1.159. Отсортировать массив целых чисел с помощью библиотечной функции

```
void * bsearch( const void * search,
             const void * base, size_t nmeber, size_t size,
            int (*compar)(const void *,const void *));
```

из стандартной библиотеки `stdlib.h`, выполняющей бинарный поиск в массиве.

1.5 Задачи на работу с символьными переменными

При работе с символами на ЭВМ необходимо помнить, что в памяти компьютера хранится не изображение символа, а его целочисленный код. Как именно происходит восстановление внешнего вида символа по его коду зависит от типа кодирования. Наиболее простым является однобайтовое кодирование символов на основе табицы ASCII (American Standart Code Information Interchange). Символы с нулевого по 127 ASCII-таблицы фиксированы. Это связано с тем, что ранее стандартом была семибитная таблица ASCII-7, использовавшаяся для кодирования телефонограмм — код символа пробивался на бумажной ленте. При этом символы с 0-го по 31-й являются управляющими непечатными символами, например, 07 - звонок, 08 - шаг назад (BackSpace), 10 - перевод строки, 13 - возврат каретки. Вторая часть таблицы, т.е. начиная с 32-го символа (пробел) по 127-й символ (т.н. del-забой — в двоичном представлении семь единиц, что пробивалось на ленте как семь отверстий), содержит знаки препинания, цифры, заглавные и строчные буквы английского алфавита. При этом цифры '0',. . . ,'9' имеют коды с 48 по 57, буквы 'A', . . . , 'Z' — с 65 по 90, буквы 'a', . . . , 'z' — с 97 по 122. Отметим, что после '0' идет '1','2' и т.д., аналогично для заглавных 'A','B', . . . и строчных 'a','b', . . . букв. Третья часть, т.е. с 128-го символа по 255-й, зависит от конкретной реализации таблицы — типа выбранной кодовой страницы. К наиболее известным кириллическим таблицам относятся cp1251 (Windows-кодировка), koi-8, koi-8r, koi-8u (Linux-кодировки) и cp866 (DOS-кодировка). Файл, содержащий кириллицу, при неправильно указанной кодировке будет отображаться "кроказябрами". Для решения подобных проблем, актуальных для большинства национальных алфавитов, на данный момент установлен стандарт двухбайтового кодирования UNICODE, описание формата которого можно найти в соответствующей литературе. Большую популярность на данный момент имеет кодировка UTF-8 (Unicode Transformation Format, 8-bit), основанная на UNICODE–таблице, но использующая по необходимости для хранения символа от одного до 4-х байт. Какое именно количество байт кодирует текущий символ определяется по старшим битами, например, байты вида (0*******) кодируют символы с кодом от 0 до 127 из первой части таблицы ASCII. Отметим, что текстовый

процессор Word использует совсем иной подход для представления даже текстовой информации, а поэтому не стоит набирать С-программу в Word.

Для хранения символьных переменных в языке С используется тип `char`, занимающий один байт. Присвоить переменной конкретный символ можно двумя способами — либо явно, указаев его в одинарных кавычках, либо по его целочисленному коду, а для считывания/печати использовать тип формата `%c`, например

```
char c1,c2,c3,c4,c5;
c1='a'; c2=97; c3='\97'; c4='\n';
scanf("%c",&c5);
printf("%c %c %c\n", c2,c4,c5);
```

Символ `'\n'` с кодом 10 обеспечивает перевод каретки, поэтому в указанном примере выдача `c4` приведет к печати содержимого `c5` с новой строки. Аналогично все linux-редакторы, встречая `'\n'` в тексте, отрабатывают переход на новую строку. Однако, в системе Windows новая строка исторически кодируется парой символов `'\r'`(код 13) и `'\n'` (код 10), поэтому нужно быть готовым, что созданные под linux-системой файлы некоторыми windows-редакторами отобразятся в виде одной длинной строки, а редакторы под Linux при работе с файлами, созданными под OS Windows, дорисуют в конце каждой строки символ `'\r'` в виде знака `'^M'`. Подобные неудобства обычно решаются правильной настройкой параметров редактора.

В так как переменная типа `char` при переводе в целые числа может принимать значения от -128 до 127, то при работе с символами, имеющими коды от 128 до 255, лучше явно указать тип `unsigned char`. Если однобайтовая переменная для экономии места используется как целочисленная, то лучше ее явно описать как `signed char`.

1.160. Считайте с экрана символ и распечатайте его десятичный, восьмеричный и шестнадцатиричный коды, заключив текст в двойные кавычки.
Решение.

```
unsigned char sym;
unsigned char newline = '\n';
unsigned char b = '\'';
 while(scanf(''%c'', &sym)==1){
  printf('' %cchar=<%c>: dec code=<%d>,oct code=<%o>hex code=<%x>%c%c'',
  b, sym, (int)sym, (int)sym, (int)sym, b, newline);
 }
```

1.161. Считайте с экрана символ и если это заглавная латинская буква, то замените ее на соответствующую прописную.
Решение.

```
unsigned char sym;
while(scanf( ''%c'', &sym)==1){
 printf(''<%c> , code = <%d> '', sym, (int)sym);
 if(sym>='A' && sym<='Z'){
  sym = sym +'a' - 'A';
```

```
  printf(’’ new char = <%c> , new code = <%d>’’, sym, (int)sym);
 }
}
```

Замечание. Конструкция `scanf("%c",&sym)` не всегда корректно (с точки зрения пользователя) обеспечивает посимвольный ввод, т.к. набранная на экране буква и следом нажатая клавиша `Enter` могут быть обработаны как два последовательно введенных символа — буква и `’\n’`. Обычно это удается отфильтровать повторным считыванием

```
scanf("%c",&sym);
if(sym==’\n’)scanf("%c",&sym);
```

Однако, более надежным является ввод целой строки (см. раздел 1.6) с последующим посимвольным разбором.

1.162. С экрана вводится последовательность символов. Найдите целое число, которое получится после вычеркивания всех лишних (т.е. отличных от цифр) символов. Усложните задачу считая, что все встречаемые до первой цифры символы + и - определяют итоговый знак числа, т.е. плюсы игнорируются, а два минуса дают плюс.

1.163. Введите с клавиатуры символ и определите сколько раз он встречается в текстовом файле.

1.164. Вычислите, сколько раз каждый символ таблицы ASCII встречается в текстовом файле.
Решение.

```
#include <stdio.h>
void count (FILE * fin, int *nmb){
  unsigned char c;
  int i;
  for(i=0;i<256;i++) nmb[i]=0;
   while (fscanf(fin,"%c",&c)==1) nmb[(unsigned int)c]++;
return;
}
void prn (int *nmb){
  int i;
 for(i=0;i<256;i++){
  printf("code=<%5d> char=<%c> val=<%10d>\n",i,(unsigned char)i,nmb[i]);
 }
return;
}
```

1.165. Для каждого символа, содержащегося в строке, определите не более, чем за $O(n)$ действий, сколько раз он встречается в файле, содержащем n символов.

```
int  count (FILE * fin, unsigned char *str, int *nmb){
   unsigned char c;
   int i=0;
```

```
   int buf[256];
   for (i=0; i<256;i++) buf[i]=0;
   while (fscanf(fin, "%c", &c)==1) buf[(unsigned int)c]++;
   for (i=0; str[i]!='\0'; i++) nmb[i]=buf[(unsigned char)str[i]];
 return i;
}
```

Замечание. При работе с файлом как потоком символов удобно использовать функцию `int fgetc(FILE *)` из стандартной библиотеки `stdio.h`, возвращающую в случае успеха неотрицательный код считанного символа. Так как диапазон возвращаемых значений должен быть шире, чем диапазон всевозможных кодов символов от 0 до 255 (что позволит обрабатывать конец файла и различные сбои при считывании), то его тип — `int`. В случае ошибки при считывании, либо при достижении конца файла функция возвращает именованную целочисленную константу `EOF`, т.е. End Of File. Проверить, достигнут ли конец файла можно последующим вызовом `feof()`.

1.166. Напишите функцию, получающую в качестве параметра имя текстового файла и подсчитывающую количество заглавных латинских букв в файле.

Идеи реализации.

```
int c;
int num=0;
  while((c=fgetc(fin))!=EOF){ if( ('A'<c) && (c<'Z') ) num++;}
if(feof(fin)==0)printf("Error\n");
else
printf("End of file\n");
```

1.167. Напишите функцию, получающую в качестве параметра имя текстового файла и копирующую его содержимое в другой файл с заменой всех маленьких латинских букв на большие.

Идеи реализации.

```
int c;
  while((c=fgetc(fin))!=EOF){
    if('a'<c && c<'z')c = c+'A'-'a';
    if(fputc(c, fout)==EOF) break;
 }
```

1.168. Напишите функцию-фильтр, копирующую содержимое одного файла в другой файл, за исключением символов, содержащихся в заданном текстовом массиве. Функция должна иметь заголовок

```
void   copyfilter (FILE *fin, FILE *fout, unsigned char *badsym, int n);
```

где `fin,fout` — указатели на входной и выходной файлы, `badsym` — массив символов, которые не надо пропускать на выход, `n` — его длина.

1.169. Напишите функцию-фильтр, которая при каждом обращении возвращает очередной допустимый символ из заданного тестового файла. При

достижении конца файла функция возвращает значение `EOF`. Набор допустимых символов задается массивом — параметром функции. Функция должна иметь заголовок

```
int   filter (FILE *fin, unsigned char *goodsym, int n);
```

где `fin` — указатель на входной файл, `goodsym` — массив символов, которые нужно пропускать на выход, `n` — его длина.

Идеи реализации. Просмотр строки `goodsym` для каждого вновь прочитанного символа приведет к значительным затратам времени на обработку большого файла. Можно ввести в функцию массив `static int good[256]`, в который при обращении с ненулевым указателем `goodsym` записать единицы для допустимых и нули для остальных символов. Этот вызов можно рассматривать как инициализацию. Далее, обращаясь к функции с нулевым значением параметра `goodsym`, нужно проверять элементы массива `good` и выдавать ответ. Эта часть функции фактически сводится к выполнению операторов

```
int sym;
 while ( (sym=fgetc(fin))!=EOF)if(good[sym]) break;
return sym;
```

1.6 Алгоритмы работы с текстовыми строками

Простейшие функции работы с текстовыми строками входят в библиотеки всех современных компиляторов языка C (прототипы этих функций описаны в файле `string.h`). Однако самостоятельная реализация этих функций представляет собой очень полезное упражнение, особенно, если стремиться научиться решать подобные задачи наиболее эффективно. Напомним, что строкой в языке C называется последовательность элементов типа `char`, заканчивающаяся `'\0'`, т.е. символом с нулевым кодом. Такой символ не имеет изображения, а поэтому не может встречаться в текстовом файле. Это дает возможность в том числе использовать его в качестве управляющего символа, означающего конец символьного массива. Например:

```
char s1[10]; // s1[] - массив из 10-и символов;
s1[0]='a'; s1[1]='b'; s1[2]='\0';  // s1[] - массив из 10-и символов,
                                   // являющийся строкой из двух элементов;
s1[0]='\0';  // s1[] - массив из 10-и символов,
             // являющийся строкой из нуля элементов (пустой строкой)
```

Следующие присвоения допустимы только на этапе описания символьных массивов:

```
char s2[10] = "abc"; // s1[] - массив из 10-и элементов,
                     // являющийся строкой из трех элементов;
                     // символ s1[3]='\0' добавляется автоматически;
char s3[]="a";       // s3[] - массив из 2-х элементов,
                     // являющийся строкой из одного элемента;
```

Для считывания с экрана слова (т.е. последовательности символов, не содержащих пробелы и знаки табуляции) обычно используется функция
`scanf("%s",&str[0])`, что эквивалентно `scanf("%s",str)`.
При этом в конец считанной последовательности автоматически добавляется `'\0'`. Если до нажатия клавиши `enter` на экране было набрано несколько слов, то считается только первое, а остальные останутся в буфере ввода до очередного к нему обращения. Для ввода полной строки, набранной на экране, удобно использовать функцию `gets(&str[0])`, которая считывает и сохраняет все элементы, включая символ перехода на новую строку, и автоматически добавляет за ними символ `'\0'`. При этом всегда необходимо заранее выделить место, достаточное для сохранения в памяти считанной информации, иначе произойдет нарушение правил работы с памятью. Для защиты от ввода недопустимо больших строк обычно используют функцию
`char * fgets(char *, int , FILE *)`
с последующим выделением нужной информации посредством
`sscanf(const char*, const char*,...)`.
Отметим, что группы функции

`{scanf(),printf()} и {gets(),puts(),fgets(.,.,stdin), fputs(.,.,stdout)}`

не всегда совместимы, поэтому в рамках одного потока стоит пользоваться либо только первым, либо только вторым набором. Их смешение может привести к трудно контролируемым ошибкам.

1.170. Изучите работу функций `scanf()` и `printf()` по формату `"%s"`.
Решение.

```
#define N 64
char str[N];
while(scanf("%s", str)==1){
 for(i=0;i<N;i++){
  printf('' i=%d  char(str[i])=<%c> code(str[i])=%3d\n'',i,str[i], (int)str[i]);
 }
printf("%s",str);
}
```

Проанализируйте результат для входных данных `"123"`, `" 123  "`, `" 123  45 6"`, `" "`, а также некорректного ввода типа строки из ста единиц.

1.171. Реализуйте функцию, вычисляющую длину строки, т.е. количество символов до `'\0'`.
Решение.

```
int  my_strlen (cons char *s)
{
 int i=0;
 while (s[i]!='\0')i++;
return i;
}
```

1.172. Реализуйте функцию, копирующую одну строку в другую.
Решение. Одно из возможных решений может выглядеть так.

```
char * my_strcpy (char *dst, const char *src)
{
   char *s = dst;
   while ( *(dst++) = *(src++) );
   return s;
}
```

В данном случае компактность кода существенно выше его "прозрачности".

1.173. Реализуйте аналоги функций, выполняющих стандартные операции с текстовыми строками: strncpy(), strcmp(), strstr(), strcat(), strspn(), strcspn().

1.174. Напишите функцию, подсчитывающую количество слов в файле.

Указание. Например, можно использовать конструкцию вида

```
num=0;
while(fscanf(f,"%s",str)==1)num++;
```

1.175. Напишите функцию, подсчитывающую количество непустых строк в файле.

Указание. Используйте конструкцию типа

```
num=0;
char str[256];
while(fgets(str,256,f)!=NULL)if(str[0]!='\n' && str[0]!='\r') num++;
```

для строк, содержащих не более 255 символов в строке (включаяя `'\n'`, либо `'\r\n'`) и автоматически добавляемый функцией `fgets()` символ `'\0'`. Модифицируйте алгоритм на случай строк неизвестной длины.

1.176. Назовем словом группу символов, не содержащую внутри себя символов из заданного набора символов-разделителей. Примерами разделителей для слов обычного литературного текста могут служить символы "`.,;:!? ()[]-+*/"`" и т.п. Реализуйте функцию, которая при каждом обращении к ней выделяет из указанного текстового файла очередное слово. Функция должна иметь прототип

```
int  GetWord (FILE *f, char *word, char *delim);
```

где `f` — указатель на входной файл, `word` — указатель на буфер для очередного слова, `delim` — указатель на сроку с символами-разделителями. Возвращаемое значение — 0, если слово прочитано, и -1 в случае конца файла или какого-либо другого отказа.

1.177. В файле, содержащем литературный текст, возможны переносы слов между строками. Модифицируйте функцию `GetWord` из задачи 1.176 так, чтобы она обнаруживала и правильно выводила перенесенные слова. Можно считать, что слово разбито переносом на две части, если за допустимыми (буквенными) символами идет символ "`-`", за которым следуют один или несколько символов перевода строки, и далее, возможно, несколько пробелов до следующего допустимого символа.

1.178. Используя функцию `GetWord` из задачи 1.176 выполните следующую обработку текстового файла:

1) подсчитайте количество слов в исходном файле;

2) подсчитайте максимальную, минимальную и среднюю длину слов;

3) подсчитайте среднее количество слов в одном предложении (предложение — это последовательность слов, оканчивающаяся одним из символов ".!?");

4) выберите и напечатайте все слова, начинающиеся с заглавной буквы.

1.179. Напишите функцию сортировки массива слов, учитывая при сравнении только первую букву.

Указание.

```
#include <stdlib.h>
int cmp(const void * a, const void * b);
int main(){
  const char * mas[] = {"atan()", "sin()", "abs()", "aaa()"};
  int n=4;
  for(i=0;i<n;i++) printf("Исходный массив: %s ", mas[i]);
  printf("\n'');
  qsort((void*)mas, n, sizeof(char*), cmp);
  for(i=0;i<n;i++) printf("Итоговый массив: %s ", mas[i]);
  printf("\n'');
 return 0;
}
int cmp(const void * a, const void * b) {
    char **ca = (char **)a;
    char **cb = (char **)b;
printf("\n cmp:<%s, %s>\n", *ca, *cb);
    return (*ca)[0]-(*cb)[0];
    }
```

В результате получим массив, упорядоченный с учетом только первой буквы.

1.180. Модифицируйте решение задачи 1.179 для сортировки массива слов в лексикографическом порядке.

1.181. Реализуйте функцию сортировки массива слов в лексиграфическом порядке, не используя `qsort`. Для создания и хранения массива используйте следующую конструкцию.

```
 int Nwords = 1000;
 int Lmax = 30;
 char **mas;
 int n;

 mas=(char**)malloc(Nwords*sizeof(char*));
 if(mas==NULL)exit(1);
 for(n=0;n<Nwords;n++){
   mas[n]=(char*)malloc(Lmax*sizeof(char));
```

```
   if(mas[n]==NULL)exit(2);
 }

 n=0;
 while(n<Nwords){
  if(fscant(in,"%s",mas[n])!=1)break;
  n++;
 }
```

1.182. Модифицируйте решение задачи 1.181 так, чтобы для хранения каждого слова выделялось в точности требуемое количество байт.

Указание. Завести буфер `char buf[Lmax]` достаточной длины, в буфер считывать очередное `n`-е слово, вычислять его длину, выделять требуемое место `mas[n]` и затем копировать из буфера в массив. Предусмотреть защиту от вводимых слов длины больше, чем `Lmax`.

1.183. По заданному текстовому файлу сформируйте файл-словарь, содержащий все слова из исходного файла, записанные в алфавитном порядке по одному в строке.

Идеи реализации. Считая, что максимальное количество слов нам известно, организуем хранение слов на базе массива указателей на слова, а при добавлении очередного слова реализуем алгоритм сортировки вставками с бинарным поиском на основе функции сравнения `strcmp()`.

1.184. С экрана последовательно вводятся строки следующего формата

<Фамилия> <Имя> <id> <dq>

Известно, что количество строк не превосходит тысячи. Здесь <Фамилия> и <Имя> - последовательности, содержащие не более 20 и 15 символов соответственно; <id> - положительное целое число, являющееся уникальным идентефикатором, <dq> - рейтинг. Требуется считать информацию, сохранив ее в памяти ЭВМ, и выдать на экран имена и <id> всех пользователей, имеющих значение <dq> не менее 1000, упорядочив их по алфавиту.

Указание. Для хранения вводимой информации удобно создать массив структур подходящего типа, далее заполнить его данными с экрана, произвести контрольную печать, отсортировать по алфавиту по полю <Фамилия>, распечатать требуемую информацию. Описание структуры и прототипов функций:

```
struct {
 char Fname[21];
 char Lname[16];
 int id;
 int dq;
} DataStr;
int readData(struct DataStr *Data);
void prnData(struct DataStr *Data, int N);
```

Основной блок:

```
int Nmax = 1000;
```

```
int N;
struct DataStr * Data;
Data = (struct DataStr *)malloc(Nmax*sizeof(struct DataStr));
if(Data==NULL)exit(1);
N = readData(Data);
prnData(Data,N);
```

Рабочие функции:

```
int readData(struct DataStr *Data){
int n=0;
while(fscanf(stdin,"%s %s %d %d",
 Data[n].Fname,  Data[n].Lname),
 &(Data[n].id),  &(Data[n].dq)==4)n++;
return n;
}
void prnData(struct DataStr *Data, int N){
int n=0;
for(n=0;n<N;n++){
 fprintf(stdout,"%s %s %d %d\n",
 Data[n].Fname,  Data[n].Lname,
 Data[n].id,  Data[n].dq);
}
return;
}
```

1.185. С экрана последовательно вводятся строки, см. задачу 1.184, следующего формата

<Фамилия> <Имя> <id> <dq>

Требуется считать информацию, сохранив ее в памяти ЭВМ, и выдать на экран имена и <id> всех пользователей, имеющих значение <dq> не менее 1000, упорядочив их по алфавиту. Если встречаются пользователи с одинаковой фамилией, то их необходимо упорядочить по содержимому <id>.

Указание. Отсортировать введенный массив по совокупности полей <Фамилия>+<id>, реализовав соответствующую функцию сравнения, и распечатать пользователей с допустимым <dq>.

1.7 Разбор чисел и битовые операции

При решении задач данного раздела можно анализировать остатки от деления целых чисел, полученных с помощью оператора `%`, либо использовать битовые операции `<<, >> , ~, ^, |, &`, соответственно обеспечивающие сдвиг влево, вправо, отрицание (not), исключающее или (xor, сложение по модулю 2), побитовое или (or, логическое сложение), побитовое и (and, логическое умножение).