КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 12 — Ввод/вывод

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.11.03

Оглавление

Ввод/вывод <stdio.h>

Введение

Заголовок **<stdio.h>** определяет несколько макросов, а также объявляет три типа и множество функций для выполнения операций ввода и вывода.

Объявленные типы

size_t

целый тип без знака в котором представляется результат оператора **sizeof**.

FILE

тип объекта, способного записывать всю информацию, необходимую для управления потоком, включая *индикатор положения в файле*, указатель на связанный с ним буфер (если таковой есть), *индикатор ошибки*, куда записывается информация, произошла ли ошибка чтения/записи, а также *индикатор конца файла*, куда записывается информация, был ли достигнут конец файла;

fpos_t

это полный тип объекта, отличный от типа массива, способный записывать всю информацию, необходимую для уникального указания любой позиции в файле.

Макросы

NULL

обозначает значение нулевого указателя.

EOF

расширяется до целочисленного константного выражения с типом **int** и отрицательным значением, которое возвращается несколькими функциями для указания *конца файла*, то есть больше ввода из потока нет.

FOPEN_MAX

расширяется до целочисленного константного выражения, которое представляет собой минимальное количество файлов, которые, как гарантирует реализация, могут быть открыты одновременно.

FILENAME_MAX

расширяется до целочисленного константного выражения, которое является размером, массива типа **char**, достаточно большого, чтобы вместить самую длинную строку с именем файла, которую реализация гарантированно может открыть.

Если реализация не накладывает практического ограничения на длину строк имени файла, то значение **FILENAME_MAX** является рекомендованным размером массива, предназначенного для хранения строки с именем файла.

Содержимое строки с именем файла подчиняется другим системным ограничениям, поэтому нельзя ожидать, что все возможные строки длиной **FILENAME_MAX** будут успешно интерпретированы в качестве имени файла и будут открыты.

_IOFBF (полная буферизация ввода/вывода)
_IOLBF (буферизация строками)
_IONBF (небуферизованный ввод/вывод)

расширяются до целочисленных константных выражений с различными значениями, подходящими для использования в качестве третьего аргумента функции **setvbuf**.

BUFSIZ

расширяется до целочисленного константного выражения, которое является размером буфера, используемого функцией **setbuf**.

L_tmpnam

расширяется до целочисленного константного выражения, которое является размером массива **char**, достаточно большого, чтобы содержать строку с именем временного файла, сгенерированную функцией **tmpnam()**.

TMP_MAX

расширяется до целочисленного константного выражения, которое является минимальным количеством уникальных имен файлов, которые могут быть сгенерированы функцией **tmpnam()**.

SEEK_CUR
SEEK_END
SEEK_SET

расширяются до целочисленных константных выражений с различными значениями, подходящими для использования в качестве третьего аргумента функции **fseek**;

stderr
stdin
stdout

являются выражениями типа «указатель на **FILE**» (**FILE** *), которые указывают на объекты типа **FILE**, связанные, соответственно, со стандартными потоками — входным, выходным и ошибок.

В заголовке **<wchar.h>** объявляются ряд функций, полезных для ввода и вывода *широких символов*.

Функции ввода/вывода широких символов обеспечивают операции, аналогичные операциям над однобайтовыми символами, за исключением того, что основными внутренними элементами для программы являются широкие символы.

Внешнее представление (в файле) представляет собой последовательность «обобщенных» многобайтовых символов, как будет описано далее.

В **<stdio.h>** также объявлены функции ввода/вывода

Байтовые функции ввода/вывода:

Функции ввода байтовых символов:
fgetc, **fgets**, **getc**, **getchar**, **fscanf**, **scanf**, **vfscanf**, **vscanf**.

Функции вывода байтовых символов:
fputc, **fputs**, **putc**, **putchar**, **fprintf**, **printf**, **vfprintf**, **vprintf**,
put, **ungetc**, **fread**, **fwrite**.

Функции ввода/вывода широких символов

Функции ввода широких символов – выполняют ввод широких символов и широких строк:
fgetwc, **fgetws**, **getwc**, **getwchar**, **fwscanf**, **wscanf**, **vfwscanf**, **vwscanf**.

Функции вывода широких символов – выполняют вывод широких символов и широких строк:
fputwc, **fputws**, **putwc**, **putwchar**, **fwprintf**, **wprintf**, **vfwprintf**, **vwprintf**, **ungetwc**

Потоки

Ввод и вывод, будь то физические устройства, такие как терминалы и ленточные накопители или файлы, поддерживаемые на структурированных устройствах хранения, отображаются в логические ***потоки*** данных, свойства которых более однородны, чем свойства различных устройств ввода/вывода. **Потоки – это абстракции.**

Поддерживаются две формы отображения устройств:

— на ***текстовые*** потоки;

— на ***двоичные*** потоки.[1]

Текстовый поток – это упорядоченная последовательность символов, состоящая из *строк*, каждая строка состоит из нуля или более символов плюс завершающий символ новой строки. Требует ли последняя строка завершающего символа новой строки, определяется реализацией.

Чтобы соответствовать различным соглашениям для представления текста в среде хоста, как при вводе, так и при выводе, символы могут добавляться, изменяться или удаляться.

Таким образом, взаимно однозначного соответствия между символами в потоке и символами во внешнем представлении может не быть.

Данные, считанные из текстового потока, будут полностью эквивалентны данным, ранее записанными в этот поток, только если:

- данные состоят исключительно из печатных символов, управляющих символов горизонтальной табуляции и символов новой строки;

- символу новой строки не предшествует пробел;

- последний символ – символ новой строки.

Появится ли ли пробел, записанный непосредственно перед символом новой строки в поток при выводе, при чтении из потока, определяется реализацией.

---

1 (std 266) Реализация не обязана различать текстовые потоки и двоичные потоки.

Бинарный поток – это упорядоченная последовательность символов, которая может прозрачно записывать внутренние данные.

Данные, считанные из двоичного потока, полностью эквивалентны данным, ранее записанным в этот поток *в той же реализации*.

Такой поток, однако, может иметь определенное реализацией количество нулевых символов, добавляемых в конец потока.

С каждым из потоков связан объект типа **`FILE`**, который используется для управления этим потоком.

Программе при ее запуске предоставляются три предопределенных потока — **`stdin`**, **`stdout`** и **`stderr`**.

Ориентация потока

Каждый поток имеет ***ориентацию***.

После того, как поток связывается с внешним файлом (открытие), но перед выполнением каких-либо операций с ним, поток ориентации не имеет.

Ориентацию поток приобретает при первом вызове функций ввода/вывода. Как только к потоку без ориентации будет применена функция ввода/вывода широких символов, поток становится потоком, ориентированным на широкие символы (wide-ориентированным потоком).

Точно так же, как только к потоку без ориентации будет применена байтовая функция ввода/вывода, поток становится потоком с байтовой ориентацией.

Изменить ориентацию потока можно вызвав функцию **`freopen`** или функцию **`fwide`**.

Успешный вызов **`freopen`** удаляет любую ориентацию.

Три предопределенных потока **`stdin`**, **`stdout`** и **`stderr`**, предоставляемые программе при запуске, являются неориентированными.

Байтовые функции ввода/вывода не должны применяться к потоку, ориентированному на широкие символы, а широкие символьные функции ввода/вывода не должны применяться к байт-ориентированному потоку.

Остальные операции потока не влияют и не зависят от ориентации потока, за исключением некоторых дополнительных ограничений:

- двоичные потоки, ориентированные на широкие символы, имеют ограничения на позиционирование в файле;

- для потоков, ориентированных на широкие символы, после успешного вызова функции позиционирования файла, которая оставляет индикатор положения в пределах файла, функция вывода широких символов может перезаписать часть многобайтового символа, в связи с чем содержимое файла, позади записанных байтов, с этого момента будет не определено.

Каждый поток, ориентированный на широкие символы, имеет связанный с ним объект типа **`mbstate_t`**, в котором хранится текущее состояние синтаксического разбора потока.

Успешный вызов **`fgetpos`** сохраняет представление значения этого объекта **`mbstate_t`** как часть значения объекта **`fpos_t`**.

Последующий успешный вызов **`fsetpos`** с использованием того же самого сохраненного значения **`fpos_t`** восстанавливает значение связанного объекта типа **`mbstate_t`**, а также позицию в контролируемом потоке.

Каждый поток имеет связанную с ним возможность блокировки, которая используется для предотвращения гонок данных, когда несколько нитей (потоков исполнения) обращаются к потоку, а также для ограничения чередования потоковых операций, выполняемых несколькими нитями.

Только один поток может удерживать эту блокировку одновременно.

Блокировка реентерабельна — один поток может удерживать блокировку несколько раз в данный момент времени.

Все функции, которые читают, пишут, позиционируют или запрашивают позицию потока, перед доступом к потоку его блокируют.

Когда доступ будет завершен они снимают блокировку, связанную с потоком.

Ограничения среды исполнения

Поддерживаются текстовые файлы со строками, содержащими не менее 254 символов, включая завершающий символ новой строки.

Значение макроса **`BUFSIZ`** не может быть менее 256.

Файлы

Поток связывается с внешним файлом (который может быть физическим устройством) с помощью операции открытия файла.

В ряде случаев при этом может происходить ***создание*** нового файла.

Создание существующего файла может приводить, если это необходимо, к удалению его прежнего содержимого.

Если файл может поддерживать запросы позиционирования (например, файл на диске, а не терминал), то в случае, когда файл *не открывается* в режиме добавления, ***индикатор позиции в файле***, связанный с потоком, позиционируется в начало (на нулевой символ) файла.

В случае режима добавления, будет находится индикатор положения файла в начале или в конце файла, определяется реализацией.

Перемещение индикатора положения файла автоматически поддерживается запросами на чтение, запись и позиционирование. Это позволяет работать с файлом упорядоченным образом.

Двоичные файлы обычно не усекаются за точкой записи[2].

Будет ли запись в текстовый поток приводить к усечению файла после точки записи, определяется реализацией.

---

2 За исключением случаев, определенных в fopen()

Буферизация

Когда поток *небуферизован*, символы появляются из источника или записываются в место назначения настолько быстро, насколько это можно.

В случае *буферизованного* потока символы могут накапливаться и передаваться в среду хоста или из нее в виде блоков по мере заполнения буфера.

Когда поток *полностью буферизован* (FBF – fully buffered stream), символы будут передаваться в виде блоков в/из среды хоста только когда буфер заполнится.

Когда поток *буферизирован строками* (LBS – line bufferd stream), символы будут передаваться в или из среды хоста в виде блока всякий раз, как встречается символ новой строки.

В ряде случаев символы могут передаваться блоками по заполнению буфера при запросе ввода из небуферизованного потока или из потока со строчной буферизацией.

Поддержка этих характеристик определяется реализацией и может зависеть от функций **`setbuf`** и **`setvbuf`**.

Закрытие файла

Файл может быть отсоединен от управляющего потока путем ***закрытия*** файла.

Выходные потоки в этом случае сбрасываются (любое незаписанное содержимое буфера передается в хост-среду) до того, как ассоциирование потока и файла будет завершено.

После закрытия ассоциированного файла (включая стандартные текстовые потоки) значение указателя на объект **FILE** становится неопределенным.

Будет ли после закрытия файла существовать в действительности файл нулевой длины (в котором ни один из символов не был записан потоком вывода), определяется реализацией.

Файл может быть впоследствии повторно открыт в процессе того же или другого исполнения программы, а его содержимое восстановлено или изменено (если в файле можно переместиться в начало).

Если функция **main** выполняет возврат к исходному вызвавшему ее объекту или если вызывается функция **exit**, все открытые файлы закрываются (следовательно, все выходные потоки сбрасываются) до того, как программа будет завершена.

Другие способы завершению программы, такие как вызов функции прерывания, не обязательно будут закрывать все файлы должным образом.

Адрес объекта **FILE**, используемого для управления потоком, может быть существенным – копия объекта **FILE** не обязательно сможет работать вместо оригинала.

Предопределенные потоки

При запуске программы три текстовых потока уже предопределены и не требуют явного открытия

- **поток стандартного ввода** (stdin) – для чтения обычного ввода;
- **поток стандартного вывода** (stdout) – для записи обычного вывода;
- **стандартный поток ошибок** (stderr) – для записи диагностического вывода.

При первоначальном открытии стандартный поток ошибок не является полностью буферизованным.

Стандартные входные и стандартные выходные потоки полностью буферизуются тогда и только тогда, когда можно определить, что поток не ссылается на интерактивное устройство.

Функции, которые открывают дополнительные к этим трем (не временные) файлы, требуют имя файла, которое является строкой.

Правила составления допустимых имен файлов определяются реализацией и хост-платформой.

Возможность одновременного открытия одного и того же файла несколько раз также зависит от реализации (хост-платформы).

Текстовые и двоичные потоки, ориентированные на широкие сиволы, с концептуальной точки зрения являются последовательностями широких символов. Однако, внешний файл, связанный с потоком, ориентированным на широкие символы, представляет собой последовательность многобайтовых символов, относительно которых можно в целом сказать следующее:

В отличие от многобайтовых кодировок, допустимых для использования внутри программы, многобайтовые системы кодирования (кодировки) в файлах могут содержать встроенные нулевые байты.

Кроме того, кодировки, используемые для многобайтовых символов, могут различаться в разных файлах. Как природа, так и выбор таких кодировок определяются реализацией.

Функции ввода широких символов читают многобайтовые символы из потока и преобразуют их в широкие символы, как если бы они были прочитаны последовательными вызовами функции **`fgetwc`** (чтение широкого символа из протока).

Каждое преобразование мультибайтового символа в широкий выполняется так, как это происходит как при вызове функции **`mbrtowc`**, причем состояние преобразования описывается собственным объектом потока типа **`mbstate_t`**.

Функции вывода широких символов преобразуют широкие символы в многобайтовые символы и записывают их в поток, как если бы они были записаны последовательными вызовами функции **`fputwc`** (запись широкого символа в поток).

Каждое преобразование широкого символа в мультибайтовый выполняется так, как это происходит при вызове функции **`wcrtomb`**, причем состояние преобразования описывается собственным объектом потока типа **`mbstate_t`**.

Байт-функции ввода читают символы из потока, как будто это делается путем последовательных вызовов функции **fgetc**.

Выходные байтовые функции записывают символы в поток, как будто путем последовательных вызовов функции **fputc**.

В некоторых случаях некоторые из байтовых функций ввода/вывода также выполняют преобразования между многобайтовыми символами и широкими символами. Эти преобразования выполняются так, как если бы они выполнялись функциями **mbrtowc** и **wcrtomb**.

Если последовательность символов, представляемая функцией **mbrtowc**, не образует действительный (обобщенный) многобайтовый символ, или если значение кода, переданное в **wcrtomb**, не соответствует действительному (обобщенному) многобайтовому символу, возникает ошибка кодирования.

Если и только если происходит ошибка кодирования, функции ввода/вывода широких символов и байтовые функции ввода/вывода сохраняют значение макроса **EILSEQ** в **errno**.

Значение **FOPEN_MAX** всегда не менее восьми, включая три стандартных текстовых потока.

Функции доступа к файлам

fopen – открытие файла

```
#include <stdio.h>

FILE *fopen(const char * restrict filename, // строка с именем файла
            const char * restrict mode);    // режим открытия файла
```

Функция **fopen** открывает файл, имя которого указано в строке, на которую указывает **filename**, и связывает с ним поток.

Аргумент **mode** указывает на строку. Если строка является одной из нижеуказанных, файл открывается в указанном режиме. В противном случае поведение не определено.

- **r** открыть текстовый файл только для чтения;
- **w** обрезать до нулевой длины или создать текстовый файл только для записи;
- **wx** создать текстовый файл для записи в монопольном режиме (e**X**clusive);
- **a** добавление; открыть или создать текстовый файл только для записи в конце файла;
- **rb** открыть бинарный файл только для чтения;
- **wb** урезать до нулевой длины или создать двоичный файл только для записи;
- **wbx** создать двоичный файл только для записи в монопольном режиме;
- **ab** добавление; открыть или создать двоичный файл только для записи в конце файла;

r+ открыть текстовый файл для обновления (чтение и запись);
w+ усечь до нулевой длины или создать текстовый файл для обновления;
w+x создать текстовый файл для обновления в монопольном режиме;
a+ открыть или создать текстовый файл для обновления и записи в конец файла;
r+b или **rb+** открыть двоичный файл для обновления (чтение и запись);
w+b или **wb+** усечь до нулевой длины или создать двоичный файл для обновления;
w+bx или **wb+x** *создать* двоичный файл для обновления в монопольном режиме;
a+b или **ab+** открыть или создать двоичный файл для обновления и записи в конец файла;

Режим fopen()	**Флаги open()**
r	O_RDONLY |
w	O_WRONLY | O_CREAT | O_TRUNC
wx	O_WRONLY | O_CREAT | O_EXCL
a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR |
w+	O_RDWR | O_CREAT | O_TRUNC
wx+	O_RDWR | O_CREAT | O_EXCL
a+	O_RDWR | O_CREAT | O_APPEND

O_RDONLY — только для чтения
O_WRONLY — только для записи
O_APPEND — в режиме добавления
O_CREAT — если pathname не существует, то создать обычный файл
O_TRUNC — усечение при записи (O_RDONLY, O_RDWR)
O_EXCL — гарантирует создание нового файла

Чтение

Если файл не существует или не может быть прочитан, открытие файла в режиме чтения (символ «**r**» в качестве первого символа в аргументе **mode**) завершается неудачно.

Эксклюзивный (монопольный) доступ

Открытие файла в монопольным режиме (символ «**x**» в качестве последнего символа в аргументе **mode**) завершается неудачно, если файл уже существует или не может быть создан.

В случае успеха файл создается с монопольным доступом в той степени, в которой базовая система поддерживает монопольный доступ.

Добавление

Открытие файла в режиме добавления (символ «**a**» в качестве первого символа в **mode**) приводит к принудительному выполнению всех последующих записей начиная с текущего конца файла, как если бы это делалось с помощью вызова функции **fseek()**.

Обновление

Когда файл открывается с режимом обновления (символ «**+**» в качестве второго или третьего символа в приведенном выше списке значений аргументов **mode**), в ассоциированном потоке могут выполняться как ввод, так и вывод.

За выводом не должен следовать сразу же ввод без промежуточного вызова функции **fflush** или функции позиционирования файла (**fseek**, **fsetpos**, **rewind**).

А за вводом не должен непосредственно следовать вывод без промежуточного вызова функций позиционирования файла (если только операция ввода не завершается концом файла).

В некоторых реализациях (*nix, POSIX) вместо открытия (или создания) текстового файла в режиме обновления файл может открываться (или создаваться) как двоичный поток.

При открытии поток полностью буферизуется тогда и только тогда, когда можно определить, что он не ссылается на интерактивное устройство.

Индикаторы ошибок и конца файла для открытого потока очищаются.

Функция **fopen** возвращает указатель на объект, управляющий потоком. Если операция открытия не удалась, **fopen** возвращает нулевой указатель.

freopen – открыть файл и связать его с потоком/изменить режим потока

```
#include <stdio.h>

FILE *freopen(const char * restrict filename,
              const char * restrict mode,     // режим потока
              FILE * restrict stream);
```

Функция **freopen( )** открывает файл, имя которого является строкой, на которую указывает **filename**, и связывает с ним поток, на который указывает **stream**.

Если операция открытия завершается неудачно **freopen( )** возвращает нулевой указатель. В противном случае **freopen( )** возвращает указатель на объект **FILE** для потока **stream**.

Аргумент **mode** используется так же, как в функции **fopen**.

Основное использование функции **freopen** — изменение файла, связанного со стандартными потоками **stderr**, **stdin** или **stdout**, поскольку эти идентификаторы не являются изменяемыми l-value и им не может быть присвоено значение, возвращаемое функцией **fopen**.

Если **filename** является **NULL**, функция **freopen** пытается изменить режим потока на режим, указанный в **mode**.

Какие изменения режима разрешены (если вообще разрешено изменение режима) и при каких обстоятельствах, определяется реализацией. В частности, в *nix:

- если указано **+**, режим дескриптора файла должен быть **O_RDWR**.
- если указано **r**, режим дескриптора файла должен быть **O_RDONLY** или **O_RDWR**.
- если указано **a** или **w**, режим дескриптора файла должен быть **O_WRONLY** или **O_RDWR**.

freopen( ) сначала пытается закрыть файл, связанный с указанным потоком. Невозможность закрыть файл игнорируется.

Индикаторы ошибок и конца файла для данного потока очищаются.

Пример перенаправления потока стандартного вывода в файл

```
char *stdout_arg;    // указатель на строку с именем файла перенаправления
int stdout_redirect; // флаг перенаправления (из строки параметров)
...
if (stdout_redirect) {
    fflush(stdout);
    FILE *fp = freopen(args_info.stdout_arg, "w", stdout);
    if (NULL == fp) {
        perror("Ошибка перенаправления stdout в файл");
        exit(errno);
    }
}
```

`fclose` – закрытие файла

```
#include <stdio.h>

int fclose(FILE *stream);
```

Успешный вызов функции **`fclose`** вызывает сброс потока, на который указывает **`stream`**, и закрытие соответствующего файла.

Любые незаписанные буферизованные данные для потока доставляются в хост-среду для записи в файл.

Любые непрочитанные буферизованные данные отбрасываются.

Независимо от того, успешен ли вызов, поток отсоединяется от файла, и любой буфер, установленный функцией **`setbuf`** или **`setvbuf`**, от потока отсоединяется.

Если буфер был выделен автоматически, он освобождается.

Функция **`fclose`** возвращает ноль, если поток был успешно закрыт, или **`EOF`**, если обнаружены какие-либо ошибки.

`fflush()` – сброс/очистка потока

```
#include <stdio.h>

int fflush(FILE *stream);
```

Если **`stream`** указывает на выходной поток или поток в режиме обновления, и самая последняя операция не была завершена, функция **`fflush`** выталкивает любые незаписанные данные, предназначенные для этого потока, в хост-среду для записи в файл.

Если **`stream`** является нулевым указателем, функция **`fflush`** выполняет это действие сброса для всех потоков, окрытых на запись или обновление.

Функция **`fflush`** устанавливает индикатор ошибки для потока и возвращает **`EOF`** в случае ошибки записи, в противном случае возвращает ноль.

setvbuf(), **setbuf()** – операции с буферизацией потока

```
#include <stdio.h>

int setvbuf(FILE * restrict stream, // контролируемый поток
            char * restrict buf,    // указатель на пользовательский массив
            int mode,               // способ буферизации потока
            size_t size);           // размер буферного массива

void setbuf(FILE * restrict stream,
            char * restrict buf);
```

Функция **setvbuf()** может использоваться только сразу после того, как поток, на который указывает **stream**, был связан с открытым файлом, и перед выполнением в потоке любой другой операции (кроме неудачного вызова **setvbuf**).

Аргумент **mode** определяет способ буферизации потока:

_IOFBF — полная буферизация;

_IOLBF — строковая буферизация;

_IONBF — отключить буферизацию.

Если **buf** не является нулевым указателем, массив, на который он указывает, может использоваться вместо буфера, выделяемого функцией **setvbuf**. Размер массива в данном случае указывается аргументом **size**.

В случае, когда **buf** является **NULL**, аргумент **size** определяет размер буфера, выделяемого функцией **setvbuf**.

Содержимое массива в любое время является неопределенным.

Функция **setbuf()** эквивалентна функции **setvbuf()**, вызываемой со значениями **_IOFBF** для режима и **BUFSIZ** для размера, если **buf** указан, или, если **buf** является нулевым указателем, со значением **_IONBF** для **mode**.

```
(void)setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
```

Функция **setbuf** не возвращает значения.

```
stdio.h

/* The possibilities for the third argument to `setvbuf'.  */
#define _IOFBF 0      /* Fully buffered.  */
#define _IOLBF 1      /* Line buffered.  */
#define _IONBF 2      /* No buffering.  */

/* Default buffer size.  */
#define BUFSIZ 8192
```

Операции над файлами

remove() — удаление файла

```
#include <stdio.h>

int remove(const char *filename);
```

Вызов функции удаления файла **remove** приводит к тому, что файл, чье имя является строкой, на которую указывает **filename**, более *не будет доступен по данному имени*.

Последующая попытка открыть этот файл с этим именем не удастся, если он не будет создан заново.

Если файл открыт, поведение функции удаления определяется реализацией.

Функция **remove** возвращает ноль, если операция завершается успешно, и ненулевое значение, если происходит сбой.

rename() — переименование файла

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Функция переименования приводит к тому, что файл, имя которого является строкой, на которую указывает **old**, впредь узнаваться по имени, которое дано строкой, на которую указывает **new**.

Файл больше не будет доступен под именем **old**. Если файл, имя которого является строкой, на которую указывает **new**, существует до вызова функции переименования, поведение определяется реализацией.

В случае успешного выполнения операции **rename** возвращает ноль и ненулевое значение в случае сбоя. В этом случае, если файл существовал ранее, он остается известен под своим первоначальным именем.

Среди причин сбоя функции **rename**, может быть тот факт, что файл в этот момент открыт.

tmpfile() – создание временного файла

```
#include <stdio.h>

FILE *tmpfile(void);
```

Функция **tmpfile** создает временный двоичный файл, имя которого отличается от имени любого другого существующего файла.

Временный файл будет автоматически удален, когда он будет закрыт или при завершении программы.

Если программа завершается ненормально, то, будет ли удален открытый временный файл, определяется реализацией.

Временный файл открывается для обновления в режиме «**wb+**».

Обычно существует возможность за время существования программы открыть как минимум **TMP_MAX** временных файлов (этот предел может использоваться совместно с **tmpnam()** — генерация уникальной строки в алфавите имен файла), и нет никаких ограничений на количество одновременно открытых файлов, кроме данного и ограничения на количество открытых файлов вообще.

Функция **tmpfile** возвращает указатель на поток файла, который она создала. Если файл не может быть создан, функция **tmpfile** возвращает нулевой указатель.

tmpnam() – генерация уникальной строки в алфавите имен файла

```
#include <stdio.h>

char *tmpnam(char *s);
```

Функция **tmpnam()** генерирует и возвращает строку, которая является допустимым именем файла и не совпадает ни с одним из имен существующих файлов.

Функция всякий раз, когда она вызывается, потенциально может генерировать как минимум **TMP_MAX** разных строк.

Если аргумент **s** равен **NULL**, это имя создается во внутреннем статическом буфере и может быть перезаписано при следующем вызове **tmpnam()**.

Если **s** не равно **NULL**, имя копируется в массив символов (длиной не менее **L_TMPNAM**), на который указывает **s**, и в случае успеха возвращается значение **s**.

Вызовы функции **tmpnam()** с аргументом **NULL** в качестве указателя могут привести к гонкам данных.

Следует отметить, что стандартом установлено, что ни одна библиотечная функция не должна вызывать функцию **tmpnam()**.

Если подходящая строка не может быть сгенерирована, функция **tmpnam()** возвращает нулевой указатель.

Значение макро **TMP_MAX** не может быть меньше 25.

Функции символьного ввода/вывода

`fgetc`, `fgets`, `getc`, `getchar`, `ungetc` – функции для ввода символов и строк

```
#include <stdio.h>

int   fgetc(FILE *stream); // откуда читать символ
int   getc(FILE *stream);  // возможно, реализована как макрос
int   getchar(void);       // getc(stdin)

char *fgets(char * restrict buff,    // массив для приема символов
            int             size,    // размер буфера
            FILE * restrict stream); // откуда читать

int   ungetc(int c,          // вернуть символ в поток
             FILE *stream);
```

Функция **`fgetc()`** считывает очередной символ из потока **`stream`** и возвращает **`unsigned char`** преобразованный в **`int`**, или **`EOF`** при достижении конца файла или при возникновении ошибки.

Функция **`getc()`** похожа на **`fgetc()`**, но она может быть реализована как макрос, который каждый раз вычисляет значение выражения **`stream`** поэтому ее аргумент не должен быть выражением со сторонними эффектами.

Функция **`getchar()`** эквивалентна **`getc(stdin)`**.

fgets()

считывает максимум **size-1** символов из **stream** и записывает их в буфер, на который указывает **buff**. Чтение прерывается по достижении **EOF** или символа новой строки.

Если получен символ новой строки, то он заносится в буфер.

В конец буфера за последним символом добавляется нулевой байт (**'\0'**).

Функция **fgets()** возвращает **buff** при удачном выполнении.

Если достигнут конец файла, но ничего не было прочитано, возвращается **NULL**, а содержимое массива **buff** не изменяется.

Если была ошибка чтения, возвращается **NULL**, а содержимое массива **buff** не определено.

ungetc()

заносит символ **c** обратно в **stream**, преобразуя его в **unsigned char** (если это возможно) для дальнейших операций чтения. Занесённые обратно символы будут возвращаться в обратном порядке.

Успешный промежуточный вызов (с потоком, на который указывает **stream**) функций позиционирования файла (**fseek**, **fsetpos** или **rewind**) сбрасывает любые занесенные назад в поток символы.

Внешнее хранилище, соответствующее потоку, при вызове **ungetc()** не изменяется.

Гарантируется только одно занесение символов.

Многократное занесение символов в поток без их промежуточного чтения может вызвать крах программы.

Занести EOF нельзя — функция завершится ошибкой, поток при этом не изменится.

При успешном выполнении функция **ungetc()** возвращает **c**, а при ошибке **EOF**.

Вызовы вышеприведенных функций могут смешиваться друг с другом и с другими функциями ввода из библиотеки **stdio** для того же потока ввода.

fputc, fputs, putc, putchar, puts — вывод символов и строк

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);   // возможно, реализована как макрос
int putchar(int c);              // putc(stdout)

int fputs(const char * restrict s,
          FILE * restrict stream);

int puts(const char *s);
```

fputc() записывает символ **c** в **stream**, преобразуя его в **unsigned char**.

putc() является эквивалентом **fputc()** за исключением того, что она может быть реализована в качестве макрокоманды, которая вычисляет значение выражения **stream** более одного раза.

putchar(c) является эквивалентом **putc(c, stdout)**.

fputs() записывает строку **s** в поток **stream** без добавления нулевого байта (**'\0'**).

puts() записывает строку **s** и символ новой строки в **stdout**.

Вызовы описанных здесь функций могут быть смешаны между собой и вызовами других функций вывода из библиотеки stdio в пределах одного и того же потока вывода.

Функции прямого ввода/вывода

`fread()`, `fwrite()` — ввод/вывод из двоичного потока

```
#include <stdio.h>

size_t fread(void * restrict ptr,          // расположение в памяти
             size_t size,                  // размер элемента
             size_t nmemb,                 // количество элементов
             FILE * restrict stream);      // откуда

size_t fwrite(const void * restrict ptr,   // расположение в памяти
              size_t size,                 // размер элемента
              size_t nmemb,                // количество элементов
              FILE * restrict stream);     // куда
```

Функция **`fread()`** считывает **`nmemb`** единиц данных (размер **`size`** байт каждая) из потока, на который указывает **`stream`**, и сохраняет их в расположение, на которое указывает **`ptr`**.

Функция **`fwrite()`** записывает **`nmemb`** единиц данных (размер **`size`** байт каждая) в поток, на который указывает **`stream`**, получая их из расположения, на которое указывает **`ptr`**.

При успешном выполнении функции **`fread()`** и **`fwrite()`** возвращают количество считанных или записанных единиц.

Это количество равно количеству переданных байт только, если значение `size` равно 1.

В случае ошибки или по достижении конца файла возвращается меньшее количество единиц (или ноль).

Функция **`fread()`** не отличает возникновение ошибки и факт достижения конца файла, поэтому для точного определения того, что произошло, необходимо вызывать функции **`feof(3)`** и **`ferror(3)`**.

Функции позиционирования в файле

fgetpos( ), fseek( ), fsetpos( ), ftell( ), rewind( ) — меняют положение в потоке

```
#include <stdio.h>

int  fseek(FILE *stream,
           long int offset,
           int whence);
long int ftell(FILE *stream);
void rewind(FILE *stream);
int  fgetpos(FILE * restrict stream,
             fpos_t * restrict pos);
int  fsetpos(FILE *stream,
             const fpos_t *pos);
```

Функция **fseek( )** устанавливает положение файлового индикатора в потоке **stream**.

Новое положение (в байтах) получается прибавлением **offset** байтов к положению, которое задаётся параметром **whence** («откуда»).

Если значение **whence** равно **SEEK_SET**, **SEEK_CUR** или **SEEK_END**, то смещение указывается относительно начала файла, текущего положения указателя или конца файла, соответственно.

При успешном вызове функции **fseek( )** индикатор конца файла потока очищается и отменяется влияние функции **ungetc(3)** на этот же поток (до того, как будет установлено новое положение индикатора).

После успешного завершения **fgetpos()**, **fseek()**, **fsetpos()** возвращают **0.**

В случае ошибки они возвращают **-1** и устанавливают индикатор ошибки.

Функция **ftell()** возвращает текущее значение файлового индикатора положения для потока, на который указывает **stream**. В случае ошибки она возвращают **-1** и устанавливает индикатор ошибки.

Функция **rewind()** устанавливает файловый индикатор положения для потока, на который указывает **stream**, равным началу файла. Эта функция эквивалентна вызову:

```
(void)fseek(stream, 0L, SEEK_SET)
```

за исключением того, что этом случае также сбрасывается индикатор ошибок потока.

Функции **fgetpos()** и **fsetpos()** эквивалентны **ftell()** и **fseek()** (где значение **whence** равно **SEEK_SET**). Эти функции сохраняют или устанавливают текущее значение файлового смещения в объектах типа **fpos_t**, определяемых **pos**.

В некоторых не-UNIX системах объект типа **fpos_t** может быть сложным объектом, а данные функции могут быть единственным переносимым способом изменения положения в текстовом потоке.

Функции обработки ошибок

`clearerr`, `feof`, `ferror` — проверка и сброс состояния потока

```
#include <stdio.h>

void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

`clearerr()`
очищает индикаторы конца файла и ошибки потока, указанного в **`stream`**.

`feof()`
проверяет индикатор конца файла для потока, указанного в **`stream`**, возвращая при этом ненулевое значение, если индикатор установлен.
Индикатор конца файла может быть очищен только функцией **`clearerr()`**.

`ferror()`
проверяет индикатор ошибки для потока, указанного в **`stream`**, возвращая при этом ненулевое значение, если индикатор установлен. Индикатор ошибки может быть очищен только функцией **`clearerr()`**.
Данные функции не завершаются с ошибкой и поэтому не устанавливают значение внешней переменной **`errno`**.

`perror` – печатает системное сообщение об ошибке

```
#include <stdio.h>

void perror(const char *s);
```

Когда завершается с ошибкой системный вызов, обычно возвращается **-1** и изменяется переменная **errno** для указания что пошло не так (её значения можно найти в **<errno.h>**). Многие библиотечные функции работают аналогично.

Функция **perror()** отображает номер ошибки, который установлен в **errno** на текстовое сообщение об ошибке и записывает полученную последовательность символов в стандартный поток ошибок **stderr**.

Если **s** не является нулевым указателем, или символ, на который указывает **s**, не является нулевым символом, в начале выводится строка, на которую указывает **s**, затем следует двоеточие (**:**) и шпация, после чего выводится соответствующая строка сообщения об ошибке, за которой следует символ новой строки.

Содержимое строк сообщения об ошибке такое же, как возвращаемое функцией **strerror** с аргументом **errno**.

Функция значения не возвращает.

После успешного выполнения системного вызова или библиотечной функции значение **errno** становится неопределенным — вызов может также изменить эту переменную даже при успешном выполнении, например из-за ошибки другой библиотечной функции, которая вызывалась при его работе. То есть, если после вызова функции сразу не вызывается **perror()**, значение **errno** нужно куда-нибудь сохранить.

Пример

```
FILE *ply_file; // файл, подлежащий разбору
...

long hdr_pos = ftell(ifile); // сохраним состояние сдвига в заголовке PLY
fseek(ply_file, vertex_data_shift, SEEK_SET);
size_t read = fread(vertex, 1, bytes, ply_file);
if (bytes != read) {
    perror("vertices fread request: ");
    exit(errno);
}
```