[1. Файлы в C.](#_Toc515770627)

[1.1. Файлы и потоки.](#_Toc515770628)

[1.2. Связь с файлами.](#_Toc515770629)

[1.3. Закрытие потоков.](#_Toc515770630)

[1.4. Обработка ошибок в C и C++](#_Toc515770631)

[1.4.1. Переменная errno и коды ошибок](#_Toc515770632)

[1.5. Переключение и работа с файлами.](#_Toc515770633)

[1.5.1. Переключение вывода.](#_Toc515770634)

[1.5.2. Переключение ввода.](#_Toc515770635)

[1.5.3. Комбинированное переключение.](#_Toc515770636)

[1.5.4. Конвейерная пересылка](#_Toc515770637)

[1.6. Текстовые и бинарные (двоичные) файлы](#_Toc515770638)

[1.7. Потоковый ввод-вывод.](#_Toc515770639)

[1.8. Соединение и отсоединение потока от файла.](#_Toc515770640)

[1.8.1. Функция fopen](#_Toc515770641)

[1.8.2. Функция fclose](#_Toc515770642)

[1.8.3. Функция freopen](#_Toc515770643)

[1.9. Работа с индикаторами ошибки, позиции и конца файла (ferror, clearerr, feof, rewind, fseek, fsetpos, ftell, fgetpos).](#_Toc515770644)

[1.10. Блочный ввод-вывод (fwrite, fread).](#_Toc515770645)

[1.11. Символьный ввод-вывод.](#_Toc515770646)

[1.11.1. Функции fputc, putc, fgetc, getc, ungetc](#_Toc515770647)

[1.11.2. Функции fputs и fgets](#_Toc515770648)

[1.11.3. Функции fprintf, fscanf](#_Toc515770649)

[1.12. Работа с буферами (setvbuf, setbuf, fflush).](#_Toc515770650)

[1.13. Стандартные потоки (putchar, getchar, puts, gets, perror).](#_Toc515770651)

[1.14. Служебные функции для работы с файлами (remove, rename, tmpfile, tmpnam).](#_Toc515770652)

[1.15. Низкоуровневый ввод и вывод в С.](#_Toc515770653)

[1.16. Форматированный вывод.](#_Toc515770654)

[1.16.1. Использование функций printf() и fprintf().](#_Toc515770655)

[1.17. Примеры](#_Toc515770656)

[1.17.1. Содержимое файла](#_Toc515770657)

[1.17.2. Чтение/запись из/в файл](#_Toc515770658)

[1.17.3. Назначение буфера потоку stderr](#_Toc515770659)

[1.17.4. Демо для функции setvbuf()](#_Toc515770660)

[1.17.5. Структура и файл](#_Toc515770661)

[1.17.6. Сжатие файла](#_Toc515770662)

[1.17.7. Функции fgets и fputs](#_Toc515770663)

[1.17.8. Функции fread и fwrite, fseek, ftell, rewind](#_Toc515770664)

[1.17.9. Использование fseek](#_Toc515770665)

1. Файлы в C.

http://www.itmathrepetitor.ru/s-lekciya-6-fajjly-i-potoki-struktury-i-pe/

## Файлы и потоки.

Интуитивное определение файла звучит примерно так. **Файл** -- именованная область на жестком диске. На самом деле с точки зрения ОС UNIX это совсем не так. В ОС UNIX файл -- очень удобная абстракция. С точки зрения UNIX файлом называется "что-нибудь", из чего можно считывать информацию или во что можно записывать информацию. Файлы это:

* Файлы в обычном смысле: файлы, которые хранятся на жестком диске (можно считывать из них и запиcывать в них информацию);
* Экран монитора: файл, в который можно выводить информацию (отобразится на экране монитора);
* Клавиатура: файл, из которого можно считывать информацию;
* Принтер: файл, в который можно выводить информацию (печать текста);
* Модем: файл, из которого можно считывать информацию и в который можно записывать информацию (обмен информации по сети);

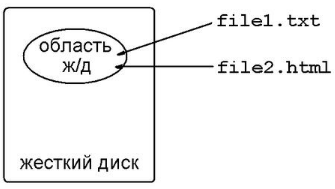
**Файл** -- это всё, что предназначено для ввода или вывода информации.

С этой точки зрения файлы бывают разными: принтер может только выводить информацию, а клавиатура -- только вводить. У такого рода файлов есть много особенностей. У файла на жестком диске есть понятие конца файла. Мы можем его считывать до тех пор, пока он не кончится. Тогда как у клавиатуры нет конца.

Неправильно думать, что между сущностями "файл" и "название файла" есть взаимно однозначное соответствие.

Можно привести аналогию из жизни: если представить, что файл -- это банка с некоторым содержимым, то название файла -- это этикетка на этой банке. Логично предположить, что у банки может быть несколько этикеток.

С точки зрения UNIX:



Правильно говорить, что у названия есть файл. И наоборот: неправильно говорить, что у файла есть название. Никакого эффективного способа узнать имя файла не существует (но можно перебрать все файлы файловой системы).

**Чтение и запись: printf и scanf**

Всем хорошо известная функция printf:

printf("Hello!");//печать текста на экран;

printf("N = %d", N);//форматированный вывод на экран: вывести число N в десятичной записи;

printf("N = %x", N);//форматированный вывод на экран: вывести число N в шестнадцатеричной записи;

Аналогично парная функция scanf:

scanf("%d", &N);//считывание с клавиатуры значения переменной N в десятичной записи;

char **\*ptr =** newchar[10];

scanf("%s", ptr);//считывание с клавиатуры строки в массив \*ptr

Тут могут возникать различные проблемы.

1. **Проблема безопасности:**

char **\*ptr =** newchar[10];

scanf("%s", ptr);

Тут налицо потенциальная проблема переполнения буфера (в данном примере в буфере всего 10 байт).

Никогда не следует пользоваться scanf-ом для чтения строк.

**scanf + "%s" -- запрещенная комбинация!**

1. **Форматная строка не компилируется:** она будет разбираться в момент исполнения программы. Это обозначает проблему быстродействия. scanf -- не предназначен для чтения большого количества информации. Аналогично printf -- тоже сравнительно медленный (однако существенно быстрее, чем scanf).
2. **Проблема безопасной работы со стеком**:

printf("%d %d", N);

Проблема состоит в том, что форматная строка "%d %d" будет проанализирована в момент исполнения. В данном случае произойдет ошибка при работе со стеком: во время исполнения будет взят лишний int.

Перечисленные недостатки означают, что использование функций printf и scanf небезопасно и малоэффективно. Существенным плюсом этих функций является возможность простого форматированного ввода и вывода.

Для нас файл является частью памяти, обычно на диске, со своим именем. Мы считаем, например, stdio.h именем файла, содержащего некоторую полезную информацию. Для операционной системы файл более сложен, но это системные проблемы, а не наши. Однако мы должны знать, что означает файл для программы на языке Си. В предлагаемых для обсуждения функциях, работающих с файлами, язык Си «рассматривает» файл как структуру. Действительно, файл stdio.h содержит определение структуры файла.

В stdio.h (на моем компьютере) фрагмент описания файла выглядит следующим образом:

#ifndef \_FILE\_DEFINED

struct \_iobuf {

char \*\_ptr; /\* текущий указатель буфера \*/

int \_cnt; /\* текущий счетчик байтов \*/

char \*\_base; /\* базовый адрес буфера ввода-вывода\*/

int \_flag; /\* управляющий признак \*/

int \_file; /\* номер файла \*/

int \_charbuf;

int \_bufsiz;

char \*\_tmpfname;

};

typedef struct \_iobuf FILE;

#define \_FILE\_DEFINED

#endif

А вот типичный пример, взятый из IBM-версии компилятора Lattice С:

struct \_iobuf

{

char \*\_ptr; /\* текущий указатель буфера \*/

int \_cnt; /\* текущий счетчик байтов \*/

char \*\_ base; /\* базовый адрес буфера ввода-вывода\*/

char \_ flag; /\* управляющий признак \*/

char \_ file; /\* номер файла \*/

};

#define FILE struct \_iobuf /\* краткая запись \*/

Здесь мы не собираемся разбираться детально в этом определении. Главное состоит в том, что файл является структурой, и что краткое наименование шаблона файла — FILE. (Многие системы используют директиву typedef для установления этого соответствия.) Таким образом, программа, имеющая дело с файлами, будет использовать тип структуры FILE, чтобы делать так.

Имея это в виду, мы сможем лучше понять операции над файлами.

**Файлом** называют последовательность байтов, хранящихся на внешнем носителе информации. Под доступом к файлу понимают запись и чтение данных из файла. **Потоком** называется логический интерфейс (программа), который обеспечивает доступ к файлу. Прежде чем использовать поток для доступа к файлу, его необходимо соединить с этим файлом, то есть обеспечить поток информацией о файле. Эта информация хранится в структуре типа FILE. Потому считается, что поток имеет тип FILE\*, то есть является указателем на файл. Когда поток соединяют с файлом, то говорят, что файл **открывают**. Когда поток отсоединяют от файла, то говорят, что файл **закрывают**.

Каждый поток может работать в двух режимах: **текстовом** и **бинарном**. Режим работы потока задается при его соединении с файлом.

В **текстовом** режиме поток записывает и читает из файла текстовые строки, которые заканчиваются символом ‘\n’ и могут содержать символ ‘\t’. По стандарту поток должен обеспечивать обработку строк длиной не менее 254 символа, включая символ ‘\n’. Стандартом допускается, что при чтении и записи данных текстовым потоком может происходить их преобразование.

В **бинарном** режим поток записывает и читает данные из файла в том виде, в котором они хранятся в оперативной памяти.

Далее будут описаны стандартные функции для работы с файлами, используемые в языке С. Прототипы этих функций находятся в заголовочном файле stdio.h.

Термин поток происходит из представления процесса ввода-вывода в файл в виде последовательности или потока байтов. Над потоком можно выполнять следующие операции:

* Считывание блока данных из потока в оперативную память;
* Запись блока данных из оперативной памяти в поток;
* Обновление блока данных в потоке;
* Считывание записи из потока;
* Занесение записи в поток.

**Все потоковые функции ввода-вывода обеспечивают буферизированный, форматированный или не форматированный ввод и вывод.**

Когда начинается выполнение программы, автоматически открываются следующие потоки:

* stdin - стандартное устройство ввода;
* stdout - стандартное устройство вывода;
* stderr - стандартное устройство сообщений об ошибках;
* stdprn - стандартное устройство печати;
* stdaux - стандартное вспомогательное устройство.

Все они называются стандартными (или предопределенными) потоками ввода-вывода. По умолчанию **стандартным устройством ввода, вывода и сообщений об ошибках** является пользовательский терминал. Поток **стандартного устройства печати** относится к принтеру, а поток **стандартного вспомогательного устройства** - к вспомогательному порту компьютера. По умолчанию при открытии все стандартные потоки, за исключением потоков stderr и stdaux, буферизуются.

## Связь с файлами.

Часто нам бывает нужна программа получения информации от файла или размещения результатов в файле. Один способ организации связи программы с файлом заключается в использовании **операций переключения** < и >. Этот метод прост, но ограничен. Например, предположим, вы хотите написать диалоговую программу, которая спрашивает у вас названия книг (звучит фамильярно?), и вы намерены сохранить весь список в файле. Если вы используете переключение как, например, в

books > bklist

то ваши диалоговые приглашения также будут переключены на bklist. И тогда не только нежелательная чепуха запишется в bklist, но и пользователь будет избавлен от вопросов, на которые он, как предполагалось, должен отвечать.

К счастью, язык Си предоставляет и более мощные методы связи с файлами. Один подход заключается в использовании функции fopen(), которая открывает файл, затем применяются специальные функции ввода-вывода для чтения файла или записи в этот файл и далее используется функция fclose() для закрытия файла. Однако прежде чем исследовать эти функции, нам нужно хотя бы кратко познакомиться с сущностью файла.

**Чтение и запись файлов: FILE\*, fopen, fprintf, fscanf**

Есть несколько способов работы с файлами c использованием языков C и C++.

Самый распространенный связан со структурой FILE (это не класс, потому что сущность языка C). Эта структура определена в заголовочном файле стандартной библиотеки <stdio.h>. **Размер этой структуры и ее поля зависят от ОС и от версии компилятора.** Поэтому никто не пользуется структурой FILE. Обычно пользуются указателем на эту структуру: FILE\*. Например:

FILE \*f = fopen("file1.txt", "r");

fopen -- функция из стандартной библиотеки. Первый параметр -- имя файла (в текущем каталоге). Второй параметр задает режим открытия файла; в данном случае "r" означает, что файл будет открыт только для чтения. Эта функция возвращает ненулевой указатель, если открытие прошло успешно; и возвращает NULL, если произошла ошибка. Ошибка может возникать в следующих ситуациях:

* не существует файла;
* у программы недостаточно прав доступа для работы с файлом;

Для дальнейшей корректной работы следует писать примерно такой код:

if **(f == NULL) {**

// файл не удалось открыть

}

else **{**

// Работа с файлом

}

Допустим, что нам удалось открыть файл, т.е. f != NULL. Тогда для того, чтобы считывать файл, можно использовать функцию:

fscanf(f, "%s", ptr);

Эта функция работает аналогично функции scanf. Поэтому использовать эту функцию небезопасно! Все проблемы, перечисленные для scanf-а, имеют место и при работе с fscanf-ом.

Если мы хотим записать в файл что-то, то мы должны сначала открыть его на запись:

FILE \*f = fopen("file2.html", "w");

Тут "w" означает, что мы открываем файл на запись (от write). Если файл не существовал, то он создастся и откроется на запись, а если он существовал, то он сначала будет уничтожен, а затем создан заново, и потом файл будет открыт на запись.

Еще один способ открыть файл -- это открыть его на дозапись. Это можно сделать с помощью параметра "a" (от append). Если файл не существовал, то он создастся и откроется на запись, а если он существовал, то он откроется на запись, и запись будет производится в конец файла.

Затем можно использовать функцию fprintf(f, ...)

**Замечание**

Подробности и примеры использования fopen в вопросе «1.8. Соединение и отсоединение потока от файла»

## Закрытие потоков.

По завершении работы с потоком он должен быть закрыт. Это осуществляется с помощью функции fclose(), которая имеет следующий прототип:

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

Все буферы, связанные с потоком, освобождаются перед закрытием потока. В случае успеха fclose() возвращает 0; в случае ошибки - EOF. Если ваша программа не закрывает поток с помощью явного вызова fclose (), то он закрывается автоматически по ее завершению.

Две функции **fclose()** и **fcloseall()** используются для закрытия потока или всех потоков, соответственно. Функция fclose() закрывает отдельный файл, а функция fcloseall() закрывает все открытые потоки, за исключением **stdin, stdout, stderr, stdprn и stdaux**. Однако, если ваша программа не закрывает поток явно, то он закрывается автоматически по ее завершению. Поскольку количество потоков, открытых одновременно, ограничено, то желательно закрывать поток по окончанию работы с ним.

Наш пример показывает, как закрывать файл:

fclose(in);

Просто используйте функцию fclose(). Заметим, что аргументом ее является in, указатель на файл, а не test, имя файла.

Для программы, более серьезной, чем эта, следовало бы посмотреть, успешно ли закрыт файл. Функция fclose() возвращает значение 0, если файл закрыт успешно, и — 1 в противном случае.

**Зачем нужно закрывать файлы**

* Зададимся вопросом: "Что надо сделать после того, как мы поработали с файлом?"

Формальный ответ: "Закрыть файл." Это можно сделать с помощью функции:

fclose(f);

Но зачем это делать?

Ввиду механического устройства жесткого диска, данные в файл попадают не сразу. Сначала данные записываются в так называемый буфер (область оперативной памяти), и когда он переполнится, то данные из буфера будут записаны в файл. Такая схема придумана для ускорения работы с файлами. На самом деле, буфер -- это поле структуры FILE: указатель на массив char-ов.

Если мы напишем fprintf(...), то запись произведется в буфер. И только тогда, когда буфер будет заполнен до конца, он будет сразу весь записан на жесткий диск. По этой причине, если мы не закроем файл функцией fclose(f), то последние данные из буфера не запишутся в файл. Отсутствие этой команды может привести к потере данных в файле, который был открыт для записи (дозаписи).

* А зачем закрывать файлы, открытые только на чтение?

Если не закрывать файлы (которые открыты даже для чтения), то это может привести к ограничению доступа к файлу для других программ. Какие именно ограничения наложатся - это зависит от ОС. Но в ОС Windows если файл открыт на чтение и не закрывается, то из другой программы его нельзя удалить.

* В любой ОС есть ограничение на количество одновременно открытых файлов. И это еще одна причина для закрытия файлов.

**Замечание**

Подробности и примеры использования fclose в вопросе «1.8. Соединение и отсоединение потока от файла»

## Обработка ошибок в C и C++

Ошибки, увы, неизбежны, поэтому их обработка занимает очень важное место в программировании. И если алгоритмические ошибки можно выявить и исправить во время написания и тестирования программы, то ошибок во время выполнения избежать нельзя в принципе. Рассмотрим функции стандартной библиотеки ([C Standard Library](https://en.wikipedia.org/wiki/C_standard_library)) и [POSIX](https://en.wikipedia.org/wiki/POSIX), используемые в обработке ошибок.

### Переменная errno и коды ошибок

[**<errno.h>**](https://en.wikipedia.org/wiki/Errno.h) **(С) и <cerrno> (C++)**

**errno** – переменная, хранящая целочисленный код последней ошибки. В каждом потоке существует своя локальная версия **errno**, чем и обусловливается её безопасность в многопоточной среде. Обычно **errno** реализуется в виде макроса, разворачивающегося в вызов функции, возвращающей указатель на целочисленный буфер. **При запуске программы значение errno равно нулю**.

Все коды ошибок имеют положительные значения, и могут использоваться в директивах препроцессора **#if**. В целях удобства и переносимости заголовочный файл **<errno.h>** определяет макросы, соответствующие кодам ошибок.

Стандарт ISO C определяет следующие коды:

**EDOM** – (Error domain) ошибка области определения.

**EILSEQ** – (Error invalid sequence) ошибочная последовательность байтов.

**ERANGE** – (Error range) результат слишком велик.

Прочие коды ошибок (несколько десятков) и их описания определены в стандарте POSIX. Кроме того, в спецификациях стандартных функций обычно указываются используемые ими коды ошибок и их описания.

Если вызов функции завершился ошибкой, то она устанавливает переменную **errno** в ненулевое значение. Если же вызов прошёл успешно, функция обычно не проверяет и не меняет переменную **errno**. Поэтому перед вызовом функции её нужно установить в **0**.

Макрос **errno** возвращает последний номер ошибки. Этот макрос заменяется на модифицируемое значение типа int, поэтому **errno** может изменять ход работы программы.

В C++, ошибка всегда объявляется как макрос.

#include<iostream>// для оператора cout

#include<cerrno>// для макроса errno

#include<math.h>// для функции sqrt

usingnamespace **std;**

int **main()**

{

**setlocale(LC\_ALL,** "Rus");

double **x = -1, answer;**

**answer = sqrt(x);**

**cout <<** "EDOM = " **<< EDOM << endl**

**<<** "nerrno = " **<< errno << endl;**

if **(errno == EDOM)** // если значения макросов errno и EDOM равны

**cout <<** "Ошибка области допустимых значений\n";

**system(**"pause");

return **0;**

}



#include<errno.h>

#include<stdio.h>

#include<stdlib.h>

#include<iostream>// для оператора cout

void **main(**void)

{

**setlocale(LC\_ALL,** "Rus");

**FILE \*fp;**

**errno = 0;**

**fp = fopen(**"Nofile", "r");

if **( errno != 0 )**

**{**

**perror(**"Error occurred while opening file.\n");

**system(**"pause");

**exit(1);**

**}**

}



Здесь представлена наиболее полная информация для работы с файлами

https://msdn.microsoft.com/en-us/library/z5hh6ee9.aspx

// crt\_fopen\_s.c

// This program opens two files. It uses

// fclose to close the first file and

// \_fcloseall to close all remaining files.

#include<stdio.h>

#include<iostream>

FILE \*stream, \*stream2;

int **main(** void **)**

{

**errno\_t err;**

// Open for read (will fail if file "crt\_fopen\_s.c" does not exist)

**err = fopen\_s( &stream,** "crt\_fopen\_s.c", "r" **);**

if( err == 0 )

**{**

**printf(** "The file 'crt\_fopen\_s.c' was opened\n" **);**

**}**

else

**{**

**printf(** "The file 'crt\_fopen\_s.c' was not opened\n" **);**

**}**

// Open for write

**err = fopen\_s( &stream2,** "data2", "w+" **);**

if( err == 0 )

**{**

**printf(** "The file 'data2' was opened\n" **);**

**}**

else

**{**

**printf(** "The file 'data2' was not opened\n" **);**

**}**

// Close stream if it is not NULL

if( stream )

**{**

**err = fclose( stream );**

if **( err == 0 )**

**{**

**printf(** "The file 'crt\_fopen\_s.c' was closed\n" **);**

**}**

else

**{**

**printf(** "The file 'crt\_fopen\_s.c' was not closed\n" **);**

**}**

**}**

// All other files are closed:

int **numclosed = \_fcloseall( );**

**printf(** "Number of files closed by \_fcloseall: %u\n", numclosed );

**system(**"pause");

}



## Переключение и работа с файлами.

Понятие ввода-вывода включает в себя функции, данные и устройства. Рассмотрим, например, нашу программу ввод-вывод4. В ней используется функция getchar(), осуществляющая ввод, причем устройство ввода — клавиатура (в соответствии с нашим предположением), а входные данные — отдельные символы. Нам бы хотелось сохранить функции ввода и тип данных, но изменить источник их поступления в программу. Зададимся вопросом: откуда программа узнает, где искать входные данные?

По умолчанию Си-программа рассматривает «стандартный ввод» как источник поступления данных. «Стандартным вводом» называется устройство, принятое в качестве обычного средства ввода данных в машину. Это может быть устройство чтения данных с магнитной ленты или перфокарт, телетайп или (как мы продолжаем считать) терминал. Современная машина — это послушный инструмент, и мы можем воздействовать на нее так, чтобы она вводила данные из любого источника. В частности, мы можем сообщить программе, что источник входных данных — файл, а не клавиатура.

Существуют два способа написания программ, работающих с файлами. Первый способ заключается в явном использовании специальных функций, которые открывают и закрывают файлы, организуют чтение и запись данных и т. п.; мы не хотим пока касаться этого вопроса. Второй способ состоит в том, чтобы использовать программу, спроектированную первоначально в предположении, что данные в нее вводятся с клавиатуры и выводятся на экран, но переключить ввод и вывод на другие информационные каналы, например, из файла в файл. Этот способ в некоторых отношениях обладает меньшими возможностями, чем первый, но зато гораздо проще в использовании. Мы изучим понятие переключения в данном разделе.

Операция переключения — это средство ОС UNIX, а не самого языка Си. Но она оказалась настолько полезной, что при переносе компилятора с языка Си на другие вычислительные системы часто вместе с ним переносится в какой-то форме и эта операция. Более того, многие из вновь созданных операционных систем, таких, как MS-DOS 2, включают в себя данное средство. Поэтому, даже если вы не работаете в среде ОС UNIX существует большая вероятность того, что вы в той или иной форме сможете воспользоваться операцией переключения. Мы обсудим сначала возможности этой операции в ОС UNIX, а затем и в других системах.

### Переключение вывода.

Предположим, вы осуществили компиляцию программы ввод-вывод4 и поместили выполняемый объектный код в файл с именем getput4. Затем, чтобы запустить данную программу, вы вводите с терминала только имя файла

getput4

и программа выполняется так, как было описано выше, т. е. получает в качестве входных данных символы, вводимые с клавиатуры. Теперь предположим, что вы хотите посмотреть, как наша программа работает с «текстовым файлом» с именем words. (Текстовой файл — это файл, содержащий некоторый текст, т. е. данные в виде символов. Это может быть, например, очерк или программа на языке Си. Файл, содержащий команды на машинном языке, например файл, полученный в результате компиляции данной программы, не является текстовым. Поскольку наша программа занимается обработкой символов, она должна использоваться вместе с текстовыми файлами.) Все, что для этого требуется — ввести вместо команды, указанной выше, следующую:

getput4 < words

Символ < служит обозначением операции переключения, используемой в ОС UNIX. Выполнение указанной операции приводит к тому, что содержимое файла words будет направлено в файл с именем getput4. Сама программа ввод-вывод4 не знает (и не должна знать), что входные данные поступают из некоторого файла, а не с терминала; на ее вход просто поступает поток символов, она читает их и последовательно, по одному выводит на печать до тех пор, пока не встретит признак EOF. В операционной системе UNIX файлы и устройства ввода-вывода в логическом смысле представляют собой одно и то же, поэтому теперь файл для данной программы является устройством ввода-вывода. Если вы попробуете ввести команду

getput4 < words

то в результате на экране могут появиться, например, следующие строки:

В одном мгновеньи видеть вечность,

Огромный мир - в зерне песка,

В единой горсти - бесконечность,

И небо - в чашечке цветка

Но мы, конечно, не можем гарантировать, что в файле, который выберете вы, тоже окажется четверостишие Вильяма Блейка.

### Переключение ввода.

Теперь предположим (если вы еще не устали и в состоянии что-нибудь предположить), вам хочется, чтобы слова, вводимые с клавиатуры, попадали в файл с именем mywords. Для этого вы должны ввести команду

getput4 > mywords

и начать ввод символов. Символ > служит обозначением еще одной операции переключения, используемой в ОС UNIX. Ее выполнение приводит к тому, что создается новый файл с именем mywords, а затем результат работы программы ввод-вывод4, представляющий собой копию вводимых символов, направляется в данный файл. Если файл с именем mywords уже существует, он обычно уничтожается, и вместо него создается новый. (В некоторых реализациях ОС UNIX, однако, вам предоставляется возможность защитить существующие файлы.) На экране в данном случае появятся лишь вводимые вами символы, их же копии будут направлены в указанный файл. Чтобы закончить работу программы, введите признак EOF (в системе UNIX это обычно символ [CTRL/D]). Попробуйте воспользоваться описанной здесь операцией. Если вам ничего другого не придет в голову, просто воспроизведите на своей машине пример, приведенный ниже. Знак приглашения, выводимый на экран интерпретатором команд SHELL, обозначается здесь символом %. Не забывайте оканчивать каждую введенную строку символом [возврат], чтобы содержимое буфера пересылалось в программу.

% getput4 > mywords

У вас не должно быть трудностей с запоминанием того, какая операция переключения для чего предназначена. Необходимо помнить только, что знак каждой операции указывает на направление информационного потока. Вы можете по ассоциации представлять себе этот знак в виде воронки. [CTRL/D]

%

После того как введен символ [CTRL/D], программа заканчивает свою работу и возвращает управление операционной системе UNIX, на что указывает повторное появление знака приглашения. Как убедиться в том, что наша программа вообще работала? В ОС UNIX существует команда Is, которая выводит на экран имена файлов; обращение к ней должно продемонстрировать вам, что файл с именем mywords теперь существует. Чтобы проверить его содержимое, вы можете воспользоваться командой cat или запустить заново программу ввод-вывод4, направляя в нее на этот раз содержимое входного файла.

% getput4 < mywords

У вас не должно быть трудностей с запоминанием того, какая операция переключения для чего предназначена. Необходимо помнить только, что знак каждой операции указывает на направление информационного потока. Вы можете по ассоциации представлять себе этот знак в виде воронки.

### Комбинированное переключение.

Предположим теперь, что вы хотите создать копию файла mywords и назвать ее savewords. Введите для этого команду

getput4 < mywords >savewords

и требуемое задание будет выполнено. Команда

getput4 >savewords < mywords

приведет к такому же результату, поскольку порядок указания операций переключения не имеет значения.

Нельзя использовать в одной команде один и тот же файл и для ввода и для вывода одновременно.

getput4 <mywords >mywords /\*НЕПРАВИЛЬНО\*/

Причина этого заключается в том, что указание операции >mywords приводит к стиранию исходного файла перед его использованием в качестве входного.

Теперь, мы думаем, настало время суммировать правила, касающиеся использования двух операций переключения < и >.

1. Операция переключения связывает выполняемую программу (в том числе и стандартные команды ОС UNIX) с некоторым файлом. Она не может использоваться для связи одного файла с другим или одной программы с другой.
2. Имя выполняемой программы должно стоять слева от знака операции, а имя файла — справа от него.
3. При использовании этих операций ввод не может осуществляться более чем из одного файла, а вывод — более чем в один файл.
4. Обычно между именем и операцией пробелы не обязательны кроме тех редких случаев, когда используются некоторые символы специального назначения в интерпретаторе команд UNIX. Мы могли бы писать, например, так: getput4< words, или, что более предпочтительно, getput4 < words.

Мы уже привели выше несколько примеров правильного использования операций переключения. Ниже дается несколько ошибочных примеров (addup и count — выполняемые программы, a fish и stars — текстовые файлы).

fish > stars //Нарушение правила 1

addup < count //Нарушение правила 1

stars > count //Нарушение правила 2

addup < fish < stars //Нарушение правила 3

count > stars > fish //Нарушение правила 3

В ОС UNIX **применяются также операция >>, позволяющая добавлять данные в конец существующего файла, и операция «канал» (]), связывающая файл вывода одной программы с вводом другой**. Для получения более детальной информации обо всех этих операциях вам необходимо обратиться к руководству по ОС UNIX (по аналогии с этим нам приходит в голову название «ОС UNIX: руководство для начинающих»).

Рассмотрим еще один пример: напишем очень простую программу, шифрующую сообщения. C этой целью мы немного изменим программу ввод-вывод4 и получим:

// простой шифр

// заменяет каждый символ текста

// следующим по порядку из кода ASCII

// X24.C

#include <stdio.h>

void main(void)

{

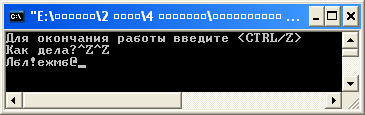
char ch;

printf("Для окончания работы введите <CTRL/Z>\n");

while ((ch=getchar()) != EOF)

putchar(ch+1);

}



Функция putchar() переводит целое «ch + 1» в соответствующий символ.

Выполните теперь компиляцию программы и поместите выполняемый объектный код в файл с именем simplecode. Затем занесите приведенные ниже строки (Перевод: знание орфографии — залог четкости письма) в файл с именем original. (Для этого можно воспользоваться системным текстовым редактором или, как было показано ранее, программой ввод-вывод4.)

Good spelling is an aid

to clear writing.

Теперь введите команду

simplecode < original

Результат должен выглядеть приблизительно так:

!!!!!Hppe!tqfmmjoh!jtbo!bje^Kup!dmfbs!xsjujohl^k

Буква G заменится на Н, о на р и т. д. Вас может удивить следующее: во-первых, что пробелы превратились в восклицательные знаки. Это служит напоминанием, что пробел — такой же символ, как и все остальные. Во-вторых, две строки слились в одну. Почему? Потому что в тексте, содержащемся в файле original, в конце первой строки находится символ «новая строка», служащий указанием компьютеру начать вывод следующего слова с новой строки. Но этот символ также был изменен. В нашей системе он был заменен символом ^К, являющимся аналогом специального символа [CTRL/K], и поэтому последующий вывод на печать был продолжен на прежней строке. Если мы хотим иметь программу шифровки сообщений, сохраняющую первоначальную структуру текста (по строкам), нам необходимо средство, позволяющее изменять все символы, кроме символа «новая строка». В дальнейшем мы узнаем, как это сделать.

### Конвейерная пересылка

Можно также осуществить соединение выходного потока одной программы с входным потоком другой. Это называется конвейерной пересылкой. Если имеются две выполнимые программы example1 и example2, то конвейерная пересылка между ними организуется с помощью символа вертикальной черты ‘|’. Следующая строка организует конвейерную пересылку между example1 и example2:

example1 | example2

Организацию конвейерной пересылки обеспечит операционная система.

## Текстовые и бинарные (двоичные) файлы

Язык С++ унаследовал от языка C библиотеку стандартных функций ввода-вывода. Функции ввода-вывода объявлены в заголовочном файле <stdio.h>. Операции ввода-вывода осуществляются с файлами. Файл может быть **текстовым** или **бинарным (двоичным)**. **Различие между ними** заключается в том, что в **текстовом** файле последовательности символов разбиты на строки. Признаком конца строки является пара символов CR (возврат каретки) и LF (перевод строки) или, что то же самое - '\г' + '\n'. При вводе информации из текстового файла эта пара символов заменяется символом CR, при выводе, наоборот, - символ CR заменяется парой символов CR и LF. **Бинарный (или двоичный)** файл - это просто последовательность символов. Обычно двоичные файлы используются в том случае, если они являются источником информации, не предполагающей ее непосредственного представления человеку. При вводе и выводе информации в бинарные файлы никакого преобразования символов не производится.

**Текстовые и бинарные файлы; что меняет опция t/b**

Рассмотрим строку:

fopen(f, "file1.txt", "w");

Почему второй параметр "w" является строкой, а не символом?

На самом деле бывает много способов прочитать/записать файл. Например:

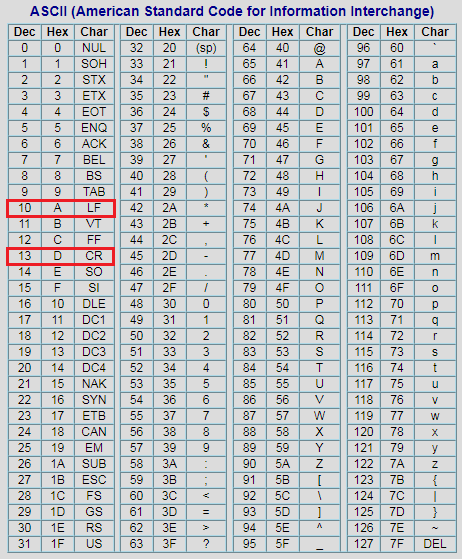
fopen("file1.txt", "wt") -- откроет файл как текстовый файл;

fopen("file1.txt", "wb") -- откроет файл как бинарный файл.

Но в чем отличие? **Разница заключается лишь в том, что символы переноса строк запишутся по разному.** Рассмотрим пример в UNIX и Windows.

Исходная строка кода выглядит так:

fprintf("Hello\n");



1. Откроем в Windows файл на запись с параметром "wb" (как бинарный файл). Это означает, что в него запишется в точности то, что мы передали в функции fprintf. Тогда в файл запишутся ровно 6 байт:   
     
   Hello\10
2. А теперь мы откроем в Windows файл на запись с параметром "wt" (как текстовый файл). Тогда в файл запишутся ровно 7 байт:  
     
   Hello\10\13  
     
   Тут \10\13 означает символы перевода строки в ОС Windows.
3. Откроем в UNIX файл на запись с параметром "wt" или "wb". Тогда в файл запишутся ровно 6 байт:  
     
   Hello\10  
     
   Тут \10 означает символы перевода строки в ОС UNIX.

В ОС UNIX разницы все-таки нет.

**Различие между "wt" и "wb" объясняется тем, что в разных операционных системах символы перевода строки разные. При чтении файла, т.е. при открытии файла с параметрами "rt" или "rb", проблема следующая. Если мы поставим параметр "rb", то при чтении файла символ \10 будет восприниматься как перевод строки. А если поставим параметр "rt", то при чтении файла пара символов \10\13 будет восприниматься как символ перевода строки.**

http://rsdn.org/forum/cpp/2277645.hot

**Двоичные файлы**

Двоичные и текстовые файлы — это, как говорят в Одессе, «две большие разницы». Обмен данными между программой и двоичным потоком выполняется без всякого преобразования, поэтому работает быстрее. **Двоичный** **ввод** выполняется функцией **fread()**, имеющей следующий прототип:

size\_t fread(void **\*buffer, size\_t size, size\_t n, FILE \*stream);**

Тип **size\_t** обычно определен как **unsigned int**. **Первый** параметр часто определяет массив (или указатель на динамический массив), в который будет прочитана информация; **третий** параметр задает размер одного элемента данных в байтах, а **второй** — количество читаемых элементов. **Четвертый** параметр определяет **двоичный файл**, из которого информация вводится. Общее количество считанных байтов равно **size\*n**. Однако **возвращает функция количество корректно прочитанных элементов, а не байтов**.

**Вывод в двоичный файл** выполняется функцией **fwrite()**, которая имеет совершенно аналогичный прототип:

size\_t fwrite(constvoid **\*buffer, size\_t size, size\_t n, FILE \*stream);**

Функция записывает **n** элементов размера **size** в двоичный файл **stream** из буфера, указатель на который задается в качестве **первого** аргумента. Общее количество выводимых байтов равно **size\*n**. Однако **возвращает функция количество корректно записанных элементов, а не байтов**.

В качестве элементов могут использоваться любые переменные любых типов, в том числе и динамические. Даже массив может быть одним-единственным элементом! Рассмотрим несколько простых примеров, аналогичных примерам для текстовых файлов. Создадим на диске C: каталог **BinFiles** и все двоичные файлы будем размещать в нем. Переделаем пример создания файла

//Создание и чтение двоичных файлов

#include<cstdio>

#include<cstdlib>

#include<ctime>

int **main()**

{ int **m[10]={0};**

**srand((**unsigned)time(NULL)); // инициализация датчика случайных чисел

**FILE \*stream;**

/\* открываем двоичный файл для записи \*/

//if((stream = fopen("c:/binfiles/number1.bin", "wb" )) == NULL)

if((stream = fopen("number1.bin", "wb" **)) == NULL)**

return **1;** // ошибка при открытии

/\* заполняем массив m числами \*/

for(int **i = 0; i < 10; i++)**

**m[i] = rand()%10;** // случайные числа от 0 до 9

// заполняем файл number1.bin элементами-числами

for(int **i = 0; i < 10; i++)**

**fwrite(&m[i], 1,** sizeof(int), stream);

**fclose(stream);** // закрываем файл

/\* открываем другой файл для записи \*/

//if((stream = fopen("c:/binfiles/number2.bin", "wb" )) == NULL)

if((stream = fopen("number2.bin", "wb" **)) == NULL)**

return **1;** // ошибка при открытии

// заполняем файл number2.bin элементом-массивом

**fwrite(m, 1,** sizeof(m), stream); // массив - один элемент

**fclose(stream);** // закрываем файл

// вывод второго двоичного файла на экран

// открываем файл для чтения

//if((stream = fopen("c:/binfiles/number2.bin", "rb" )) == NULL)

if((stream = fopen("number2.bin", "rb" **)) == NULL)**

return **1;** // ошибка при открытии

int **a = 0;** // сюда вводим

// читаем второй файл поэлементно

// правильный цикл

**printf(**"number2.bin\n");

**fread(&a, 1,** sizeof(int), stream); // предварительное чтение

while(!feof(stream)) // пока не конец файла

**{ printf(**"%d\n", a);

**fread(&a, 1,** sizeof(int), stream);

**}**

**fclose(stream);**

// открываем первый файл в режиме чтения

**printf(**"number1.bin\n");

//if((stream = fopen("c:/binfiles/number1.bin", "rb" )) == NULL)

if((stream = fopen("number1.bin", "rb" **)) == NULL)**

return **1;**

int **t[10] = {0};** // массив для чтения

// читаем первый файл как массив

**fread(t, 1,** sizeof(t), stream);

for(int **i = 0; i < 10; i++)** // выводим поэлементно

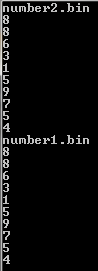
**printf(**"%d\n",t[i]);

**fclose(stream);**

char **ch = getchar();**

return **0;**

}



## Потоковый ввод-вывод.

Многие широко используемые языки высокого уровня имеют механизмы ввода/вывода, позволяющие создавать нетривиальные алгоритмы для получения и отображения сложных структур данных. Это не относится к языку С, в котором имеется весьма развитая библиотека функций ввода/вывода, хотя, исторически, ввод/вывод никогда не являлся частью самого языка С. Если вы пользуетесь только простыми операторами ввода/вывода, подобными операторам Паскаля readln и writeln, то такая ситуация может вас удивить. Здесь описываются более 20 различных способов организации ввода/вывода в С.

Библиотека стандартных функций ввода/вывода С позволяет считывать данные и записывать их в файлы и устройства. Однако в самом языке С отсутствуют какие-либо предопределенные файловые структуры. В С все данные обрабатываются как последовательность байт. Имеется **три основных типа функций ввода/вывода**:

* потоковые,
* работающие с консолью и портами,
* низкоуровневые.

В **потоковых** функциях ввода/вывода файлы или объекты данных рассматриваются как поток отдельных символов. Выбирая соответствующую потоковую функцию, вы можете обрабатывать данные любого необходимого размера или формата, начиная от отдельных символов и заканчивая большими, сложными структурами данных.

На техническом уровне, когда программа открывает файл для ввода/вывода при помощи потоковых функций, открытый файл связывается с некоторой структурой типа FILE (предопределенной в stdio.h), содержащей базовую информацию об этом файле. После открытия потока возвращается указатель на файловую структуру. Указатель файла, иногда называемый указателем потока или потоком, используется для ссылки к файлу при всех последующих операциях ввода/вывода.

Все потоковые функции ввода/вывода обеспечивают буферизированный, форматированный или неформатированный ввод и вывод. Буферизированный поток обеспечивает место для промежуточного хранения всей информации, вводимой из потока или записываемой в поток.

Поскольку дисковый ввод/вывод занимает довольно много времени, буферизация потока разгружает приложение. Вместо того чтобы вводить данные из потока по одному символу или по одному элементу данных, потоковые функции ввода/вывода получают данные поблочно. Когда приложению необходимо обработать введенную информацию, оно просто обращается к буферу, что гораздо быстрее. Когда буфер становится пустым, выполняется считывание с диска другого блока.

Во многих языках высокого уровня существует одна проблема с буферизированным вводом/выводом, которую нужно принимать во внимание. Например: **если ваша программа выполнила несколько операторов вывода, которые не заполнили буфер вывода, и запись на диск не произошла, то по завершении программы эта информация будет потеряна.** Для решения этой проблемы обычно выполняется вызов соответствующей функции для очистки буфера. В отличие от других языков высокого уровня, в языке С данная проблема с буферизированным вводом/выводом решается путем автоматической очистки содержимого буфера по завершении программы. Конечно, хорошо написанное приложение не должно рассчитывать на эти автоматические действия; все действия программы должны описываться в явном виде. Дополнительное замечание: если вы используете потоковый ввод/вывод и приложение заканчивается с аварийным остановом, то буферы вывода могут оказаться неочищенными, что приведет к потере данных.

Аналогичным образом выглядят процедуры, **работающие с консолью и портами**; их можно рассматривать как расширенные потоковые функции. Они позволяют читать и писать на терминал (консоль) или в порт ввода/вывода (например, в порт принтера). Функции портов ввода/вывода выполняют простое побайтное считывание и запись. Функции ввода/вывода на консоль обеспечивают несколько дополнительных возможностей. Например, можно определить: введен ли с консоли символ или имеют ли вводимые символы эхо-отображение на экране. Последним типом ввода и вывода является низкоуровневый. Функции **низкоуровнего ввода/вывода** не выполняют никакой буферизации и форматирования; они непосредственно обращаются к средствам ввода и вывода операционной системы. Эти функции позволяют обращаться к файлам и периферийным устройствам на более низком уровне, чем это делают потоковые функции. При открытии файла на этом уровне возвращается описатель файла (file handle), представляющий собой целое число, использующееся затем для обращения к этому файлу при последующих операциях. В общем случае не рекомендуется смешивать функции потокового ввода/вывода с низкоуровневыми. Поскольку потоковые функции являются буферизированными, а низкоуровневые — нет, при обращении к файлу или устройству при помощи двух разных способов возможны рассогласование или даже потеря данных в буферах. Поэтому для каждого конкретного файла необходимо использовать либо потоковые, либо низкоуровневые функции. В табл. 17.1 перечислены наиболее часто используемые в С функции потокового ввода/вывода.

Таблица 17.1. Функции С потокового ввода и вывода

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| clearer() | Сбрасывает индикатор ошибок потока и устанавливает в ноль индикатор конца файла (end-of-file) |
| fclose() | Закрывает поток |
| fcloseall() | Закрывает все потоки |
| fdopen() | Открывает поток, используя его описатель (handle), полученный при помощи creat, dup, dup2 или open |
| feof() | Проверяет поток на признак конца файла |
| ferror() | Проверяет поток на признак конца файла |
| fflush() | Проверяет поток на признак конца файла |
| fgetc() | Считывает символ из потока |
| fgetchar() | Считывает символ из потока stdin |
| fgetpos() | Возвращает текущий указатель файла |
| fgets() | Возвращает строку из потока |
| filelength() | Возвращает размер потока в байтах |
| fileno() | Возвращает описатель файла, связанного с потоком |
| flushall() | Очищает буферы всех потоков |
| fopen() | Открывает поток |
| fprintf() | Выполняет форматированную запись в поток |
| fputc() | Записывает символ в поток |
| fputchar() | Записывает символ в stdout |
| fputs() | Записывает строку в поток |
| fread() | Считывает неформатированные данные из потока |
| freopen() | Переназначает указатель потока |
| fscanf() | Считывает форматированные данные из потока |
| fseek() | Позиционирует указатель файла в указанную позицию |
| fsetpos() | Позиционирует указатель файла потока |
| fstat() | Возвращает информацию об открытом файле |
| ftell() | Возвращает положение текущего указателя файла |
| fwrite() | Записывает неформатированные данные в поток |
| getc() | Этот макрос считывает символ из потока |
| getchar() | Этот макрос считывает символ из stdin |
| gets() | Возвращает строку из stdin |
| getw() | Считывает целое число из потока |
| perror() | Выдает системную ошибку на stderr |
| printf() | Записывает форматированные данные в stdout |
| putc() | Этот макрос записывает символ в поток |
| putchar() | Этот макрос записывает символ в stdout |
| puts() | Записывает строку в stdout |
| putw() | Записывает целое число в поток |
| remove() | Удаляет файл |
| rename() | Переименует файл |
| rewind() | Позиционирует указатель файла на начало потока |
| scanf() | Сканирует и вводит форматированные данные из stdln |
| setbuf() | Отменяет автоматическую буферизацию, позволяя приложению определить собственный буфер для потока |
| setvbuf() | То же самое, что и setbuf(); позволяет также определить размер буфера |
| sprintf() | Записывает форматированные данные в строку |
| sscanf() | Сканирует и вводит форматированные данные из строки |
| tmpnam() | Создает в заданном подкаталоге файл с уникальным именем |
| ungetch() | Засылает символ в буфер клавиатуры |
| vfprintf() | Записывает форматированные данные в поток, используя указатель на форматирующую строку |
| vfscanf() | Сканирует и форматирует ввод из потока, используя указатель на форматирующую строку |
| vprintf() | Записывает форматированные данные в stdout, используя указатель на форматирующую строку |
| vscanf() | Сканирует и форматирует ввод из stdin, используя указатель на форматирующую строку |
| vsprintf() | Записывает форматированные данные в строку, используя указатель на форматирующую строку |
| vsscanf() | Сканирует и форматирует ввод строки, используя указатель на форматирующую строку |

## Соединение и отсоединение потока от файла.

### Функция fopen

Для **соединения потока** с файлом используется функция:

FILE\* fopen(constchar\* filename, constchar\* mode);

которая открывает файл, имя которого задано параметром filename, в режиме, заданнном параметром mode. В случае успешного завершения функция fopen возвращает указатель на поток, а в случае неудачи – NULL.

Параметр mode может принимать следующие значения:

“r” – чтение в текстовом режиме,

“w” – запись в текстовом режиме,

“a” – присоединение в текстовом режиме,

“rb” – чтение в бинарном режиме,

“wb” – запись в бинарном режиме,

“ab” – присоединение в бинарном режиме,

“r+” или “w+” или “a+” – чтение и запись в текстовом режиме,

“r+b” или “w+b” или “a+b” – чтение и запись в бинарном режиме,

“rb+” или “wb+” или “ab+” – чтение и запись в бинарном режиме.

При открытии файла в режимах “r”, “rb”, “r+”, “r+b” его индикатор позиции устанавливается на начало файла. В случае, если открывается несуществующий файл, то функция fopen заканчивается неудачей.

При открытии файла в режимах “w”, “wb”, “w+”, “w+b” создается новый файл. Если файл с заданным именем существует, то его содержимое стирается, а индикатор позиции устанавливается на начало файла.

При открытии файла в режимах “a”, “ab”, “a+”, “a+b” создается новый файл. Если файл с заданным именем существует, то он открывается, и индикатор позиции устанавливается на конец файла.

Следует учитывать, что если текстовый файл открывается в режиме чтения и записи, то базовая операционная система может открыть его в бинарном режиме. Максимальное количество файлов, которые можно открыть одновременно задается переменной FOPEN\_MAX (равно 20 в Visual C++). Максимальная длина имени файла задается переменной FILENAME\_MAX (равно 260 в ОС Windows).

### Функция fclose

Для **отсоединения потока** от файла используется функция:

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

которая закрывает файл, при этом освобождая все буферы потока. При успешном завершении функция возвращает 0, а в случае неудачи – EOF.

### Функция freopen

Для **перенаправления потока** используется функция:

FILE\* freopen(constchar\* filename, constchar\* mode, FILE\* stream);

которая закрывает файл, соединенный с потоком **stream**, и соединяет с этим потоком файл **filename** в режиме **mode**. В случае успеха функция возвращает указатель на поток, а в случае неудачи – NULL. Параметр mode принимает те же значения, что и в функции fopen.

Поясним подробнее разницу между текстовым и бинарным режимами работы потока. Как в текстовом так и в бинарном режиме можно использовать все функции для доступа к файлу. **При работе в бинарном режиме поток записывает на диск и считывает с диска точные копии данных, переданные функциями записи данных на диск и требуемые функциями чтения данных с диска соответственно.** Работа потока в текстовом режиме отличается тремя моментами от работы потока в бинарном режиме. **Во-первых**, в этом случае символ CTRL+Z интерпретируется как конец файла. **Во-вторых**, при записи в текстовый поток из комбинации символов “\r” (carriage return, возврат каретки) и “\n” (new line, новая строка) в файл записывается только символ “\n”, а при чтении из текстового потока символ “\n” преобразуется в комбинацию символов “\r”+”\n”. **В-третьих**, так как при записи в текстовый поток может происходить преобразование количества и представления символов, то для получения требуемой позиции в файле нужно использовать только функции **fgetpos** и **ftell**.

## Работа с индикаторами ошибки, позиции и конца файла (ferror, clearerr, feof, rewind, fseek, fsetpos, ftell, fgetpos).

**1. Работа с индикатором ошибки.** С каждым потоком связан индикатор ошибки, который находится в установленном положении, если в потоке, связанном с файлом произошла ошибка. В противном случае индикатор ошибки находится в сброшенном состоянии. Для работы с индикатором ошибки используются функции **ferror** и **clearerr**.

Функция

int **ferror(FILE\* stream);**

возвращает ненулевое значение, если индикатор ошибки установлен, в противном случае – возвращает 0.

Функция

void **clearerr(FILE\* strem);**

сбрасывает индикаторы ошибки и конца файла для потока **stream**.

**2.** **Работа с индикатором конца файла.** Структура **FILE** содержит индикатор конца файла, который устанавливается в ненулевое значение функцией чтения из файла при достижении этой функцией конца файла. Состояние конца файла читается функцией

int **feof(FILE\* stream);**

которая возвращает ненулевое значение, если индикатор конца файла установлен, в противном случае функция возвращает 0.

**3. Работа с индикатором позиции.** Для каждого файла, после его открытия, определяется индикатор позиции, который указывает на смещение от начала файла в байтах. Тип индикатора позиции определяется как

typedeflong **fpos\_t;**

Для работы с индикатором позиции предназначены следующие функции.

Функция

void **rewind(FILE\* stream);**

устанавливает индикатор позиции на начало файла, связанного с потоком **stream**. При этом сбрасывется индикатор ошибки и конца файла.

Функция

int **fseek(FILE\* stream,** long **offset,** int **mode);**

сдвигает индикатор позиции файла на **offset** байт. В случае успешного завершения функция возвращает **0**, в противном случае – ненулевое значение. Параметр **mode** указывает на режим сдвига и может принимать следующие значения:

SEEK\_SET – смещение от начала файла,

SEEK\_CUR – смещение от текущей позиции,

SEEK\_END – смещение от конца файла.

При работе с текстовым потоком должны использоваться только следующие комбинации значений параметров:

mode = SEEK\_SET offset = 0 или

offset = значению, возвращаемому функцией ftell

mode = SEEK\_CUR offset = 0

mode = SEEK\_END offset = 0

Функция

int **fsetpos(FILE\* stream,** const **fops\_t \*pos);**

устанавливает индикатор позиции файла **stream** в позицию, на которую указывает параметр **pos**. Индикатор конца файла сбрасывается. В случае успеха функция возвращает 0, а в случае неудачи возвращает ненулевое значение и устанавливает переменную **errno**.

Функция

long **ftell(FILE\* stream);**

в случае успешного завершения возвращает текущую позицию файла **stream**, а в случае неудачи – возвращает значение **1L** и устанавливает значение переменной **errno**. Для бинарного потока позиция равна смещению в байтах от начала файла, а в случае текстового потока – значению, которое может использоваться функцией **fseek**.

Функция

int **fgetpos(FILE\* stream, fops\_t\* pos);**

записывает текущую позицию файла **stream** по адресу **pos**. В случае успеха функция возвращает 0, а в случае неудачи – ненулевое значение и устанавливает значение переменной **errno**.

## Блочный ввод-вывод (fwrite, fread).

Блоком называется область оперативной памяти, содержимое которой записывается в файл. Ввод-вывод блоками используется **бинарными потоками**.

Для записи блока в файл используется функция

size\_t fwrite(constvoid\* ptr, size\_t size, size\_t nitems, FILE\* stream);

которая записывает содержимое блока памяти, на который указывает **ptr**, в файл **stream**. Длина записываемого блока определяется как произведение **size\*nitems**. Функция возвращает число записанных единиц памяти. В случае удачи это число должно быть равно **nitems**.

Для чтения блока из файла используется функция

size\_t fread(constvoid\* ptr, size\_t size, size\_t nitems, FILE\* stream);

параметры которой имеют тот же смысл, что и в функции fwrite.

По стандарту состояние индикатора позиции после работы функций **fwrite** и **fread** не определено.

В следующей программе создается бинарный файл.

// создание бинарного файла

#include<stdio.h>

struct **emp**

{

int **code;**

char **name[20];**

double **salary;**

};

int **main()**

{

**FILE\* out;** // выходной поток

struct **emp s;** // для записей файла

// открываем выходной поток в бинарном режиме

//if(!(out = fopen("C:\\employee.bin", "wb")))

if(!(out = fopen("employee.bin", "wb")))

**{**

**printf(**"Open file failed.\n");

return **0;**

**}**

**printf(**"Input code, name and salary.\n");

**printf(**"Press Ctrl+z to exit.\n");

**printf(**">");

// вводим первую запись с консоли

**scanf(**"%d%s%lf", &s.code, &s.name, &s.salary);

while **(!feof(stdin))**

**{**

// пишем запись в файл

**fwrite(&s,** sizeof(struct **emp), 1, out);**

**printf(**">");

// вводим следующие записи с консоли

**scanf(**"%d%s%lf", &s.code, &s.name, &s.salary);

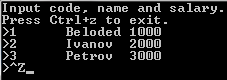
**}**

// закрываем выходной поток

**fclose(out);**

return **1;**

}



В следующей программе выполняется чтение записей из бинарного файла.

// чтение бинарного файла

#include<stdio.h>

struct **emp**

{

int **code;**

char **name[20];**

double **salary;**

};

int **main()**

{

**FILE\* in;** // выходной поток

struct **emp s;** // для записей файла

unsigned **i;** // номер записи

// открываем входной поток в бинарном режиме

//if(!(in = fopen("C:\\employee.bin", "rb")))

if(!(in = fopen("employee.bin", "rb")))

**{**

**printf(**"Open file failed.\n");

return **0;**

**}**

**printf(**"Press Ctrl+z to exit.\n");

// читаем индекс

**printf(**"Input an index: ");

**scanf(**"%u", &i);

while **(!feof(stdin))**

**{**

// устанавливает указатель на нужную запись

**fseek(in, i\***sizeof(struct **emp), SEEK\_SET);**

// читаем запись из файла

if(!fread(&s, sizeof(struct **emp), 1, in))**

**{**

**printf(**"The wrong index.\n");

continue;

**}**

// выводим запись на консоль

**printf(**"\tcode = %d name = %s sal = %f\n",

**s.code, s.name, s.salary);**

// читаем индекс

**printf(**"Input an index: ");

**scanf(**"%u", &i);

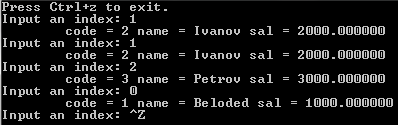
**}**

// закрываем входной поток

**fclose(in);**

return **1;**

}



## Символьный ввод-вывод.

Символьный ввод-вывод используется с **текстовыми потоками**.

### Функции fputc, putc, fgetc, getc, ungetc

**Ввод-вывод символов.** Для записи и чтения символов из текстового файла используются функции **fputc, putc, fgetc, getc**.

Функция

int **fputc(**int **c, FILE\* stream);**

записывает символ **с** в поток **stream** и продвигает индикатор позиции на следующий символ. В случае успеха функция возвращает символ **с**, а в случае неудачи **EOF** и устанавливает индикатор ошибки.

Функция

int **putc(**int **c, FILE\* stream);**

работает так же как и функция fputc, но может быть реализована как макрокоманда.

Функция

int **fgetc(FILE\* stream);**

читает символ из потока **stream** и продвигает индикатор позиции на следующий символ. В случае успешного завершения функция возвращает прочитанный символ. В случае достижения конца файла функция возвращает **EOF** и устанавливает индикатор конца файла. В случае ошибки функция возвращает **EOF** и устанавливает индикатор ошибки.

Функция

int **getc(FILE\* stream);**

работает так же кака и функция fgetc, но может быть реализована как макрокоманда.

Функция

int **ungetc(**int **c, FILE\* stream);**

записывает символ **с** в поток **stream**. Функции **fseek, fsetpos** и **rewind** игнорируют такие символы. Доступ к записанным символам выполняется по правилу **FIFO**. В случае успеха функция возвращает записанный символ, а в случае неудачи – **EOF**.

### Функции fputs и fgets

**Ввод-вывод строк.** Для записи и чтения строк из символьного потока используются символы fpus и fgets.

Функция

int **fputs(**constchar\* str, FILE\* stream);

записывает строку **str** в файл **stream**, не включая завершающий нулевой байт. В случае успешного завершения функция возвращает ненулевое число, а в случае неудачи – **EOF**.

Функция

int **fgets(**char\* str, int **n, FILE\* stream);**

читает строку из потока **stream** в строку **str**. Останавливается функция в случае, если прочитан (n-1) символ, или встретился символ **‘\n’**, или достигнут конец файла.. В любом из этих случаев в конец строки помещается символ **‘\n’**. В случае успеха функция возвращает указатель **str**, а в случае неудачи – **NULL**. Строка **str** не изменяется, если не прочитан ни один символ и встретился конец файла.

### Функции fprintf, fscanf

**Форматированный ввод-вывод.** Для форматированного ввода-вывода в текстовые файлы используются функции **fscanf** и **fprintf**.

Функция

int **fprintf(FILE\* stream,** constchar\* format, …);

выполняет вывод в файл **stream** в соответствии с форматной строкой **format**. Работает эта функция так же, как и функция форматирования строк **sprintf**, которая была рассмотрена в лабораторной работе №5.

Функция

int **fscanf(FILE\* stream,** constchar\* format, …);

выполняет ввод из файла **stream** текста в соответствии с форматной строкой **format**. Работает эта функция так же, как и функция форматирования строк **sscanf**, которая была рассмотрена в лабораторной работе №5.

В следующей программе создается текстовый файл.

// создание текстового файла

#include<stdio.h>

int **main()**

{

int **code;**

char **name[80];**

double **salary;**

**FILE\* out;** // выходной поток

// открываем выходной поток в текстовом режиме

**//**if(!(out = fopen("C:\\employee.txt", "w")))

if(!(out = fopen("employee.txt", "w")))

**{**

**printf(**"Open file failed.\n");

return **0;**

**}**

**printf(**"Input code, name and salary.\n");

**printf(**"Press Ctrl+z to exit.\n");

**printf(**">");

// вводим первую запись с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

while **(!feof(stdin))**

**{**

// пишем запись в файл

**fprintf(out,** "%d %s %f ", code, name, salary);

**printf(**">");

// вводим следующие записи с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

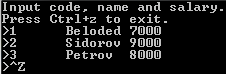
**}**

// закрываем выходной поток

**fclose(out);**

return **1;**

}



В следующей программе читается текстовый файл.

// чтение текстового файла

#include<stdio.h>

int **main()**

{

int **code;**

char **name[80];**

double **salary;**

**FILE\* in;** // входной поток

// открываем входной поток в текстовом режиме

//if(!(in = fopen("C:\\employee.txt", "r")))

if(!(in = fopen("employee.txt", "r")))

**{**

**printf(**"Open file failed.\n");

return **0;**

**}**

// читаем первую запись

**fscanf(in,** "%d%s%lf", &code, name, &salary);

while **(!feof(in))**

**{**

// выводим запись на консоль

**printf(**"code = %d name = %s sal = %f\n", code, name, salary);

// читаем следующие записи

**fscanf(in,** "%d%s%lf", &code, name, &salary);

**}**

// закрываем входной поток

**fclose(in);**

**scanf(**"%u", &code);

return **1;**

}



## Работа с буферами (setvbuf, setbuf, fflush).

**Буфером** называется область оперативной памяти, используемая потоком для временного хранения данных из файла. Для работы с буферами используются функции **setvbuf, setbuf, fflush**.

Функция

int **setvbuf(FILE\* stream,** char\* buffer, int **mode, size\_t size);**

определяет буфер ввода-вывода и режим работы с ним для потока **stream**. Вызывается эта функция после открытия файла, но перед доступом к нему. При успешном завершении функция возвращает значение **0**, а вслучае неудачи – ненулевое значение.

Параметр **buffer** указывает на блок памяти для буфера. Если этот параметр равен **NULL**, то функция **setvbuf** использует функцию **malloc** для захвата памяти под буфер.

Параметр **mode** определяет режим работы с буфером и может принимать следующие значения:

\_IOFBF вывод данных из буфера во внешнюю память выполняется только при полной загрузке буфера или при закрытии файла;

\_IOLBF вывод данных из буфера во внешнюю память выполняется при записи в буфер символа ‘\n’;

\_IONBF нет буферизации, в этом случае параметры size и buffer игнорируются.

Параметр **size** определяет длину буфера в байтах.

Функция

int **setbuf(FILE\* stream,** char\* buffer);

вызывает функцию **setvbuf**. Причем, если значение параметра **buffer** не равно **NULL**, то функция **setvbuf** вызывается следующим образом:

setvbuf(stream, 0, \_IOFBF, BUFSIZE);

где константа **BUFSIZE** задает длину буфера по умолчанию. Эта константа описана в заголовочном файле stdio.h. В противном случае функция **setvbuf** вызывается следующим образом:

setvbuf(stream, 0, \_IOFBF, BUFSIZE);

То есть в этом случае буферизация не используется.

Функция

int **fflush(FILE\* stream);**

записывает данные из буфера потока **stream**, в соединенный с этим потоком файл. В случае успешного завершения функция возвращает значение **0**, а в случае неудачи – **EOF**. Если значение параметра **stream** равно **NULL**, то освобождаются буферы всех потоков, которые работают в режиме вывода.

В следующей программе создается текстовый файл с буфером.

// создание файла с буфером

#include<stdio.h>

int **main()**

{

int **code;**

char **name[80];**

double **salary;**

**FILE\* out;**

constunsigned **size = 1024;** // размер буфера

char **buffer[size];** // буфер потока

// открываем выходной поток в текстовом режиме

//if(!(out = fopen("C:\\employee.txt", "w")))

if(!(out = fopen("employee.txt", "w")))

**{**

**printf(**"Open file failed.\n");

return **0;**

**}**

// устанавливаем буфер для потока

if(setvbuf(out, buffer, \_IOFBF, size))

**{**

**printf(**"Set buffer failed.\n");

return **0;**

**}**

**printf(**"Input code, name and salary.\n");

**printf(**"Press Ctrl+z to exit.\n");

**printf(**">");

// вводим первую запись с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

while **(!feof(stdin))**

**{**

// пишем запись в файл

**fprintf(out,** "%d %s %f ", code, name, salary);

**printf(**">");

// вводим следующие записи с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

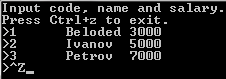
**}**

// закрываем выходной поток

**fclose(out);**

return **1;**

}



В следующей программе создается текстовый файл без буфера.

// создание файла без буфера

#include<stdio.h>

int **main()**

{

int **code;**

char **name[80];**

double **salary;**

**FILE\* out;**

// открываем выходной поток в текстовом режиме

//if(!(out = fopen("C:\\employee.txt", "w")))

if(!(out = fopen("employee.txt", "w")))

**{**

**printf(**"Open file failed.\n");

return **0;**

**}**

// нет буферизации

**setbuf(out, NULL);**

**printf(**"Input code, name and salary.\n");

**printf(**"Press Ctrl+z to exit.\n");

**printf(**">");

// вводим первую запись с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

while **(!feof(stdin))**

**{**

// пишем запись в файл

**fprintf(out,** "%d %s %f ", code, name, salary);

**printf(**">");

// вводим следующие записи с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

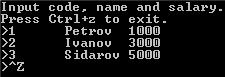
**}**

// закрываем выходной поток

**fclose(out);**

return **1;**

}



## Стандартные потоки (putchar, getchar, puts, gets, perror).

Каждой программе предоставляются **три стандартных потока**, которые по умолчанию соединены с консолью. Указатели на эти потоки возвращают макрокоманды **stdin, stdout, stderr**. Для работы со стандартными потоками предназначены рассмотренные ранее функции **scanf, printf**, а также следующие функции:

int **putchar(**int **c);** // вывод символа в stdout

int **getchar(**void); // ввод символа из stdin

int **puts(**constchar\* str); // вывод строки в stdout

char\* gets(char\* str); // ввод строки из stdin

void **perror(**constchar\* str); // вывод сообщения об ошибке в stderr

Работают эти функции так же, как и аналогичные функции для работы с файлами.

## Служебные функции для работы с файлами (remove, rename, tmpfile, tmpnam).

Здесь перечислим служебные функции для работы с файлами, которые не входят ни в одну из вышеперечисленных категорий. К ним относятся функции **remove, rename, tmpfile, tmpname**.

Функция

int **remove(**constchar\* filename);

удаляет файл с именем **filename**. Если файл открыт, то работа функции зависит от реализации. В случае успешного завершения функция возвращает **0**, а в случае неудачи – ненулевое значение.

В следующей программе показывается, как удалить файл.

// удаление файла

#include<stdio.h>

int **main()**

{

//if(remove("C:\\employee.bin"))

if(remove("employee.bin"))

**{**

**printf(**"There is no such a file.\n");

return **0;**

**}**

**printf(**"The file was deleted.\n");

return **1;**

}

Функция

int **rename(**constchar\* old\_filename, constchar\* new\_filename);

переименовывает файл с именем **old\_filename** в файл с именем **new\_filename**. Если файл с именем **new\_filename** уже существует, то работа функции зависит от реализации. В случае успешного завершения функция возвращает **0**, а в случае неудачи – ненулевое значение.

В следующей программе показывается, как переименовать файл.

// переименование файла

#include<stdio.h>

int **main()**

{

//if(rename("C:\\employee.txt", "C:\\emp.txt"))

if(rename("employee.txt", "emp.txt"))

**{**

**printf(**"There is no such a file.\n");

return **0;**

**}**

**printf(**"The file was renamed.\n");

return **1;**

}

Функция

FILE\* tmpfile(void);

создает временный файл в режиме “**w+b**”. После закрытия потока файл удаляется. В случае успешного завершения функция возвращает указатель на файл, а в случае неудачи – **NULL**.

Функция

char\* tmpnam(char\* str);

возвращает имя для временного файла. Максимальное количество имен равно **TMP\_MAX**, а максимальная длина имени равна **L\_tmpnam**. Если значение параметра **str** равно NULL, то функция возвращает указатель на свою строку, в противном случае возвращается указатель **str**.

Следующая программа показывает пример использования временного файла.

// использование временного файла

#include<stdio.h>

int **main()**

{

int **code;**

char **name[80];**

double **salary;**

**FILE\* temp;** // временный файл

// открываем временный файл

if(!(temp = tmpfile()))

**{**

**printf(**"Create temp file failed.\n");

return **0;**

**}**

**printf(**"Input code, name and salary.\n");

**printf(**"Press Ctrl+z to exit.\n");

**printf(**">");

// вводим первую запись с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

while **(!feof(stdin))**

**{**

// пишем запись в файл во временный файл

**fprintf(temp,** "%d %s %f ", code, name, salary);

**printf(**">");

// вводим следующие записи с консоли

**scanf(**"%d%s%lf", &code, &name, &salary);

**}**

// устанавливаем индикатор позиции на начало файла

**rewind(temp);**

**printf(**"\nRead records from the temporary file.\n");

// читаем первую запись из временного файла

**fscanf(temp,** "%d%s%lf", &code, name, &salary);

while **(!feof(temp))**

**{**

// выводим запись на консоль

**printf(**"code = %d name = %s sal = %f\n", code, name, salary);

// читаем следующие записи из временного файла

**fscanf(temp,** "%d%s%lf", &code, name, &salary);

**}**

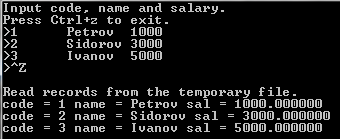
// закрываем временный файл

**fclose(temp);**

**scanf(**"%d", &code);

return **1;**

}



## Низкоуровневый ввод и вывод в С.

В следующем списке перечислены наиболее часто используемые в программах функции низкоуровневого ввода и вывода:

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| close() | Закрывает дисковый файл |
| lseek() | Поиск указанного байта в файле |
| open() | Открывает дисковый файл |
| read() | Читает данные в буфер |
| unlink() | Удаляет файл из подкаталога |
| write() | Записывает буфер данных |

Функции низкоуровневого ввода и вывода не буферизируют и не форматируют данные. Доступ к файлам, открываемых функциями нижнего уровня, осуществляется при помощи описателя файла (handle — целое число, используемое операционной системой для обращения к файлу). Для открытия файлов используется функция ореn(). Для открытия файла с атрибутами совместного использования (sharing) можно использовать макрос sopen().

**Низкоуровневые функции отличаются от своих аналогов — потоковых функций — тем, что они не требуют включения заголовочного файла stdio.h.** Однако, могут быть полезны некоторые часто используемые константы, предопределенные в **stdio.h**: например, **EOF** и **NULL**. Объявления низкоуровневых функций дается в заголовочном файле io.h.

Первоначально данная система дискового ввода/вывода была создана для операционной системы UNIX. Поскольку комитет по стандарту ANSI С решил не стандартизовывать эту UNIX-подобную систему небуферизированного, низкоуровневого ввода/вывода, то ее нельзя рекомендовать для дальнейшего использования. Для всех новых разработок рекомендуется использовать стандартизованную систему буферизированного ввода/вывода.

**File descriptors - open, close, read, write**

В языке C есть много способов работы с файлами. Помимо структуры FILE можно использовать так называемые дескрипторы файла (file descriptors). Дескриптор файла -- целое неотрицательное число. Оно обозначает номер открытого файла в таблице открытых файлов операционной системы. Использование дескрипторов файла -- более низкий уровень, чем нежели ипользование струкруты FILE. Структура FILE -- сущность языка C и его стандартной библиотеки, тогда как дескриптор файла -- сущность операционной системы. Например, при работе со структурой FILE автоматически создается буфер, и программист работает с более высокоуровневой абстракцией. А при работе с дескрипторами файла программист должен позаботится о буферизации вручную.

Пример работы с дескрипторами файла довольно прост и почти в точности повторяет процесс работы со структурой FILE:

int **fd = open(**"...");

Сходство работы с дескрипторами файла с работой со структурой FILE заключается в том, что в названии функций отсутствует буква "f". Иногда параметры функций незначительно отличаются.

Структуру FILE полезно использовать при работе с настоящими "файлами" (которые находятся на жестком диске). Ипользовать дескрипторы файла полезно в случаях работы со специальными "файлами". В этом подходе есть своя специфика работы, но сейчас просто полезно знать, что такой подход существует.

Аналогами stdin, stdout и stderr в дескрипторах файла являются числа 0, 1 и 2 соответственно. Стандарт POSIX.1 обозначил числа 0, 1, 2 символическими константами STDIN\_FILENO, STDOUT\_FILENO и STDERR\_FILENO соответственно.

**Memory mapping. Функция mmap**

Следующий способ работы с файлами удобен в тех случаях, когда приходится читать файл нелинейно: надо "ходить" вперед и назад. В предыдущих подходах такие ситуации оказывались неудобными с точки зрения программирования: получился бы громоздкий код.

В языке C был придуман удобный способ работы в таких ситуациях, который называется **memory mapping**. Соответствующая функция:

char **\*ptr = mmap(**"...");

Работает эта функция примерно так. Мы указываем этой функции файл на диске, и она "отображает" этот файл в такую-то область в памяти. В результате работы функции мы получаем указатель на начало файла. И потом мы можем работать с этим файлом как с обычным указателем на какую-то область памяти: можем "ходить" вперед и назад по этому файлу.

Можно "отобразить" не весь файл целиком, а, например, отдельную часть файла: с 3-его килобайта по 4-ый килобайт.

**Win32 API: FileCreate, FileRead, etc.**

При работе с файлами в ОС Windows можно использовать все те функции, которые были описаны выше. В ОС Windows есть своя большая стандартная библиотека **Win32 API**. В этой библиотеке также есть функции для работы с файлами: например, функции FileCreate(...) или FileOpen(...). Они по своей работе похожи на функции из стандартной библиотеки C, но отличия также присутствуют. Они заключаются в параметрах этих функций и небольших "хитростях", которые мы здесь опустим.

Если вы программируете под ОС Windows и пишите программу для работы в ОС Windows, то стоит пользоваться библиотекой Win32 API для работы с файлами.

## Форматированный вывод.

Богатый ассортимент средств управления форматом вывода в С позволяет легко создавать печатные графики, отчеты или таблицы. Двумя основными функциями, выполняющими этот форматированный вывод, являются **printf()** и эквивалентная функция для файлов — **fprintf()**. Эти функции могут использовать любые символы преобразования, показанные в табл. 11.2. Спецификация формата имеет следующий вид:

%[флаги] [ширина] [.точность] [{h | 1 | L}]тип

Таблица 11.2. Символы преобразования для функций printf() и fprintf()

|  |  |  |
| --- | --- | --- |
| **ПОЛЕ “ТИП”** | | |
| **Сим-**  **вол** | **Тип** | **Формат вывода** |
| c | int или wint\_t | printf—означает однобайтный символ  wprintf—означает широкий символ |
| C | int или wint\_t | printf—означает широкий символ  wprintf—означает однобайтный символ |
| d | int | Десятичное целое зо знаком |
| e | double | Число со знаком в виде [-]d.ddd e [знак]ddd,где d-отдельная десятичная цифра, ddd-одна или несколько десятичных цифр, ddd-ровно четыре десятичные цифры и знак “+” или ”-” |
| E | double | То же что и “е”, за исключением того, что перед экспонентой стоит “E” |
| f | double | Число со знаком в виде [-]ddd.ddd, где d-одна или несколько десятичных цифр. Число цифр после десятичной точки зависит от точности |
| g | double | Число со знаком в формате ”f” или ”e”. Используется наиболее компактная форма. Отсутствуют нули в младших разрядах. Нет десятичной точки, если за ней отсутствуют цифры |
| G | double | То же что и “g”, за исключением того, что перед экспонентой стоит “G” |
| i | int | Десятичное целое со знаком |
| n | Указатель на целое | Количество символов, записанных в поток или буфер. Адрес буфера, определяемого целочисленным аргументом |
| o | Int | Восьмеричное число без знака |
| p | Указатель на void | Печатается адрес(определяемый аргументом) |
| s | строка | printf – определяет строку однобайтных символов  wprintf - определяет строку широких символов(печать в NULL или с максимальной точностью) |
| S | строка | printf – определяет строку широких символов  wprintf - определяет строку однобайтных символов (печать в NULL или с максимальной точностью) |
| u | строка | Десятичное целое без знака |
| x | int | Шестнадцатеричное целое без знака (используются символы нижнего регистра) |
| X | int | Шестнадцатеричное целое без знака (используются символы верхнего регистра) |

Таблица 11.2 Символы преобразования для функций printf() и fprintf() (продолжение)

|  |  |
| --- | --- |
| **Флаг** | **Значение** |
| **#** | Перед ненулевыми значениями ставиться 0, 0x, или 0X (по умолчанию пробелы отсутствуют) |
| **.** | Для форматов “e”, “E”, или “f” символ “#” указывает на наличие в выходном значении десятичной точки во всех случаях (по умолчанию точка появляется тогда, когда за ним следуют цифры) |
| **-** | Если число введенных символов оказывается меньше указанного, результаты выравниваются по левому краю поля вывода(по умолчанию принято правостороннее выравнивание |
| **+** | При вводе знаковых чисел знак отображается всегда(по умолчанию знак устанавливается только перед отрицательными числами) |
| **0** | Если значению поля ширина предшествует символ ‘0’, выводимое число дополняется ведущими нулями до минимальной ширины поля вывода(по умолчанию в качестве заполнителей применяются пробелы); при левостороннем выравнивании игнорируется |
| **пробел** | Если выводится положительное знаковое число, перед ним ставится пробел(по умолчанию пробел в таких случаях не ставится); игнорируется при наличии флага + |

Поле спецификации является символом или числом, которое задает опцию форматирования. Простейший случай — знак процента и тип: например, %f. Поле "тип" используется для того, чтобы определить, что аргумент должен интерпретироваться как символ, строка или число. Поле "флаги" используется для управления печатью знаков, пробелов, десятичных точек, основания системы счисления выходных значений и так далее. Поле "ширина" указывает минимальное число печатаемых символов. Поле "точность" указывает максимальное число символов, печатаемых при выводе. Символы h | l | L являются необязательными префиксами, определяющими размер аргумента.

Все функции (в том числе scanf(), fscanf() и sscanf()) можно использовать для ввода чрезвычайно сложных данных. Взгляните, например, на следующий оператор:

scanf("%2d%5s%4f", &ivalue, psz, &fvalue);

Этот оператор вводит целое число из двух цифр, строку из пяти символов и вещественное число, занимающее максимально четыре позиции (2.97, 12.5 и так далее). Сможете ли вы определить, что выполняет следующий оператор:

scanf ("%\*[\t\n]\"%[^A-Za-z]%[^\"]\"", ps1, ps2);

Вначале оператор считывает, но не запоминает любой пустой символ (пробел). Это выполняется при помощи следующей спецификации форма-та: "%\*[\t\n]". Символ (\*) указывает функции, что она должна получить указанные данные, но не запоминать их ни в одной переменной. Если во входной строке содержатся только символы пробела, табуляции или перевода строки, то функция scanf() продолжает считывание до тех пор, пока не встретит двойную кавычку ("). Для этого используется спецификатор формата \", который указывает на то, что введенное значение должно совпадать с заданным символом. Символ двойной кавычки, однако, не вводится. огда функция scanf() обнаружила двойную кавычку, она получает указание вводить все символы, являющиеся цифрами, в переменную ps1. Для этого в спецификации формата %[^A-Za-z] используется модификатор — знак вставки (^), который указывает на то, что можно вводить любые символы, за исключением заглавных букв от "А" до "Z" и строчных букв от "а" до "z". Если бы знака вставки не было, то строка должна была бы содержать только буквенные символы. Знак тире между двумя символами "А" и "Z" и "а" и "z" говорит о том, что нужно рассматривать весь диапазон значений.

Мы уже использовали ранее функцию scanf() и формат %s для считывания строки. Основное различие между scanf() и gets() заключается в том, как они определяют, что достигли конца строки; scanf () предназначена скорее для получения слова, а не строки. Функция gets (), как мы уже видели, принимает все символы до тех пор, пока не встретит первый символ «новая строка». Функция scanf () имеет два варианта. Для любого из них строка начинается с первого встретившегося непустого символа. Если вы используете формат %s, строка продолжается до (но не включая) следующего пустого символа (пробел, табуляция или новая строка). Если вы определяете размер поля как %10s, то функция scanf() считает не более 10 символов или же считает до любого пришедшего первым пустого символа.

Функция scanf () возвращает целое значение, равное числу считанных символов, если ввод прошел успешно, или символ EOF, если он встретился.

### Использование функций printf() и fprintf().

Мы уже обсуждали функцию **printf()** довольно основательно. Подобно **puts()**, она использует указатель строки в качестве аргумента. Функция **printf()** менее удобна, чем **puts()**, но более гибка. Разница заключается в том, что **printf()** не выводит автоматически каждую строку текста с новой строки. Вы должны указать, что хотите выводить с новых строк. Так,

printf(" %s\n"**, string);**

дает то же самое, что и

puts(string);

Вы можете видеть, что первый оператор требует ввода большего числа символов и большего времени при выполнении на компьютере. С другой стороны, **printf()** позволяет легко объединять строки для печати их в одной строке. Например:

printf("Хорошо, %s, %s\n", name, MSG);

объединяет "Хорошо" с именем пользователя и с символьной строкой MSG в одну строку.

Функции ввода-вывода **fprintf()** и **fscanf()** работают почти как printf() и scanf(), но им нужен дополнительный аргумент для ссылки на сам файл. Он является первым в списке аргументов. Вот пример, иллюстрирующий обращение к этим функциям:

/\* образец использования fprintf() и fscanf() \*/

#include <stdio.h>

void main()

{

FILE \*fi;

int age;

fi = fopen("sam.txt" , "r"); /\* считывание \*/

fscanf( fi, " %d", &age); /\* fi указывает на sam \*/

fclose(fi);

fi = fopen("data.txt" , "a"); /\* дополнение \*/

fprintf( fi, "sam is %d.\n", age); /\* fi указывает на data \*/

fclose(fi);

}

Заметим, что можно было использовать fi для двух различных файлов, потому что мы закрыли первый файл, прежде чем открыть второй.

В отличие от **getc()** и **putc()** эти две функции получают указатель типа **FILE** в качестве первого аргумента. Две другие, описанные ниже, получают его в качестве последнего аргумента.

В следующем примере описываются переменные **четырех типов**: символьная, массив символов, целочисленная и вещественная; затем показано, как для каждой переменной использовать соответствующие средства управления форматированием. Исходный текст программы подробно прокомментирован, и для максимальной простоты использована нумерация строк вывода, связывающая выводимое значение и оператор, обеспечивающий это значение:

/\*Программа на С, демонстрирующая развитые средства преобразования

и форматирования\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void StopWait(void);

main()

{

char c = 'A',

psz1[] = "In making a living today many no ",

/\* Для того, чтобы жить, сегдня многие \*/

psz2[] = "longer leave any room for life.";

/\* не ыыходят из комнат в течение всей жизни. \*/

int iln = 0,

ivalue = 1234;

double dPi = 3.14159265;

/\* преобразования \*/

printf("\n[%2d] %c",++iln,c); /\*01\*//\* печать символа с \*/

printf("\n[%2d] %d",++iln,c); /\*02\*//\* печать ASCII-кода символа с \*/

printf("\n[%2d] %c",++iln,90); /\*03\*//\* печать символа с ASCII-кодом 90 \*/

printf("\n[%2d] %o",++iln,ivalue); /\*04\*//\* печать ivalue в восьмеричном виде \*/

printf("\n[%2d] %x",++iln,ivalue); /\*05\*//\* печать шестнадцатеричного числа буквами нижнего регистра \*/

printf("\n[%2d] %X",++iln,ivalue); /\*06\*//\* печать шестнадцатеричного числа буквами верхнего регистра \*/

/\* опции преобразования и форматирования \*/

printf("\n[%2d] %c",++iln,c); /\*07\*//\* минимальная ширина 1\*/

printf("\n[%2d] %5c",++iln,c); /\*08\*//\* минимальная ширина 5, выравнивание вправо \*/

printf("\n[%2d] %-5c",++iln,c); /\*09\*//\* минимальная ширина 5, выравнивание влево \*/

printf("\n[%2d] %s",++iln,psz1); /\*10\*//\* 33 не-null символа с автоматическим определением ширины \*/

printf("\n[%d] %s",++iln,psz2); /\*11\*//\* 31 не-null символ с автоматическим определением ширины \*/

printf("\n[%d] %5s",++iln,psz1); /\*12\*//\* минимальная ширина 5 переопределена, автоматическая — 33 \*/

printf("\n[%d] %38s",++iln,psz1);/\*13\*//\* минимальная ширина 38, выравнивание вправо \*/

printf("\n[%d] %-38s",++iln,psz2);/\*14\*//\* минимальная ширина 38, выравнивание влево \*/

printf("\n[%d] %d",++iln,ivalue);/\*15\*//\* ширина ivalue по умолчанию 4 \*/

printf("\n[%d] %+d",++iln,ivalue);/\*16\*//\* печать ivalue со знаком "+" \*/

printf("\n[%d] %3d",++iln,ivalue);/\*17\*//\* минимальная ширина 3 переопределена, автоматическая — 4 \*/

printf("\n[%d] %10d",++iln,ivalue);/\*18\*//\* минимальная ширина 10, выравнивание вправо \*/

printf ("\n[%d] %-10d",++iln,ivalue); /\*19\*//\* минимальная ширина 10, выравнивание влево \*/

printf ("\n[%d] %010d", ++iln, ivalue) ; /\*20\*//\* выравнивание влево с дополнением пробелов \*/

printf("\n[%d] %f",++iln,dPi); /\*21\*//\* использование количества цифр по умолчанию \*/

printf("\n[%d] %20f",++iln,dPi); /\*22\*//\* минимальная ширина 20, выравнивание вправо \*/

printf("\n[%d] %020f",++iln,dPi); /\*23\*//\* выравнивание вправо с ведущими нулями \*/

printf("\n[%d] %-20f",++iln,dPi); /\*24\*//\* минимальная ширина 20, выравнивание влево \*/

/\* дополнительная точность при форматировании \*/

printf("\n[%d] %19.19s",++iln,psz1); /\*25\*//\* минимальная ширина 19, печать всех 19 \*/

printf("\n[%d] %.2s",++iln,psz1); /\*26\*//\* печать первых 2-х символов \*/

printf("\n[%d] %19.2s",++iln,psz1); /\*27\*//\* печать 2-х символов, выравнивание вправо \*/

printf("\n[%d] %-19.2s",++iln,psz1); /\*28\*//\* печать 2-х символов, выравнивание влево \*/

printf("\n[%d] %\*.\*s",++iln,19,6,psz1); /\*29\*//\* использование аргументов printf \*/

printf("\n[%d] %10.8f",++iln,dPi); /\*30\*//\* ширина 10; 8 справа от '.' \*/

printf("\n[%d] %20.2f",++iln,dPi); /\*31\*//\* ширина 20; 2 до правой границы выравнивания \*/

printf("\n[%d] %-20.4f",++iln,dPi); /\*32\*//\* 4 десятичных позиций, выравнивание влево \*/

printf("\n[%d] %20.4f",++iln,dPi); /\*33\*//\* 4 десятичных позиций, выравнивание вправо \*/

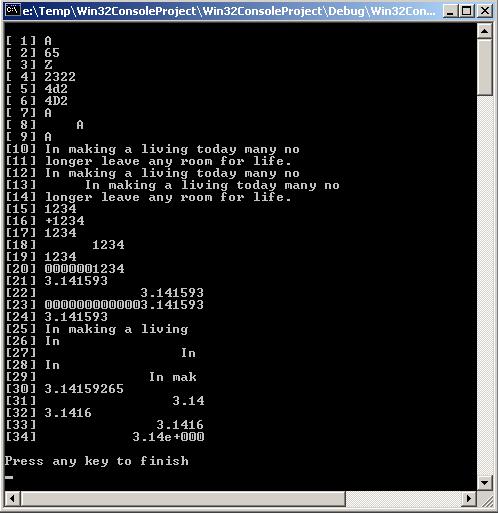
printf("\n[%d] %20.2e",++iln,dPi); /\*34\*//\* ширина 20, экспоненциальное представление \*/

StopWait(); /\* Wait a little \*/

return (0);

}

Результат работы программы выглядит следующим образом:



Изучив приведенный пример, вы сможете легко форматировать выходные данные вашей программы и выбирать те комбинации, которые соответствуют типам данных вашего приложения.

## Примеры

### Содержимое файла

/\* расскажите, что находится в файле "test" \*/

#include<stdio.h>

void **main()**

{

**FILE \*in;** /\* описываю указатель на файл \*/

int **ch;**

if **( (in = fopen(**"test.txt" **,** "r")) != NULL){

/\* открываю test для чтения, проверяя, существует ли он \*/

/\* указатель FILE ссылается теперь на test \*/

while **( (ch = getc(in) )!= EOF)** /\* получаю символ из in \*/

**putc(ch, stdout);** /\* посылаю на стандартный вывод \*/

**fclose(in);** /\* закрываю файл \*/

**}**

else

**printf(**"I couldn't open file \"test\" \n");

}



### Чтение/запись из/в файл

/\*В данной программе на С показано, как объявлять и использовать

файлы для ввода и вывода. Программа считывает значение order\_price

из файла customer.txt и вычисляет значение billing\_price,

записываемое в файл billing.txt\*/

#include<stdio.h>

#include<iomanip.h>

#define **MIN\_DISCOUNT .97**

#define **MAX\_DISCOUNT .95**

main ()

{

float **forder\_price, fbilling\_price;**

**FILE \*fin,\*fout;**

**fin=fopen (**"customer.txt", "r") ;

**fout=fopen (**"billing.txt", "w") ;

while **(fscanf(fin,**"%f",&forder\_price) != EOF) {

**fprintf(fout,**"Your order of \t$%8.2f\n", forder\_price);

if **(forder\_price < 10000)**

**fbilling\_price = forder\_price \* MIN\_DISCOUNT;**

else **fbilling\_price = forder\_price \* MAX\_DISCOUNT;**

**fprintf(fout,**"is discounted to \t$%8.2f.\n\n", fbilling\_price);

**}**

**fclose(fin);**

**fclose(fout);**

return(0);

}

### Назначение буфера потоку stderr

/\*Программа на С, демонстрирующая описание и назначение буфера

потоку stderr\*/

#include"E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void **StopWait(**void);

char **cmyoutputbuffer[BUFSIZ];**

main(void)

{

/\* связь буфера с небуферизированным выходным потоком \*/

**setbuf(stderr, cmyoutputbuffer);** /\* строка, которую нужно \*//\*ЗАКАМЕНТИРУЙ для 2 варианта\*/

/\* заполнение буфера выходного потока \*/

**fputs(**"Sample output inserted into the\n",stderr); /\*"Пример вывода в" \*/

**fputs(**"output stream buffer.\n",stderr); /\* "буфер выходного потока." \*/

/\* вывод буфера выходного потока \*/

**fflush(stderr);**

**StopWait();** /\* Wait a little \*/

return **(0);**

}

### Демо для функции setvbuf()

/\*Программа на С, демонстрирующая использование функции setvbuf()\*/

#include"E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void **StopWait(**void);

#define **MYBUFSIZ 512**

main (void)

{

char **ichar, cmybuffer[MYBUFSIZ];**

**FILE \*pfinfile, \*pfoutfile;**

**pfinfile = fopen(**"E:\\LECTURE\\AlgorithmProgramming\\sample.in", "r");

**pfoutfile = fopen(**"E:\\LECTURE\\AlgorithmProgramming\\sample.out", "w");

if **(setvbuf(pfinfile, cmybuffer, \_IOFBF, MYBUFSIZ) != 0)**

**printf(**"pfinfile buffer allocation error\n"); /\* ошибка назначения \*/

else/\* буфера \*/

**printf(**"pfinfile buffer created\n"); /\* буфер создан \*/

if **(setvbuf(pfoutfile, NULL, \_IOLBF, 132) != 0)**

**printf(**"pfoutfile buffer allocation error\n"); /\* ошибка назначения\*/

else

/\* буфера \*/

**printf(**"pfoutfile buffer created\n");

/\* буфер создан \*/

while(fscanf(pfinfile,"%c",&ichar) != EOF)

**fprintf(pfoutfile,**"%c",ichar);

**fclose(pfinfile);**

**fclose(pfoutfile);**

**StopWait();** /\* Wait a little \*/

return **(0);**

}

### Структура и файл

#include<string.h>

#include<stdio.h>

usingnamespace **std;**

struct **Client{**

int **Num;**

char **SurName[27];**

char **Name[21];**

char **SecName[21];**

};

int **main(**void)

{

**setlocale(LC\_ALL,** "Rus");

**FILE \*stream;**

**Client AClient, RClient;**

//Открываем файл для вывода

if **((stream = fopen(**"SAMPLE.DAT", "wb"))== NULL){

**printf(**"Cannot open output file.\n");

**getchar(); getchar();**

return **1;**

**}**

**AClient.Num = 1;**

**strcpy(AClient.SurName,**"Petrov");

**strcpy(AClient.Name,** "Petr");

**strcpy(AClient.SecName,** "Petrovich");

//Запись структуры в файл

**fwrite(&AClient,** sizeof(AClient), 1, stream);

//Закрываем файл

**fclose(stream);**

//Открываем файл для чтения

if **((stream = fopen(**"SAMPLE.DAT", "rb")) == NULL){

**printf(**"Cannot open input file.\n");

**getchar(); getchar();**

return **2;**

**}**

**fread(&RClient,** sizeof(RClient), 1, stream);

//Закрываем файл fclose(stream);

**printf(**"The structure contains:\n");

**printf(**"Num = %d SurName = %s "

"Name = %s SecName = %s",

**RClient.Num, RClient.SurName,**

**RClient.Name, RClient.SecName);**

**getchar();**

return **0;**

}

### Сжатие файла

В нашем примере имя файла, который следовало открыть, было записано в программе. Мы не обязаны считаться с этим ограничением. Используя аргументы командной строки, можно сообщить нашей программе имя файла, который хотим считать. В нашем следующем примере так и происходит. С помощью, примитивного приема сжимается содержимое — остается только каждый третий символ. Наконец, сжатая версия размещается в новый файл, имя которого состоит из старого имени с добавкой .red (сокращение слова reduced). Обычно весьма важны первый и последний элементы (аргумент командной строки и добавка к имени файла). Само же сжатие имеет более ограниченное применение, но, как вы увидите, им можно пользоваться.

/\* сожмите ваши файлы в 2—3 раза! \*/

#include <stdio.h>

#include<string.h>

void main(int argc, char \*argv[])

{

FILE \*in, \*out; /\* описывает два указателя типа FILE \*/

int ch;

static char name [20]; /\* память для имени выходного файла \*/

int count = 0;

if ( argc < 2) /\* проверяет, есть ли входной файл \*/

printf("Sory, i need the title of the file.\n" );

else

{

if ( (in = fopen(argv[1], "r")) != NULL)

{

strcpy(name, argv[1]); /\* копирует имя файла в массив \*/

strcat(name, " .red"); /\* добавляет .red к имени \*/

out = fopen(name, " w"); /\* открывает файл для записи \*/

while ( (ch = getc(in)) ! = EOF)

if ( count++ %3 ==0)

putc(ch, out); /\* печатает каждый третий символ \*/

fclose(in);

fclose(out);

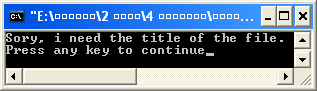
}

else

printf(" Я не смогла открыть файл\" %s\" \n", argv[1]);

}

}



Мы поместили программу в файл, названный reduce и применили эту программу к файлу, названному eddy, который содержал одну единственную строку

Даже Эдди нас опередил с детским хором.

Была выполнена команда

reduce eddy

и на выходе получен файл, названный eddy .red, который содержит

Дед спел тихо

Какая удача! Наш случайно выбранный файл сделал осмысленное сокращение.

Вот некоторые замечания по программе.

Вспомните, что argc содержит определенное количество аргументов, в число которых входит имя программного файла. Вспомните также, что с согласия операционной системы argv[0] представляет имя программы, т. е. в нашем случае reduce. Вспомните еще, что argv[1] представляет первый аргумент, в нашем случае eddy. Так как сам argv[1] является указателем на строку, он не должен заключаться в двойные кавычки в операторе вызова функции.

Мы используем argc, чтобы посмотреть, есть ли аргумент. Любые избыточные аргументы игнорируются. Помещая в программу еще один цикл, вы могли бы использовать дополнительные аргументы — имена файлов и пропускать в цикле каждый из этих файлов по очереди.

С целью создания нового имени выходного файла мы используем функцию strcpy() для копирования имени eddy в массив name. Затем применяем функцию strcat() для объединения этого имени с .red.

Программа требует, чтобы два файла были открыты одновременно, поэтому мы описали два указателя типа ' FILE'. Заметим, что каждый файл должен открываться и закрываться независимо от другого. Существует ограничение на количество файлов, которые вы можете держать открытыми одновременно. Оно зависит от типа системы, но чаще всего находится в пределах от 10 до 20. Можно использовать один и тот же указатель для различных файлов при условии, что они не открываются в одно и то же время.

### Функции fgets и fputs

/\*Программа на С, демонстрирующая считывание записей при помощи

функции fgets и их вывод на stdout при помощи функции fputs\*/

#include"stdafx.h"

#include"E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void **StopWait(**void);

#define **INULL\_CHAR 1**

#define **IMAX\_REC\_SIZE 40**

main()

{

**FILE \*pfinfile;**

char **crecord[IMAX\_REC\_SIZE + INULL\_CHAR];**

**pfinfile=fopen(**"E:\\LECTURE\\AlgorithmProgramming\\boatsale.dat", "r");

while(fgets(crecord,IMAX\_REC\_SIZE +INULL\_CHAR,pfinfile) != NULL)

**fputs(crecord,stdout);**

**fclose(pfinfile) ;**

**StopWait();** /\* Wait a little \*/

return **(0);**

}

### Функции fread и fwrite, fseek, ftell, rewind

#include<string.h>

#include<stdio.h>

usingnamespace **std;**

struct **Client{**

int **Num;**

char **SurName[27];**

char **Name[21];**

char **SecName[21];**

};

int **main(**void)

{

**setlocale(LC\_ALL,** "Rus");

**FILE \*stream;**

**Client AClient, RClient;**

longint **pos;**

//Открываем файл для вывода

if **((stream = fopen(**"SAMPLE.DAT", "wb")) == NULL){

**printf(**"Cannot open output file.\n");

**getchar(); getchar();**

return **1;**

**}**

**AClient.Num = 1;**

**strcpy(AClient.SurName,**"Petrov");

**strcpy(AClient.Name,** "Petr");

**strcpy(AClient.SecName,** "Petrovich");

//Запись структуры в файл

**fwrite (&AClient,** sizeof **(AClient), 1, stream);**

**pos = ftell(stream);**

//Выводим позицию файла и длину структуры

**printf(**"The file pos = %d "

"structure length = %d\n", pos, sizeof(AClient));

//Репозиционируем файл

**rewind(stream);**

//Открываем файл для чтения

if **((stream = fopen(**"SAMPLE.DAT", "rb"))== NULL){

**printf(**"Cannot open input file.\n");

**getchar(); getchar();**

return **2;**

**}**

**fread (&RClient,** sizeof **(RClient), 1, stream);**

//Закрываем файл

**fclose(stream);**

**printf(**"The structure contains:\n");

**printf(**"Num = %d SurName = %s Name = %s"

"SecName = %s",

**RClient.Num, RClient.SurName,**

**RClient.Name, RClient.SecName);**

**getchar(); getchar();**

return **0;**

}



**Пример 2**

/\*Программа на С, иллюстрирующая использование функций fseek,

ftell и rewind\*/

#include"E:\LECTURE\Universal\_HederFile.h"

void **StopWait(**void);

main()

{

**FILE \*pf;**

char **c;**

long **llocation;**

**pf=fopen(**"E:\\LECTURE\\test.dat","r+");

**c=fgetc(pf);**

**putchar(c);**

**c=fgetc(pf);**

**putchar(c);**

**llocation=ftell(pf);**

**c=fgetc(pf);**

**putchar(c);**

**fseek(pf,llocation,0);**

**c=fgetc(pf);**

**putchar(c);**

**fseek(pf,llocation,0);**

**fputc(**'E',pf);

**fseek(pf,llocation,0);**

**c=fgetc(pf);**

**putchar(c);**

**rewind(pf);**

**c=fgetc(pf);**

**putchar(c);**

**StopWait();** /\* Wait a little \*/

return **(0);**

}

### Использование fseek

Для файлов, которые открыты на чтение есть полезные функции. Одна из них это:

int fseek(FILE \*f, long offset, int flag);

* FILE \*f - файл, в котором передвигаемся;
* long offset - количество байтов для отступа, отступ производится в соответствии с 3-м параметром;
* int flag - позиция, от которой будет совершен отступ; в стандартной библиотеке C для этого параметра определены 3 константы:

SEEK\_SET -- начало файла;

SEEK\_CUR -- текущас позиция;

SEEK\_END -- конец файла;

int fseek() -- сама функция возвращает ноль, если операция прошло успешно, иначе возвращается ненулевое значение.

Еще одна полезная функция может определить текущее положение в файле (который открыт для чтения):

long int ftell(FILE \*f);

Вот простой пример, показывающий, как она работает. Как и в наших предыдущих примеpax, функция использует аргумент командной строки для получения имени файла, с которым она работает. Заметим, что fseek() имеет три аргумента и возвращает значение типа int.

Вот простой пример, показывающий, как она работает. Как и в наших предыдущих примеpax, функция использует аргумент командной строки для получения имени файла, с которым она работает. Заметим, что fseek() имеет три аргумента и возвращает значение типа int.

/\* использование fseek() для печати содержимого файла \*/

#include <stdio.h>

void main(int number,char \*names []) /\* не следует использовать argc и argv \*/

{

FILE \*fp;

long offset = 0L; /\* обратите внимание, что это тип long \*/

if ( number < 2)

puts ("I need a name of file as an argument");

else

{

if ((fp = fopen(names[1], "r")) == 0)

printf(" I can't open file %s.\n" , names[1]);

else

{

while( fseek(fp, offset++ , 0) == 0)

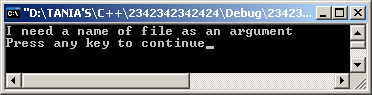
putchar(getc(fp) );

fclose(fp);

}

}

}



Первый из трех аргументов функции fseek() является указателем типа FILE на файл, в котором ведется поиск. Файл следует открыть, используя функцию fopen().

Второй аргумент назван "offset" (вот почему мы выбрали данное имя для переменной). Этот аргумент сообщает, как далеко следует передвинуться от начальной точки (см. ниже); он должен иметь\* значение типа long, которое может быть положительным (движение вперед) или отрицательным (движение назад).

Третий аргумент является кодом, определяющим начальную точку:

**Код Положение в файле**

0 начало файла

1 текущая позиция

2 конец файла

Функция fseek() возвращает 0, если все хорошо, и -1, если есть ошибка, например попытка перемещаться за границы файла.

Теперь мы можем разъяснить наш маленький цикл:

while **( fseek(fp, offset ++ , 0) == 0)**

**putchar(getc(fp) );**

Поскольку переменная offset инициализирована нулем, при первом прохождении через цикл мы имеем выражение

fseek(fp, 0L, 0);

означающее, что мы идем в файл, на который ссылается указатель fp, и находим байт, отстоящий на 0 байт от начала, т. е. первый байт. Затем функция putchar() печатает содержимое этого байта. При следующем прохождении через цикл переменная offset увеличивается до 1L, печатается следующий байт. По существу, переменная offset действует подобно индексу для элементов файла. Процесс продолжается до тех пор, пока offset не попытается попасть в fseek() после конца файла. В этом случае возвращает значение — 1 и цикл прекращается.

Этот последний пример чисто учебный. Нам не нужно использовать fseek(), потому что getc() так или иначе проходит через файл байт за байтом; fseek() приказала getc() «посмотреть» туда, куда она сама уже собиралась посмотреть.