int l_t_row, int r_b_col, int r_b_row) - описывает активное текстовое окно: первая пара аргументов задает столбец и строку левого верхнего угла, вторая пара - правого нижнего угла. Строки и столбцы нумеруются, начиная от 1. Поэтому, например, координаты левого верхнего и правого нижнего углов экрана в режимах "25 строк x 80 столбцов" задаются парами (1,1) и (80,25). Ось X направлена слева направо, а ось Y направлена сверху вниз. Обратите внимание на то, как в Turbo C задаются координаты углов: сначала столбец, затем строка. ()

Фрейм окна Turbo C имеет следующую структуру:

```
struct text_info {
 unsigned char winleft;       /* столбец, строка */
 unsigned char wintop;        /*левого верхнего угла */
 unsigned char winright;      /* столбец, строка */
 unsigned char winbottom;     /*правого нижнего угла */
 unsigned char attribute;     /* атрибуты */
 unsigned char normattr;      /* окна */
 unsigned char currmode;      /* текущий режим работы */
 unsigned char screenheight;  /* полная высота экрана */
 unsigned char screenwidth;   /* полная ширина экрана */
 unsigned char curx;          /* строка, столбец */
 unsigned char cury;          /* текущей позиции курсора */
};
```

Информация об активном окне доступна при выполнении функции gettextinfo():

void gettextinfo(struct textinfo *r) - заполняет поля структурной переменной по шаблону textjnfo, на которую ссылается. Шаблон структуры text_info, описывающей текущее окно экрана, содержится в заголовочном файле <conio.h>.

Функция window() инициализирует поля координат фрейма окна. Функции textcolor(), textbackground(), textattr() и другие управляют цветом отображаемых символов окна.

Далее в качестве примера приводится текст программы, образующей на экране окно ввода-вывода. Программа принимает с клавиатуры целое число и строку символов и затем выводит их на экран. Окно ввода-вывода занимает 25 столбцов и 10 строк в левом верхнем углу экрана. Предыдущее содержимое экрана сохраняется, а перед завершением программы восстанавливается.

```
/*
Образует окно ввода-вывода и демонстрирует его использование.
*/
#include <conio.h>
#include <alloc.h>
#include <string.h>

int main(void)
{
 unsigned number;
 char string[129], * buf_screen;
 /*
 Описание окна ввода-вывода.
*/
 window(1, 1, 25, 10);
 textbackground(BLUE); /* выбираем синий фон */
 textcolor(YELLOW);    /* выбираем желтый цвет для
символов */
 /*
 Сохранение предыдущего содержимого экрана для его
 восстановления после завершения работы программы.
 Сначала динамически распределяем память для хранения
 копии видеобуфера.
 */
 if((buf_screen = malloc(25 * 10 * 2)) == NULL)
 {
 cputs("Не хватает оперативной памяти для работы
программы.\xd\xa");
 return(-1);
 }
 gettext(1, 1, 25, 10, buf_screen); /* сохранение */
/*содержимого экрана */
 clrscr(); /* очистка экрана */
/* и "заливка" его синим цветом */
 cputs("Введите целое число: ");
 cscanf("%u", &number);
 cputs("\xd\xaВведите строку символов \xd\xa");
 cscanf("%128s", string);
 cprintf("\xd\xaВведено число %u и строка символов %s",
number,
string);
 cprintf("\xd\xaДлина строки - %d", strlen(string));
 /*
 Восстановление экрана.
 */
 cputs("\xd\xaДля продолжения нажмите любую клавишу");
 while(!kbhit()) {}
 puttext(1, 1, 25, 10, buf_screen);
 return 0;
}
```

Среди функций консольного ввода-вывода Turbo C текущей позицией курсора в окне управляет функция gotoxy().

void gotoxy(int x, int y) - устанавливает курсор в строку y и столбец x в текущем активном окне экрана. Верхний левый угол окна имеет координаты (1,1). При попытке позиционировать курсор за границы окна он останавливается на границе окна. Особенностью функции является то, что координаты x и y являются относительными, приведенными к левому верхнему углу. Например, если текущее окно было описано функцией *window(1,8,80,25)*, обращение gotoxy(5,5); установит курсор в пятый относительный столбец окна (совпадает с абсолютным столбцом 4, отсчитываемым от 0) в пятой относительной строке (так как верхняя строка окна задана равной 5, то абсолютная строка будет равна *5 + 5-1*, если отсчет строк ведется от 0).

Текущую позицию курсора в активном текстовом окне сообщают функции wherex() и wherey().

int wherex(void), int wherey(void) - сообщают столбец и строку текущей позиции курсора; возвращают целое число в диапазоне соответственно от 1 до 80 и от 1 до 25.

Кроме того, текущая позиция курсора в окне возвращается в полях curx и cury структурной переменной по шаблону textinfo, заполняемой при выполнении функции gettextinfo().

Функции вывода информации в окно экрана, в отличие от функций стандартного ввода-вывода они позволяют управлять цветом выводимых символов и не пересекают пределы активного в данный момент окна. При достижении правой вертикальной границы курсор автоматически переходит на начало следующей строки в пределах окна, а при достижении нижней горизонтальной грани выполняется скроллинг окна вверх

void clreol(void) - стирает в текстовом окне строку, на которую установлен курсор, начиная с текущей позиции курсора и до конца строки (до правой вертикальной границы окна).

void clrscr(void) - очищает все текстовое окно. Цвет "заливки" окна при очистке будет соответствовать значению, установленному символической переменной attribute в описании окна (структурная переменная по шаблону text_info). Функции управления цветом фона и символа описаны далее

void delline(void) - стирает в текстовом окне всю строку текста, на которую установлен курсор.

void insline(void) - вставляет пустую строку в текущей позиции курсора со сдвигом всех остальных строк окна на одну строку вниз. При этом самая нижняя строка текста окна теряется

int cprintf(const char *format, ...) - выполняет вывод информации с преобразованием по заданной форматной строке, на которую указывает format. Является аналогом функции стандартной библиотеки printf(), но выполняет вывод в пределах заданного окна. В отличие от printf () функция cprintf() иначе реагирует на специальный символ '\n' - курсор переводится на новую строку, но не возвращается к левой границе окна. Поэтому для

перевода курсора на начало новой строки текстового окна следует вывести последовательность символов CR-LF (0x0d.0x0a) • Остальные специальные символы воздействуют на курсор так же, как и в случае функций стандартного ввода-вывода. Функция возвращает число выведенных байтов, а не число обработанных полей, как это делает функция printf().

int cputs(const char *str) - выводит строку символов в текстовое окно, начиная с текущей позиции курсора. На начало выводимой ASCIIZ-строки указывает str. Является аналогом функции стандартной библиотеки puts(), но выполняет вывод в пределах заданного окна и при выводе не добавляет специальный символ '\n': Реакция cputs() на специальный символ '\n' аналогична реакции cprintf(): курсор переводится на новую строку, но не возвращается к левой границе окна. Поэтому для перевода курсора на начало новой строки текстового окна следует вывести последовательность символов CR-LF (0x0d,0x0a). Остальные специальные символы воздействуют на курсор так же, как и в случае функций стандартного ввода-вывода. Функция возвращает ASCII-код последнего выведенного на экран символа. В отличие от puts() в функции отсутствует возврат символа EOF. Другими словами, вывод происходит на экран в любом случае.

int movetext(int left, int top, int right,int bottom, int destleft, int desttop) - переносит окно, заданное координатами левого верхнего (left, top) и правого нижнего (right, bottom) углов, в другое место на экране, заданное координатами левого верхнего угла нового положения окна. Размеры окна по горизонтали и вертикали сохраняются. Все координаты задаются относительно координат верхнего левого угла экрана (1,1). Функция возвращает ненулевое значение, если перенос заданного окна выполнен. В противном случае возвращается 0. Функция корректно выполняет перекрывающиеся переносы, т.е. переносы, в которых прямоугольная область-источник и область, в которую окно переносится, частично покрывают друг друга.

int putch(int ch) - выводит символ в текущей позиции текстового окна экрана. Как и для функций cprintf(), cputs(), специальный символ '/n' вызывает только переход курсора на следующую строку текстового окна без возврата к его левой вертикальной границе. Остальные специальные символы воздействуют на курсор так же, как и для функций стандартного ввода-вывода.

int puttext(int left, int top, int right, int bottom,void *source) - выводит на экран текстовое окно, заданное координатами левого верхнего (left, top) и правого нижнего (right, bottom) углов. Символы и атрибуты располагаются в буфере, адрес начала которого задает указатель source. Другими словами, функция "открывает" (восстанавливает) текстовое окно экрана. Обычно используется вместе с функцией gettext(), выполняющей обратную операцию - запись в source символов/атрибутов, полностью описывающих все знакоместа текстового окна. Функция проверяет по заданным координатам окна, можно ли разместить окно на экране для текущего режима видеоадаптера и корректны ли эти коордиаты. В случае, когда окно успешно выведено, возвращается ненулевое значение.

void highvideo(void), void lowvideo(void), void normvideo(void) - функции задают соответственно использование повышенной и нормальной яркости для последующего вывода символов функциями файла <conio.h> Влияют на атрибут символа.

Описываемые далее функции управляют атрибутом символа. Атрибут задает битами 0-2 код цвета символа, бит 3 определяет повышение яркости, биты 4-6 задают код цвета фона символа, бит 7 определяет наличие или отсутствие мерцания символа. Turbo C дает возможность задать атрибут полностью либо задать только цвет символа или фона. Цвета могут задаваться либо числом, либо с использованием символических констант, значения которых определяет перечислимый тип COLORS:

```
enum COLORS {
 BLACK,       /* цвета нормальной яркости */
 BLUE,
 GREEN,
 CYAN,
 RED,
 MAGENTA,
 BROWN,
 LIGHTGRAY,
 DARKGRAY,    /* цвета повышенной яркости */
 LIGHTBLUE,
 LIGHTGREEN,
 LIGHTCYAN,
 LIGHTRED,
 LIGHTMAGENTA,
 YELLOW,
 WHITE
};
```

Яркие цвета могут задаваться только цвету символа. Кроме того, 7-й бит (бит мерцания) может быть задан как непосредственно в коде байта атрибута, так и с использованием символической константы BLINK, определяемой как 128 в <conio.h>. Следует отметить тот факт, что если для цвета фона выбираются цвета с кодами 8-15, это устанавливает в единицу бит мерцания символа в байте атрибута.

void textattr(int newattr) - устанавливает атрибут для функций, работающих с текстовыми окнами. Атрибут хранится в поле attribute структурной переменной по шаблону text_info, доступной через функцию gettextinfo(). Задаваемый атрибут может быть или числом, например, 0x70 - атрибут инверсного изображения (черные символы на светло-сером фоне), или формироваться из символических констант, значения которых задает тип COLORS. Например, для задания мерцающих ярко-красных символов на сером фоне атрибут можно сформировать следующим образом:

```
BLINK | (BLACK << 4)  | LIGHTRED
```

Тот же результат может быть получен и так: `DARKGRAY << 4) | LIGHTRED`

void textcolor(int newcolor) - задает цвет символов, не затрагивая установленный цвет фона. Цвет может быть или числом, или формироваться из символических констант, значения которых определяет перечислимый тип COLORS.

void textbackground(int newcolor) - задает цвет фона символов, не затрагивая установленный цвет символа. Цвет может быть или числом, или формироваться из символических констант, значения которых определяет перечислимый тип COLORS. Для цвета фона выбор ограничен значениями цветов 0-7. Если для цвета фона выбирается значение 8 -15, то символы будут мерцать, так как бит мерцания установится в единицу, но цвет фона будет соответствовать значениям 0-7.

int gettext(int left, int top, int right, int bottom, void *destin) - записывает в буфер destin символы и атрибуты текстового окна, заданного строкой и столбцом левого верхнего (left, top) и правого нижнего (right, bottom) углов. Первые два слова буфера занимают ширина и длина скопированного окна. Работает только в текстовых режимах видеоадаптера. Координаты задаются относительно верхнего левого угла экрана (1,1). В случае успеха возвращает ненулевое число

3. Порядок выполнения работы

3.1. Изучить теоретические сведения, изложенные в пункте 2. Получить у преподавателя вариант задания.

3.2. Разработать алгоритм решения задачи.

3.3. Разработать тестовые примеры для алгоритма, созданного в п. 3.2.

3.3. Написать программу, выполняющую вычисления в соответствии с разработанным алгоритмом и реализовать ее на ЭВМ.

3.4. Протестировать программу.

3.5. Оформить отчет по работе в соответствии с правилами, приведенными в разделе "Цели и задачи проведения вычислительной практики".

4. Контрольные вопросы

4.1. Чем потоковый ввод/вывод отличается от консольного?

4.2. Какие функции консольного ввода вы знаете?

4.3. Как можно определить, что была нажата не символьная клавиша, при помощи функций консольного ввода?

4.4. Как организована информация на экране в текстовом режиме, с точки зрения программиста?

4.5. Что такое активное окно, и как оно задается?

4.6. Что такое атрибут символа и как его задать?

4.7. Какие функции консольного вывода вы знаете?

4.8. Как при помощи функций gettext, puttext и movetext можно манипулировать прямоугольными областями на экране в текстовом режиме?

ЛИТЕРАТУРА

1. Березин Б.И., Березин С.Б. Начальный курс С и С++ М.:«ДИАЛОГ-МИФИ», 1998.
2. Болски М.И. Язык программирования СИ. Справочник. - М.: «Радио и связь», 1988.
3. Касаткин А. И., Вальвачев А.Н. Профессиональное программирование на Си: от Turbo C к Borland C++. Справочное пособие. - Мн.: «Высш. шк.», 1992.
4. Юлин В.А., Булатова И.Р. Приглашение к СИ - Мн.: «Высш. шк.», 1990.

УЧЕБНОЕ ИЗДАНИЕ

Составители: Костюк Дмитрий Александрович
Клементьева Лилия Александровна
Петров Дмитрий Олегович

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по вычислительной практике

«Основы программирования на языке Си»

для студентов специальности 1-40 02 01

«Вычислительные машины, системы и сети»

часть 2

Ответственный за выпуск: Петров Д.О.
Редактор: Строкач Т.В.
Компьютерная вёрстка: Кармаш Е.Л.
Корректор: Никитчик Е.В.

Подписано в печать 22.10.2007 г. Формат 60x84 $^1/_8$. Бумага «Снегурочка».
Гарнитура «Arial». Усл. печ. л. 2,79. Уч. изд. л. 3,0 . Тираж 100 экз. Зак. № 1092.

Отпечатано на ризографе
УО «Брестский государственный технический университет»
224017, г. Брест, ул. Московская, 267.