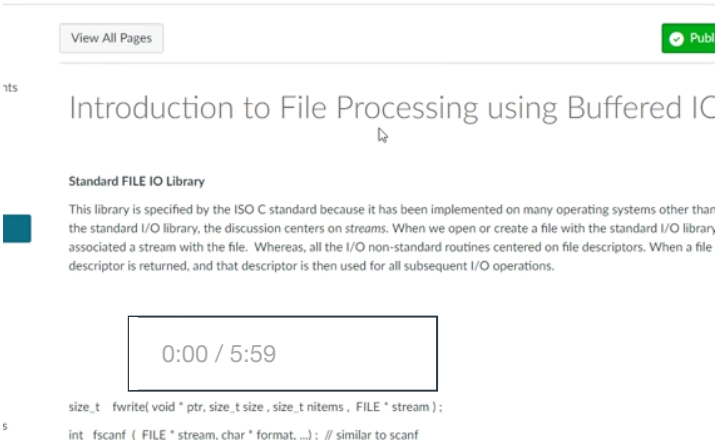
Buffered IO

Standard FILE IO Library

This library is specified by the ISO C standard because it has been implemented on many operating systems other than the UNIX System. With the standard I/O library, the discussion centers on *streams*. When we open or create a file with the standard I/O library, we say that we have associated a stream with the file. Whereas, all the I/O non-standard routines centered on file descriptors. When a file is opened, a file descriptor is returned, and that descriptor is then used for all subsequent I/O operations.

Here is the video:



The main functions that we will learn are:

Here is the list of functions that use file stream to open, read, write and close:

size_t fread(void * ptr, size_t size , size_t nitems , FILE * stream) ;

size_t fwrite(void * ptr, size_t size , size_t nitems , FILE * stream) ;

int fscanf (FILE * stream, char * format, ...) ; // similar to scanf

int fprintf (FILE * stream, char * format, ...) ; // similar to printf

char * fgets (char * str, int size, FILE * stream) ;

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

int sscanf (char * s, char *format, ...) ; // input is a buffer, not from file nor stdin

int sprintf (char * str, char * format, ...) ; // output to a buffer, not to a file nor to stdout

When we open a stream, the standard I/O function fopen returns a pointer of type FILE. This is a structure that contains all the information required by the standard I/O library to manage the stream.

To reference the stream, we pass its FILE pointer as an argument to each standard I/O function. We'll refer to a pointer to a FILE object, the type FILE *, as a *file pointer*.

Additionaly, when a program is launched, three streams are predefined and automatically available:: stdin, stdout, and stderr. The file pointers are defined in the <stdio.h> header.

The unistd.h header also defines similar file descriptors as : STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO.

Some facts about standard buffered IO functions in C are :

- C views each file simply as a sequential stream of bytes
- Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained, administrative data structure.
- When a file is opened, a stream is associated with it.
- Three files and their associated streams are automatically opened when program execution begins—the standard input, the standard output and the standard error. The standard input, standard output and standard error are manipulated using file pointers stdin, stdout and stderr.
- stdin is a FILE pointer defined in /usr/include/stdio.h , it is a data going into the program (usually from a device such as keyboard from the user). In our case scanf ("%d", &count) and scanf (stdin, "%d", &count) are same statements.

  stdout is a FILE pointer defined in /usr/include/stdio.h , it is a data going out from the program (usually to the display).
- For us printf ("%d \n", count) and printf (stdout, "%d \n", count) are same statements.

stderr is a FILE pointer , it is error whose default output is stdout, but could be redirected to the file or elsewhere.

- Streams provide communication channels between files and programs.
- For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables a program to print data on the screen.
- Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information used to process the file.
- This structure includes a file descriptor, i.e., an index into an operating system array called the open file table.
- Each array element in this open file table, contains a file control block (FCB) that the operating system uses to administer a particular file.
- The standard library provides many functions for reading data from files and for writing data to files.
  - Function fgetc, like getchar, reads one character from a file. This function receives a FILE pointer as an argument for the file from which a character will be read.
  - The call fgetc(stdin) reads one character from stdin—the standard input. This call is equivalent to the call getchar().
  -
  - Function fputc, like putchar, writes one character to a file.
  - Function fputc receives as arguments a character to be written and a pointer for the file to which the character will be written.
  - The function call fputc('a', stdout) writes the character 'a' to stdout—the standard output. This call is equivalent to putchar('a').
  - Several other functions used to read data from standard input and write data to standard output have similarly named file-processing functions. The fgets and fputs functions, for example, can be used to *read a line from a file* and *write a line to a file*, respectively.
- We will also talk about the file-processing equivalents of functions scanf and printf—fscanf and fprintf.
- C imposes no structure on a file. Thus, notions such as a record of a file do not exist as part of the C language.
-

Standard IO Library functions

Reading and Writing a Stream

Once we open a stream, we can choose from among three types of unformatted I/O:

1. Character-at-a-time I/O. We can read or write one character at a time, with the standard I/O functions handling all the buffering, if the stream is buffered.
2. Line-at-a-time I/O. If we want to read or write a line at a time, we use fgets and fputs. Each line is terminated with a newline character, and we have to specify the maximum line length that we can handle when we call fgets.
3. Direct I/O. This type of I/O is supported by the fread and fwrite functions. For each I/O operation, we read or write some number of objects, where each object is of a specified size. These two functions are often used for binary files where we read or write a structure with each operation.

Input Functions to read Character-at-a-time

Three functions allow us to read one character at a time.

#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);

All three return: next character if OK, EOF on end of file or error

The function getchar is defined to be equivalent to getc(stdin). Getc is implemented as a macro while fgetc is a function.

These three functions return the next character as an unsigned char converted to an int. The reason for specifying unsigned is so that the high-order bit, if set, doesn't cause the return value to be negative. The reason for requiring an integer return value is so that all possible character values can be returned, along with an indication that either an error occurred or the end of file has been encountered. The constant EOF in <stdio.h> is required to be a negative value. Its value is often -1.

This representation also means that we cannot store the return value from these three functions in a character variable and later compare this value with the constant EOF. Note that these functions return the same value whether an error occurs or the end of file is reached. To distinguish between the two, we must call either ferror or feof.

#include <stdio.h>

int ferror(FILE *_fp_);

int feof(FILE *_fp_);

Both return: nonzero (true) if condition is true, 0 (false) otherwise

Output Functions to write Character-at-a-time:

Output functions are available that correspond to each of the input functions we've already described.

#include <stdio.h>

int putc(int _c_, FILE *_fp_);

int fputc(int _c_, FILE *_fp_);

int putchar(int _c_);

All three return: _c_ if OK, EOF on error

As with the input functions, putchar(c) is equivalent to putc(c, stdout), and

putc can be implemented as a macro, whereas fputc cannot be implemented as a

macro.

Read Line-at-a-Time I/O

Line-at-a-time input is provided by the two functions, fgets and gets.

#include <stdio.h>

char *fgets(char *_buf_, int _n_, FILE * _fp_);

char *gets(char *_buf_);

Both return: buf if OK, NULL on end of file or error

Both specify the address of the buffer to read the line into. The gets function reads from standard input, whereas fgets reads from the specified stream. With fgets, we have to specify the size of the buffer, *n*. This function reads up through and including the next newline, but no more than *n * 1* characters, into the buffer. The buffer is terminated with a null byte. If the line, including the terminating newline, is longer than *n * 1*, only a partial line is returned, but the buffer is always null terminated. Another call to fgets will read what follows on the line. The gets function should never be used. The problem is that it doesn't allow the caller to specify the buffer size.

Even though ISO C requires an implementation to provide gets, you should use fgets instead.

Write Line-at-a-time is done using fputs and puts.

int fputs(const char *_str_, FILE *_fp_);

int puts(const char *_str_);

Both return: non-negative value if OK, EOF on error

(https://sierra.instructure.com/courses/322669/pages/sscanf-sprintf?module_item_id=6013138)

The function fputs writes the null-terminated string to the specified stream. The null byte at the end is not written. Note that this need not be line-at-a-time output, since the string need not contain a newline as the last non-null character. Usually, this is the case — the last non-null character is a newline—but it's not required. The puts function writes the null-terminated string to the standard output, without writing the null byte. But puts then writes a newline character to the standard output.

The puts function is not unsafe, like its counterpart gets. Nevertheless, we'll avoid using it, to prevent having to remember whether it appends a newline. If we always use fgets and fputs, we know that we always have to deal with the newline character at the end of each line.

Binary I/O

If we're doing binary I/O, we often would like to read or write an entire structure at a time. To do this using getc or putc, we have to loop through the entire structure, one byte at a time, reading or writing each byte. We can't use the line-at-a-time functions, since fputs stops writing when it hits a null byte, and there might be null bytes within the structure. Similarly, fgets won't

work correctly on input if any of the data bytes are nulls or newlines. Therefore, the following two functions are provided for binary I/O.

```
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite (const void *ptr, size_t size, size_t nobj,FILE *restrict fp);
```

Both return: number of objects read or written

These functions have two common uses:

1. Read or write a binary array. For example, to write elements 2 through 5 of a floating-point array, we could write

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
err_sys("fwrite error");
```

Here, we specify *size* as the size of each element of the array and *nobj* as the number of elements.

2. Read or write a structure. For example, we could write

```
struct {
    unsigned char age ;
    char name[NAMESIZE];
} person  ;
if ( fwrite(&person  , sizeof(person  ),  1 , fp )  != 1 )
     printf ("fwrite error \n");
```

Here, we specify *size* as the size of structure and *nobj* as 1 (the number of objects to write).

Both fread and fwrite return the number of objects read or written. For the read case, this number can be less than *nobj* if an error occurs or if the end of file is

encountered. In this situation, ferror or feof must be called. For the write case, if the return value is less than the requested *nobj*, an error has occurred.

Function file FOPEN AND MODES TO OPEN, and FCLOSE

Function fopen

Standard I/O routines do not operate directly on file descriptors. Instead, they use their own unique identifier, known as the file pointer. Inside the C library, the file pointer maps to a file descriptor. The file pointer is represented by a pointer to the FILE typedef, which is defined in .

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

The syntax is:

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

In Standard IO library, a FILE pointer is returned. This pointer is a representative of a stream. We say a stream is attached interfacing the user program and the actual file.

The actual path to the structure definition of FILE is : /usr/include/libio.h

The actual structure

```
struct _IO_FILE {


};
```

Streams can : input, output and/or both

Modes for reading, writing and appending :

- Files may be opened in one of several modes .
- To create a file, or to discard the contents of a file before writing data, open the file for writing using mode set to "w"
- To read an existing file, open it for reading ("r").
- To add records to the end of an existing file, open the file for appending with mode set to "a"
- To open a file so that it may be written to and read from, open the file for updating in one of the three update modes—"r+", "w+" or "a+".
- Mode "r+" opens an existing file for reading and writing.
- Mode "w+" creates a file for reading and writing.
- If the file already exists, it's opened and its current contents are discarded.
- Mode "a+" opens a file for reading and writing—all writing is done at the end of the file.
- If the file does not exist, it's created.
- Each file open mode has a corresponding binary mode (containing the letter b) for manipulating binary files.

Upon success, fopen( ) returns a valid FILE pointer. On failure, it returns NULL, and sets errno appropriately.

Function FCLOSE:

The fclose( ) function closes a given stream:

int fclose (FILE *stream);

Any buffered and not-yet-written data is first flushed. On success, fclose( ) returns 0. On failure, it returns EOF and sets errno appropriately.

Unbuffered IO

Unbuffered File IO

There are functions available for file I/O—open a file, read a file, write a file, and so on. Most file I/O on a UNIX system can be performed using only five functions: open, read, write, lseek, and close. These functions are referred to as unbuffered IO. The term *unbuffered* means that each read or write invokes a system call in the kernel. We talked about the system call steps before.

NOTE: These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1.

Systemwide, kernel has a data structure that stores list of open file descriptors mapping to a file name. All open files are referred to by file descriptors. A file descriptor is a non-negative integer that the kernel uses to identify the files accessed by a process. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by open as an argument to either read or write.

NOTE: Remember , in buffered IO, we refer open files as streams using FILE data type. Whereas in unbuffered IO, we use file descriptors.

UNIX System shells associate file descriptor 0 (STDIN_FILENO) with the standard input of a process, file descriptor 1 (STDOUT_FILENO) with the standard output, and file descriptor 2 (STDERR_FILENO) with the standard error. This convention is used by the shells and many applications, many applications would break if these associations weren't followed.

File Descriptor	**Purpose**	**POSIX Name**	**Stdio stream**
0	standard input	STDIN_FILENO	**stdin**

1	standard output	STDOUT_FILENO **stdout**
2	standard error	STDERR_FILENO **stderr**

These constants are defined in the <unistd.h> header.

The following are the five key system calls for performing file I/O. You have to include this include file

#include <unistd.h>

The five system calls we read are:

- **open**
- **lseek**
- **write**
- **read**
- **close**

OPEN

The open() system call either opens an existing file or creates and opens a new file.

#include <sys/stat.h>

#include <fcntl.h>

int open(const char *_path_, int _oflag_, ... /* mode_t _mode_ */);

1. The *path* parameter is the name of the file to open or create.

2. The oflag argument takes various values, we can bitwise OR them together, notable and frequently used are

- O_RDONLY Open for reading only.
- O_WRONLY Open for writing only.
- O_RDWR Open for reading and writing.
- O_APPEND Append to the end of file on each write
- O_CREAT Create the file if it doesn't exist.
- O_EXCL Generate an error if O_CREAT is also specified and the file already exists.
- O_TRUNC If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0

3. The optional last argument is shown as ..., because the number and types of the remaining arguments may vary. The last argument is used only when a new file is being created. If you look at our man pages, it states "mode specifies the permissions to use in case a new file is created. This argument must be supplied when O_CREAT is specified in flags; if O_CREAT is not specified, then mode is ignored."

Some common values of mode are

`S_IRWXU -` This is equivalent to '(S_IRUSR | S_IWUSR | S_IXUSR)'.

`S_IRGRP -` Read permission bit for the group owner of the file. Usually 040.

`S_IWGRP -` Write permission bit for the group owner of the file. Usually 020.

`S_IXGRP -` Execute or search permission bit for the group owner of the file. Usually 010.

`S_IRWXG -` This is equivalent to '(S_IRGRP | S_IWGRP | S_IXGRP)'.

On RETURN: On success, open() returns a file descriptor that is used to refer to the file in subsequent system calls. If an error occurs, open() returns –1 and errno is set accordingly.

Some Errors from open() function

If an error occurs while trying to open the file, open() returns –1, and errno identifies the cause of the error. The following are some possible errors that can occur (notable being EACCES):

EACCES The file permissions don't allow the calling process to

	open the file in the mode specified by flags. Alternatively, because of directory permissions, the file could not be accessed, or the file did not exist and could not be created.
EISDIR	The specified file is a directory, and the caller attempted to open it for writing. This isn't allowed. (On the other hand, there are occasions when it can be useful to open a directory for reading)
EMFILE	The process resource limit on the number of open file descriptors has been reached
ENFILE	The system-wide limit on the number of open files has been reached
ENOENT	The specified file doesn't exist, and O_CREAT was not specified, or O_CREAT was specified, and one of the directories in pathname doesn't exist or is a symbolic link pointing to a nonexistent pathname (a dangling link).

lseek Function

For each open file, the kernel records a file offset, sometimes also called the read/write offset or pointer. This is the location in the file at which the next read() or write() will commence. The file offset is expressed as an ordinal byte position relative to the start of the file. The first byte of the file is at offset 0. The file offset is set to point to the start of the file when the file is opened and is automatically adjusted by each subsequent call to read() or write() so that it points to the next byte of the file after the byte(s) just read or written. Thus, successive read() and write() calls progress sequentially through a file. The lseek() system call adjusts the file offset of the open file referred to by the file descriptor fd, according to the values specified in offset and whence. By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified.

An open file's offset can be set explicitly by calling lseek.

off_t lseek(int *fd*, off_t *offset*, int *whence*) ;

Returns new file offset if successful, or –1 on error

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is SEEK_SET, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is SEEK_CUR, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative. In other words, relative to the current position.
- If *whence* is SEEK_END, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Because a successful call to lseek returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

off_t currentPosition = currpos = lseek (fd, 0, SEEK_CUR);

Here are some other examples of lseek() calls, along with comments indicating

where the file offset is moved to:

lseek (fd, 0, SEEK_SET) ; /* Start of file */

lseek (fd, 0, SEEK_END) ; /* Next byte after the end of the file */

lseek (fd, -1, SEEK_END) ; /* Last byte of file */

lseek (fd, -10, SEEK_CUR) ; /* Ten bytes prior to current location */

lseek (fd, 1000, SEEK_END) ; /* 1001 bytes past last byte of file causing holes */

Write Function

Data is written to an open file with the write function.

ssize_t write (int *fd*, const void **buffer*, size_t *nbytes*) ;

Returns: number of bytes written if OK, 1 on error

The arguments to write() are similar to those for read(): buffer is the address of the data to be written; nbytes is the number of bytes to write from buffer; and fd is a file descriptor referring to the file to which data is to be written.

For a regular file, the write operation starts at the file's current offset. If the O_APPEND option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

```
char address[32] = "1600 Washington Blvd" ;

   int fw = open ( "write_out", O_WRONLY | O_CREAT, S_IRWXU );
   if( fw < 0 )
   {
      perror ( "ERROR in OPENING FILE ");
      exit (1) ;
   }

   lseek ( fw, 0, SEEK_SET); // set the pointer at the start of the file
   write ( fw, address, strlen ( address) );

   close (fw);
```

read Function

The read() system call reads data from the open file referred to by the descriptor fd.

ssize_t read (int *fd*, void **buffer*, size_t *nbytes*) ;

Returns number of bytes read, 0 on EOF, or –1 on error

The *nbytes* argument specifies the maximum number of bytes to read. (The size_t data type is an unsigned integer type.) The buffer argument supplies the address of the memory buffer into which the input data is to be placed. This buffer must be at least count bytes long.

Note: These IO calls don't allocate memory for buffers that are used to return information to the caller. Instead, we must have buffer already allocated and must pass a pointer to this allocated memory buffer and correct size.

A successful call to read() returns the number of bytes actually read, or 0 if end-offile is encountered. On error, the usual –1 is returned. The ssize_t data type is a signed integer type used to hold a byte count or a –1 error indication.

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time.
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
- When interrupted by a signal and a partial amount of data has already been read.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

Closing a File: close()

The close() system call closes an open file descriptor, freeing it for subsequent reuse by the process. When a process terminates, all of its open file descriptors are automatically closed.

int **close**(int fd) ;

Returns 0 on success, or –1 on error

Closing a file also releases any record locks that the process may have on the file. When a process terminates, all of its open files are closed automatically by the kernel. Many programs take advantage of this fact and don't explicitly close open files. However, It is usually good practice to close unneeded file descriptors explicitly, since this makes our code more readable and reliable in the face of

subsequent modifications. Furthermore, file descriptors are a consumable resource, so failure to close a file descriptor could result in a process running out of descriptors. This is a particularly important issue when writing long-lived programs that deal with multiple

files, such as shells or network servers. Just like every other system call, a call to close() should be bracketed with errorchecking code, such as the following:

```
if ( close ( fd ) == -1 )
   exit (  ) ;
```

buffered and unbuffered

What is buffered vs unbuffered ?

The output from functions like printf , fprintf will not be instant. The actual output will be buffered in the kernel. When the buffer is full or when you have \n character, the system would write it out to the output using the write function.

But when you the function write , the actual output will not be buffered in the kernel, the system call will be called upon to output to the stdout.

Messages that you would write to the stderr will always be write and there is no buffer involved. In our case, stderr is piped to the stdout. So, any stderr messages such as "segmentation fault (core dumped)" exception messages will be instant and there won't be any delay in outputting the data.

Now how do we prove that printf uses buffer and write doesn't ?

So we will write a program

Buffered Program :

```
main  ( )
{
 printf ( "Hello World" )  ;  // buffered output, there is no \n
 int *ptr ;
*ptr = 20;  // yields segmentation fault
}
```

In the above program, we are trying to print the string "Hello World" first followed by *ptr = 20 . Because printf didn' have \n character, it buffers data. Before it could write it out to stdout, it executes the *ptr = 20 statement. But this *ptr = 20 is illegal, it crashes the program. When it crashes, the exception message "segementation fault (core dumped) " takes precedence because it is exception and it gets piped to stderr which in turn piped to stdout. You would see this message , not the Hello World message which is still sitting in kernel somewhere.

Unbuffered Output :

In this program, we use write function instead of printf function. See the following program:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main (  )
{
   write (STDIN_FILENO, "Hello World", strlen ("hello world") ) ; // unbuffered
   int *ptr ;
  *ptr = 10;  // segmentation fault
}
```

If you run this program, you will see Hello World will be written first followed by segmentation fault core dumped message. That shows write is unbuffered.

Now which is efficient ?

In case you are reading one char by char and using unbuffered functions, it will be inefficient because you are interrupting the CPU and unnecessarily clogging the IO. Unbuffered function are efficient when you want to write or read vast amount of data into your buffer. But then, it is beyond the scope of our course to run experiments to prove one way or other.

Return values of system function: read

ssize_t read (int fd, void *buf, size_t len);

On success, the number of bytes written into buf is returned.

On error, the call returns -1, and errno is set.

The file position is advanced by the number of bytes read from fd.

read function may return a positive non-zero value less than len . When only less than len bytes are available , system call may have been interrupted by a signal, pipe may have broken.

When read returns with value 0 to indicate end of file, no bytes are read. EOF is not error. It simply means there are no bytes to read past the EOF. When you are read data from a socket, then read trying to read len bytes , but there are no data to read, it then get blocked until there is a data.

In the case of EOF, there is no data at all.

In the case of blocking, there is a wait for the data to become available.

When you get -1, you evaluate the error and probably re-read from the descriptor

The call to read can return with many possibilities:

- if it is -1 and errno is set to EINTR, indicates signal interrupted from reading.
- it it is zero, EOF reached.
- if the call returns value less than len, but greater than 0, there are bytes read into the buffer.
- if the call returned a value equal to len, all requested bytes have been read.

When you want to read from a socket and don't want to get blocked when there are no data is available, you issue non-blocking call via the open function. Then read will return with a value -1 and error is to EAGAIN.