**Get next line - что она делает?**

Читает строку в файле, пока не увидит \n и “записывает” прочитанное значение в строку (то есть фун-я нужна для того, чтобы читать файл построчно).

***int get\_next\_line(int fd, char \*\*line)***

fd - файловый дескриптор

line - значение (value) (по сути в нашем случае строка), которое было прочитано

**Файловый дескриптор** — это неотрицательное целое число. Когда создается новый поток ввода-вывода, ядро возвращает процессу, создавшему поток ввода-вывода, его файловый дескриптор. *(короче, это совсем низкоуровневая штука, которая взаимодействует с ядром и процессором)*

Инфа про file descriptors:

* <http://www.bottomupcs.com/file_descriptors.xhtml>
* <https://www.computerhope.com/jargon/f/file-descriptor.htm#:~:text=A%20file%20descriptor%20is%20a,Grants%20access.>

Фун-я **open** - используется для открытия файлов, она принимает в кач-ве аргумента имя открываемого файла и флаги (O\_RDONLY - это только для чтения), (O\_WRONLY - это только для письма и т.д.), и возвращает int.

Если произошла ошибка, то вернется -1. Если файл открылся, то мы сможем его прочитать и использовать фун-ю read.

Фун-я **read** - используется для чтения файла. Принимает fd (файловый дескриптор), буфер, в который она будет читать и размер буфера (кол-во читаемых символов, т.е.байт если быть точным). Возвращает кол-во прочитанных байт, или 0 - если прочел до конца файла, -1 - при ошибке.

Например, можно читать по одному символу, тогда это так:

char c;

read(fd, &c, 1);

**GET\_NEXT\_LINE:**

три типа возвращаемых значений:

**1** - все было прочитано, без каких-либо ошибок и конец файла не был достигнут (осталось еще то, что можно читать) - тогда возвращается 1, а в line записана новая линия.

**0** - линия была прочитана, строка записана, но файл закончился (т.е.это была последняя строка)

**-1** - если произошла ошибка

*Зачем это нужно?*

Обычно мы будем использовать его в цикле, и пока он будет возвращать ненулевое значение, мы будем читать файл.

Если это буфер, то

char c[10]

read(fd, c, 10);

То есть тут будет читать по 10 символов за раз!

**ЧТО НУЖНО ЗНАТЬ, ДО ТОГО, КАК НАЧАТЬ ПИСАТЬ СВОЮ GNL:**

**Про работу указателей и почему у нас в прототипе \*\*line**

Любой тип переменных (int/char/long/double) - это как “коробка”, куда влезает переменная. И, соответственно, коробки у нас разного размера (1 байт, 4 байта и т.д.)

void f2 (\*int i) // i - address of memory (так мы говорим, на какой тип памяти указывает указатель (в нашем случае это int) и сам указатель (\*)

{

i = i \* 2;

}

Но если вызвать такую фун-ю в мэйне, то ничего с новой переменной мейна не сделается, так как у нас две разные “коробки”, в которых хранятся значения. Фун-я завершила свое действие, поменяла значение в свой “коробке”, но ничего не вернула.

Чтобы изменения вступили в силу, нужно либо сделать return значения, либо как-то ссылаться на ячейку в мейне. ***В этом сила поинтеров!***

* Взятие значения у адреса - с помощью \*, например: \*i = \*i \* 2; но это только когда мы ИСПОЛЬЗУЕМ наш созданный поинтер (мы разыменовываем).
* Взятие адреса у переменной: с помощью &

***Почему прототип gnl int get\_next\_line(int fd, char \*\*line), а не int get\_next\_line(int fd, char \*line) ???***

char line - это буква

char \*line - “строка” (или адрес первого элемента строки)

char \*\*line - “адрес строки” - чтобы все изменения, которые мы делаем в подаваемый аргумент, чтобы эти изменения произошли в main фун-и.

Например, чтобы изменения вступили в силу, нужно тогда было бы сделать так:

char \*get\_next\_line(int fd, char \*line);

а в мэйне записать:

char \*line;

line = get\_next\_line(4, line);

Вот в этом видео подробно вся эта тема разжевывается, как раз на примере gnl:

<https://youtu.be/Edf98rf3Z3s>

Или так, как у нас в прототипе - мы возвращаем int (0 / 1 / -1), но при этом в параметре мы ссылаемся на подаваемую строку в мейне, поэтому все изменения, которые будут проходить со строкой внутри фун-и тоже вступят в силу. Вот так будет выглядеть main (либо сам вызов get\_next\_line можно зациклить):

int main(void)

{

char \*line;

int fd;

fd = open("text.txt", O\_RDONLY);

get\_next\_line(fd, &line);

printf("%s\n", line);

}

**Что такое статические переменные?**

Вкратце - статическая переменная сохраняет свое значение даже при втором вызове фун-и (напоминаю, что обычно жизнь переменных ограничивается жизнью самой фун-и, и когда фун-я завершает свое действие, то все как бы “стирается” - например: был у нас каунтер 0, дошел до 10 в цикле, фун-я завершилась, мб вернула какое-то значение. Потом мы еще раз вызываем фун-ю, и в этот раз каунтер у нас опять 0. Однако если сделать его static, то он запомнит предыдущее значение и начнет считать с 10.

Еще одно свойство static переменной - ее можно не инициализировать и тогда ей автоматически присваивается значение 0, тогда она еще будет храниться в сегменте памяти компа как unitialized data (bss - сегмент).

Про другие свойства статических переменных тут: <https://www.geeksforgeeks.org/static-variables-in-c/?ref=lbp>

Подробнее про сегменты, память компьютера: <https://www.geeksforgeeks.org/memory-layout-of-c-program/>

**ПРО ЗАЩИТУ ПИРАМ**

Есть пару нюансов, с которым я столкнулась при защите gnl:

1. *Нужно ли #define буфер?*

В сабже написано, что компилиться твоя программа должна так: **gcc -Wall -Wextra -Werror -D BUFFER\_SIZE=16 \*.c**

Но я решила в .h файле все равно прописать #ifndef #define BUFFER\_SIZE [любое число] #endif. В слаке был даже жуткий срач на эту тему (к сожалению, удалилась эта прекрасная полемика, надо было скринить)). Я руководствовалась той логикой, что каждый раз прописывать размер буфера при компиляции, когда сам тестишь программу - неудобно. Но почему-то одна проверяющая прям придиралась к этому. Короче, наверное лучше убрать этот #define перед сдачей)

1. *Строчка в сабже “you should read as little as possible each time get\_next\_line is called”.*

Не знаю, как будет (было) у вас, но моя фун-я работала так, что каждый раз читалось ровно столько, сколько был размер буфера, потом “отрезал” линию и запоминал остаток. Поэтому не всегда читалось “как можно меньше”. Пир, который это заметил, вообще не придирался к этом, просто обратил мое внимание на этот момент. Если у вас также, и кто-то будет валить, то смело говорите ***should не равно must!*** (should - это в англ. рекомендация, а не обязательное выполнение условий :)

1. *Когда размер буфера >= 10 млн*

Тут с одним пиром мы долго спорили)) В общем, в сабже там просто риторический вопрос “Должна ли фун-я работать при таком огромном значении буфера?”.

Короче, если буфер статический, то будет выпадать сегфолт, так как в этом случае мы берем слишком много памяти в стеке, и получается stack overflow.

Многие буфер маллочили - но мне показалось, что это бредово маллочить буфер, поэтому не стала этого делать. Да, моя программа показывает segfault при буфере в 10 млн. (но на остальных больших значениях до все прекрасно работало). Убедить в своей правоте пира не смогла, просто договорились дать финальное слово Мулинет. Мулинет поставила ОК :)