University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

I/O in Unix and C

File Descriptors Used for All I/O in Unix and C

Unix (and **C**) supports
- a **unified notion of I/O**
- known as **file descriptors**.

Programs can use file descriptors to…
- read from the keyboard,
- write to the display (virtual or physical),
- read and write files,
- communicate with devices (such as printers),
- communicate over network connections, and
- communicate with other programs.

Programs Can Be Oblivious to "Type" of File Descriptor

For the most part,
- **programs do not need to know**
- **what "kind" of communication happens** with a file descriptor.

For example,
- most of the original Internet services
- were written and debugged using keyboard and display
- then simply attached to Internet connections*
- without modifying the programs.

*`inetd` accepted the incoming network connections and launched programs with a network connection replacing the keyboard and display.

File Descriptors are Small Integers

How does it work?

Remember that **access to any device is** usually a **privileged** operation.

OS maintains information
- **about each I/O channel.**
- For a given program,
- the information is kept **in an array**.

What's a file descriptor?

An **index into the array—a small integer**.

OS Implementation Has Evolved Over Time

Originally,
- the **array** of I/O channels in the OS
- **had fixed size**.

Today, most **OS's grow** the array **dynamically**.

But the **mechanism on the user side (programs) is the same**:
- an I/O channel is a small integer
- called a file descriptor.

Most I/O in C Uses Streams

In **C**,
- **most I/O uses an additional abstraction**
- built on top of file descriptors.

Streams provide
- a **continuous sequence of bytes**
- typically including some kind of **buffering**.

Buffering Happens for Both Reading and Writing

Buffering means
- **waiting until** a certain **amount or type of data is available** before sending anything, or
- **reading extra data** in anticipation of future requests for data.

For example,
- **when you type** at the keyboard
- **data** are **not** usually **delivered** to a program
- **until you press <Enter>**.
- That way, programs do not need to implement <Backspace>.

OS and Stream Both Buffer on Reads

Examples of buffering when reading…

Data on disk come in 4kB or 8kB blocks,
- and access time can be ~**10 msec**,
- so the **OS does not read 1B** from a file
- even if your program requests **1B**.

Similarly, making a **system call**
- is **too expensive** just **to obtain 1B**, so
- **streams read more and buffer** the rest
- **until your program asks for more**.

A C Stream is a **FILE***

Where does buffering happen with a stream in C?

In a data structure, of course!

A **stream in C is**
- a pointer to that structure,
- with **type FILE*** (all caps).

The structure itself is not usually used.*

*You can find it: compile with –E to get preprocessed source, then go back to the header file to see the comments, but don't expect it to be easy to understand—it's meant to be used by most programmers, not modified.

Three Streams by Default, but Can Create More

A **C program has three default streams** corresponding to three default file descriptors:

stdin	(descriptor 0)	keyboard
stdout	(descriptor 1)	display (normal)
stderr	(descriptor 2)	display (error)

You can **create other stream variables by writing**, for example:

```
FILE* my_file;
```

Descriptors Can Be Overridden When a Program Starts

Why are normal and error output distinct?

Remember that
- **you can override each descriptor separately**, so, for example,
- you can run a program and
- save its normal output to a file
- but deliver error output to the display
- so that you notice the errors.

Open a File on Disk as a Stream Using **fopen**

What if a program wants to open a file?

Use this function:

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

- **path is the file name** (starting from the program's current working directory).
- **mode specifies** whether the file is opened for **reading, writing, or both** (see next slide).
- **returns** a **new stream** on success, **or NULL** on failure.

Mode is Passed as a String

What are the possible modes?

"r" or **"rb"***	**read only**
"w" or **"wb"**	**write only (after deleting)**
"a" or **"ab"**	**append (write only, at end)**
"r+"/"r+b"/"rb+"	**open r/w** (read/write)
"w+"/"w+b"/"wb+"	**truncate, then r/w**
"a+"/"a+b"/"ab+"	**append r/w**

*The "b" is antiquated notation meaning "binary." On some systems, such as MS-DOS, files opened in non-binary mode changed certain bytes (CR/LF) read/written to the file.

Close a Stream Using `fclose`

What about when one is done with a stream?

Use this function:

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

- **`stream` is the stream** to close.
- **returns 0 on success**, or **`EOF` (-1) on failure**.

Do NOT leave streams unclosed; the number allowed for a program is finite!

Discuss Five Kinds of I/O

We'll talk about five kinds of I/O:

1. one character at a time,
2. reading and writing strings,
3. formatted I/O,
4. binary I/O, and
5. formatted "I/O" to/from strings.

Use `fgetc` or `getc` to Read a Character from a Stream

Let's start with one character at a time.

Remember that streams are buffered.

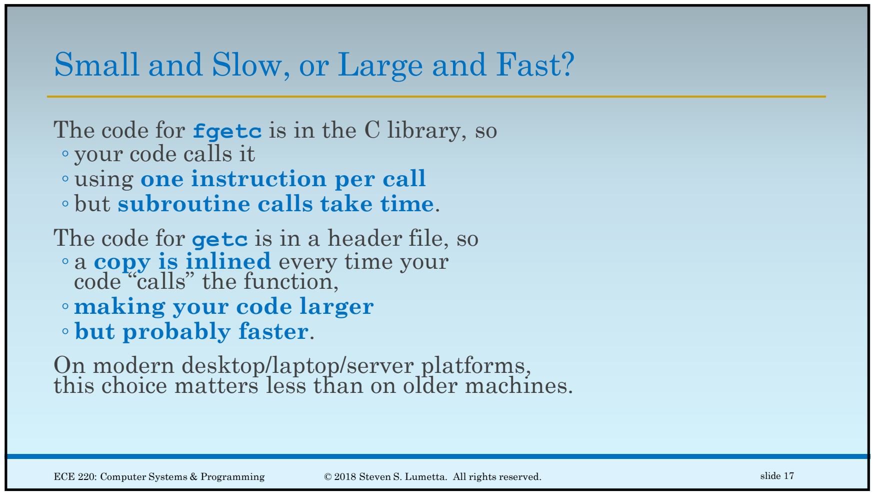
To read a character, use

```
int fgetc (FILE* stream);
int getc (FILE* stream);
```

`fgetc` is a library function.

`getc` is a preprocessor macro.

What's the difference?

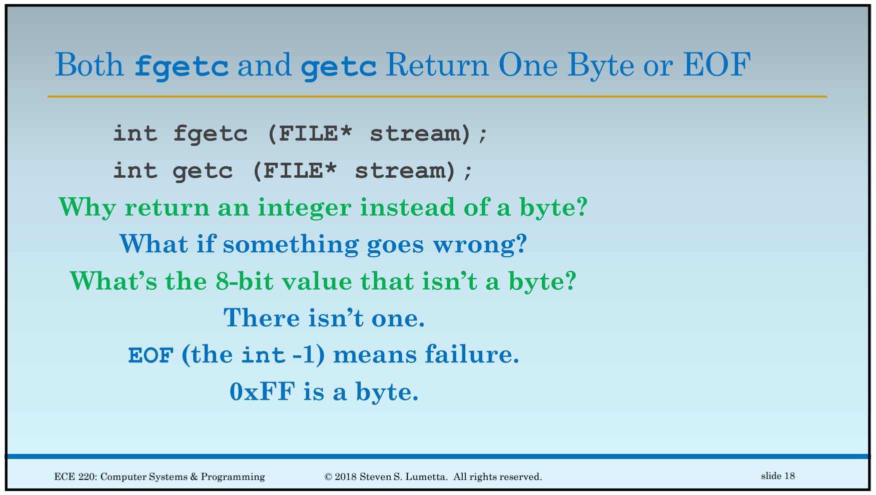


Small and Slow, or Large and Fast?

The code for **fgetc** is in the C library, so
- your code calls it
- using **one instruction per call**
- but **subroutine calls take time**.

The code for **getc** is in a header file, so
- a **copy is inlined** every time your code "calls" the function,
- **making your code larger**
- **but probably faster**.

On modern desktop/laptop/server platforms, this choice matters less than on older machines.



Both **fgetc** and **getc** Return One Byte or EOF

```
int fgetc (FILE* stream);
int getc (FILE* stream);
```

Why return an integer instead of a byte?

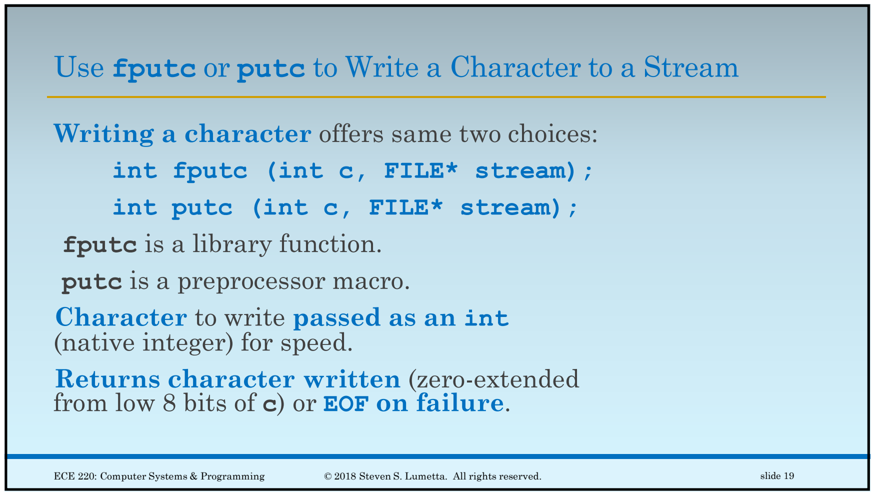
What if something goes wrong?

What's the 8-bit value that isn't a byte?

There isn't one.

EOF (the int -1) means failure.

0xFF is a byte.



Use **fputc** or **putc** to Write a Character to a Stream

Writing a character offers same two choices:

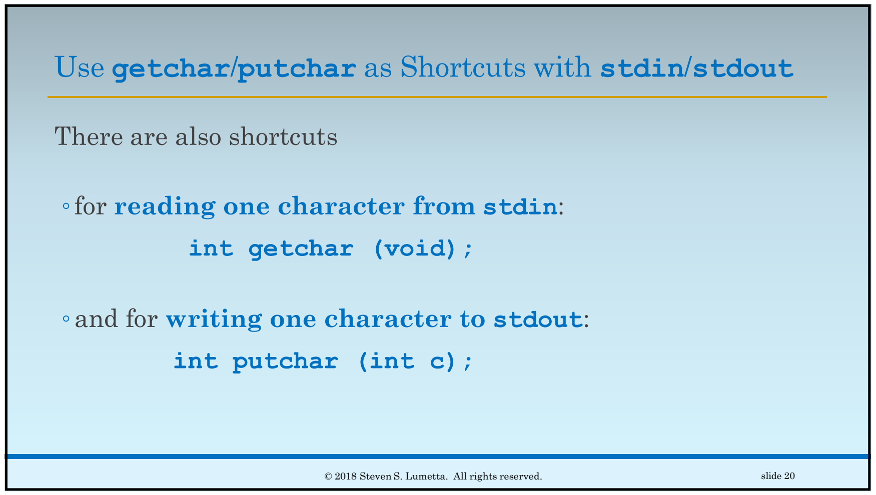
```
int fputc (int c, FILE* stream);
int putc (int c, FILE* stream);
```

fputc is a library function.

putc is a preprocessor macro.

Character to write **passed as an int** (native integer) for speed.

Returns character written (zero-extended from low 8 bits of **c**) or **EOF on failure**.



Use **getchar/putchar** as Shortcuts with **stdin/stdout**

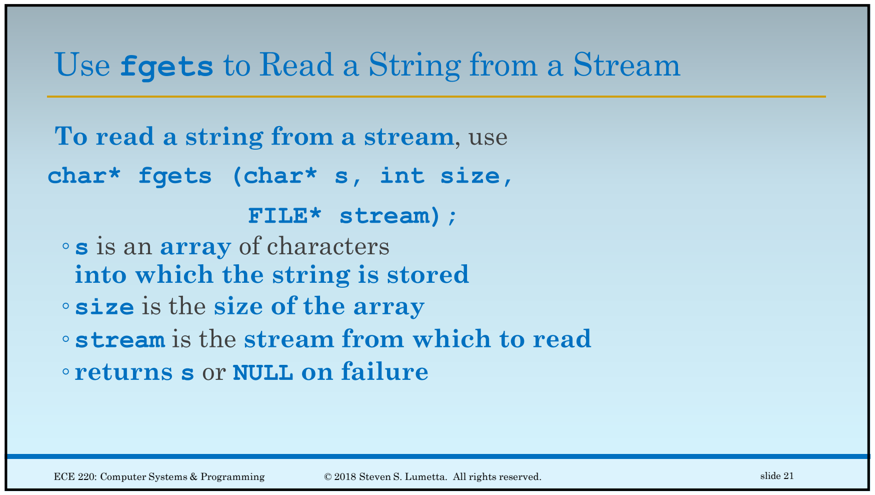
There are also shortcuts

- for **reading one character from stdin**:

```
int getchar (void);
```

- and for **writing one character to stdout**:

```
int putchar (int c);
```

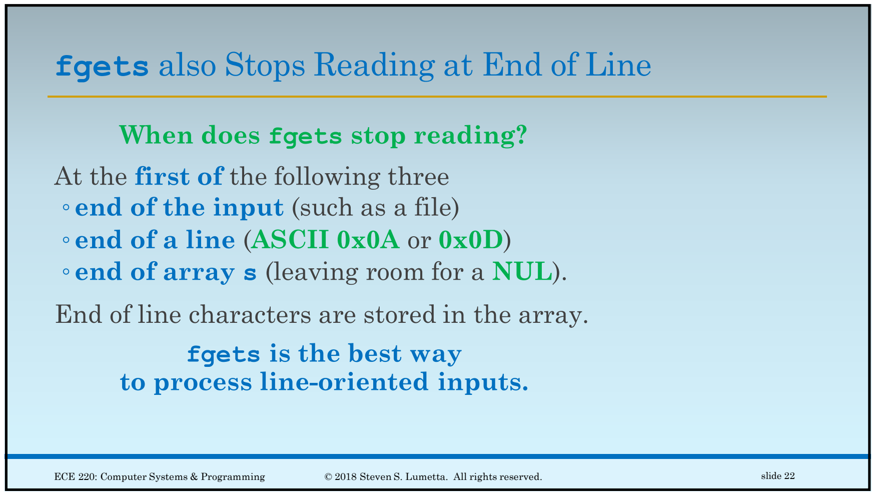


Use **fgets** to Read a String from a Stream

To read a string from a stream, use

```
char* fgets (char* s, int size,
                  FILE* stream);
```

- **s** is an **array** of characters **into which the string is stored**
- **size** is the **size of the array**
- **stream** is the **stream from which to read**
- **returns s** or **NULL on failure**



fgets also Stops Reading at End of Line

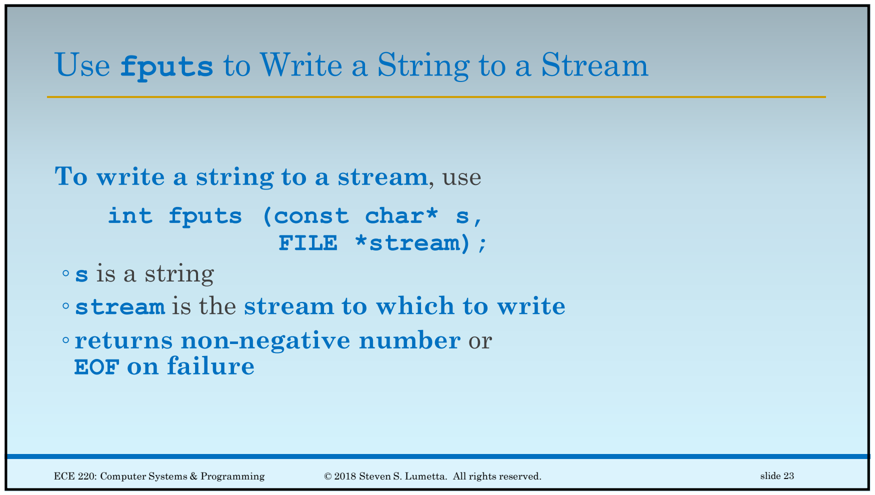
When does fgets stop reading?

At the **first of** the following three

- **end of the input** (such as a file)
- **end of a line** (**ASCII 0x0A** or **0x0D**)
- **end of array s** (leaving room for a **NUL**).

End of line characters are stored in the array.

fgets is the best way to process line-oriented inputs.

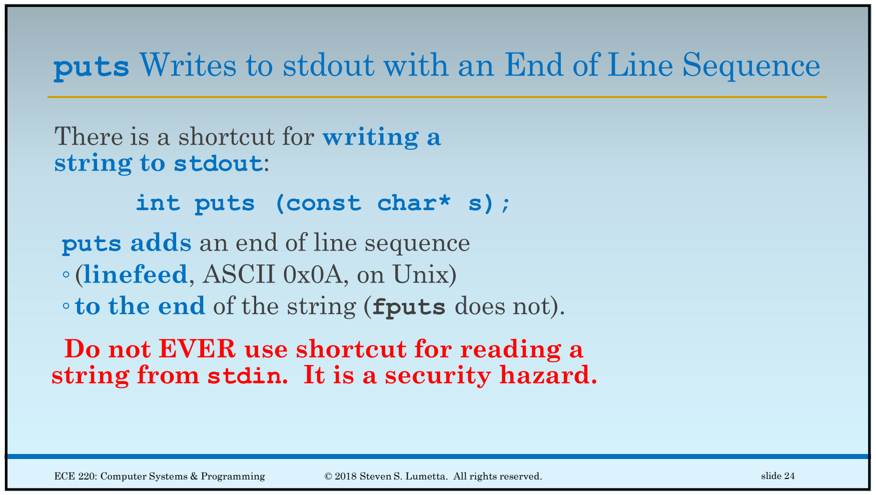


Use **fputs** to Write a String to a Stream

To write a string to a stream, use

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

- **s** is a string
- **stream** is the **stream to which to write**
- **returns non-negative number** or **EOF on failure**



puts Writes to stdout with an End of Line Sequence

There is a shortcut for **writing a string to stdout**:

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

puts adds an end of line sequence

- (**linefeed**, ASCII 0x0A, on Unix)
- **to the end** of the string (**fputs** does not).

Do not EVER use shortcut for reading a string from stdin. It is a security hazard.

Use `scanf/printf` for Formatted I/O with `stdin/stdout`

You already know the shortcut versions for formatted I/O:

`int scanf (const char* format, …);`

reads formatted input from stdin.

`int printf (const char* format, …);`

writes formatted output to stdout.

Use `fscanf` to Read Formatted Input from a Stream

To read formatted input from a stream:

```
int fscanf (FILE* stream,
     const char* format, ...);
```

- `stream` is the **stream from which to read**
- `format` is the **format specifier**
- remaining arguments are as with `scanf`
- **returns number of conversions** or **-1 on failure**

Use `fprintf` to Write Formatted Output to a Stream

To write formatted output to a stream:

```
int fprintf (FILE* stream,
     const char* format, ...);
```

- `stream` is the **stream to which to write**
- `format` is the **format specifier**
- remaining arguments are as with `printf`
- **returns number of characters printed** or **negative number on failure**

Should Data be Stored in ASCII, or in Binary?

What's better…

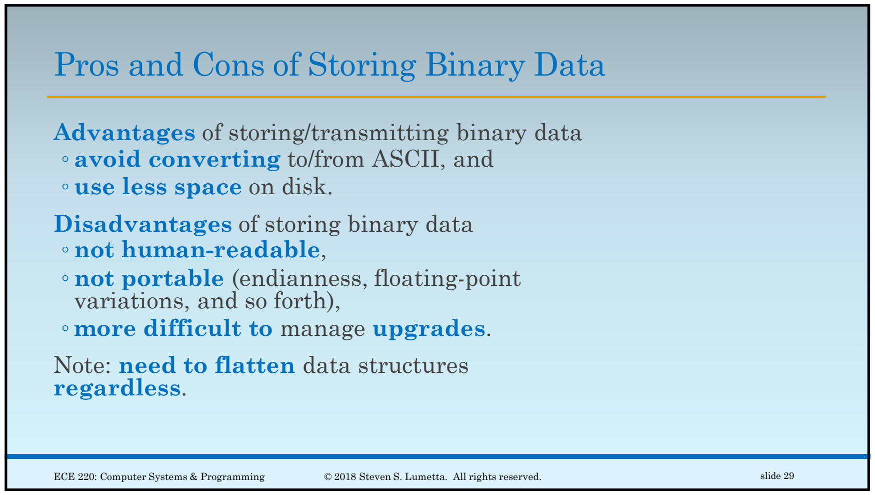
data stored as ASCII

```
N_vertices=44164
START Vertex<0>
X=293540
Y=454970
Deg=2
Neighbor0=23
Neighbor1=44142
END Vertex<0>
…
```

OR

data stored as bits

```
84 AC 00 00
A4 7A 04 00
3A F1 06 00
02 00 00 00
17 00 00 00
6E AC 00 00
```

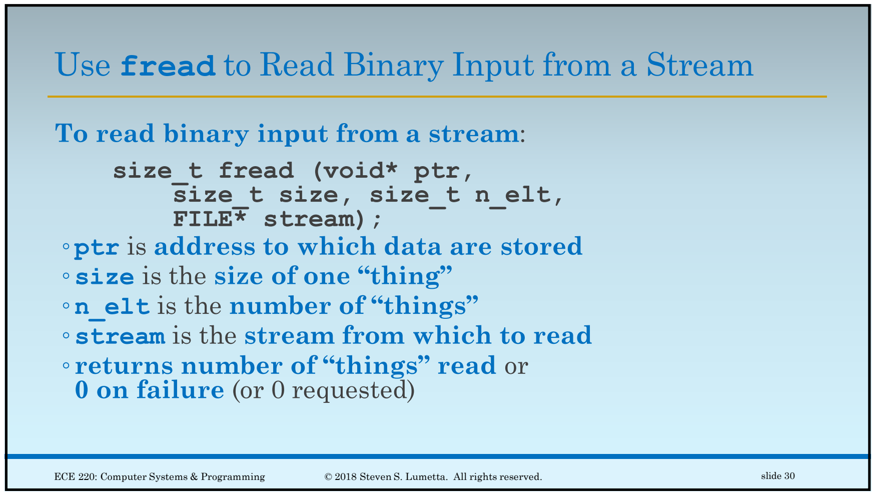


Pros and Cons of Storing Binary Data

Advantages of storing/transmitting binary data
- **avoid converting** to/from ASCII, and
- **use less space** on disk.

Disadvantages of storing binary data
- **not human-readable**,
- **not portable** (endianness, floating-point variations, and so forth),
- **more difficult to** manage **upgrades**.

Note: **need to flatten** data structures **regardless**.

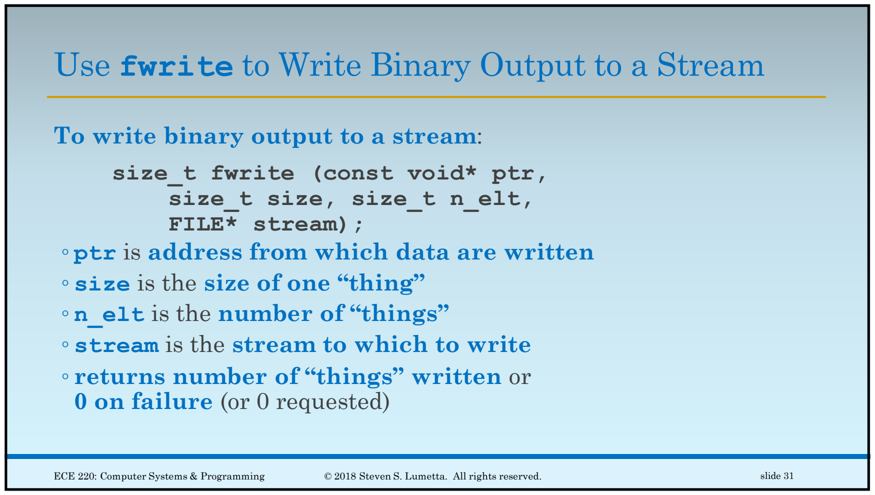


Use **fread** to Read Binary Input from a Stream

To read binary input from a stream:

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

- **ptr** is **address to which data are stored**
- **size** is the **size of one "thing"**
- **n_elt** is the **number of "things"**
- **stream** is the **stream from which to read**
- **returns number of "things" read** or **0 on failure** (or 0 requested)

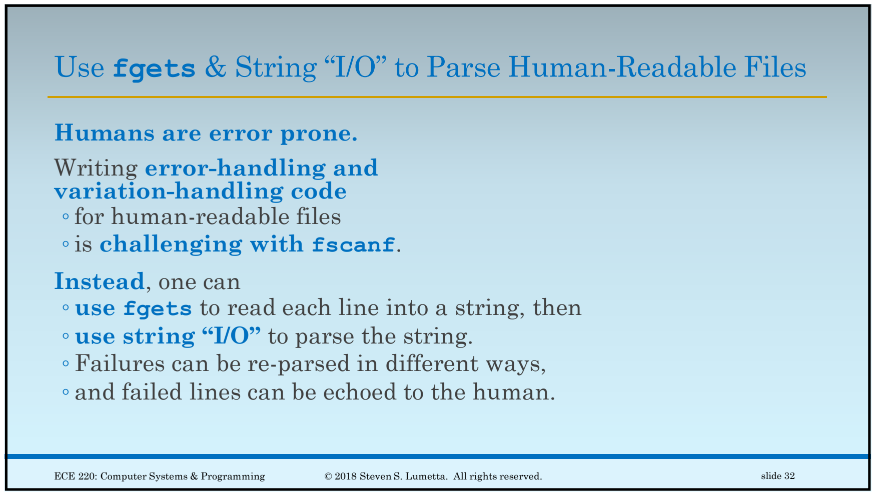


Use **fwrite** to Write Binary Output to a Stream

To write binary output to a stream:

```
size_t fwrite (const void* ptr,
     size_t size, size_t n_elt,
     FILE* stream);
```

- **ptr** is **address from which data are written**
- **size** is the **size of one "thing"**
- **n_elt** is the **number of "things"**
- **stream** is the **stream to which to write**
- **returns number of "things" written** or **0 on failure** (or 0 requested)

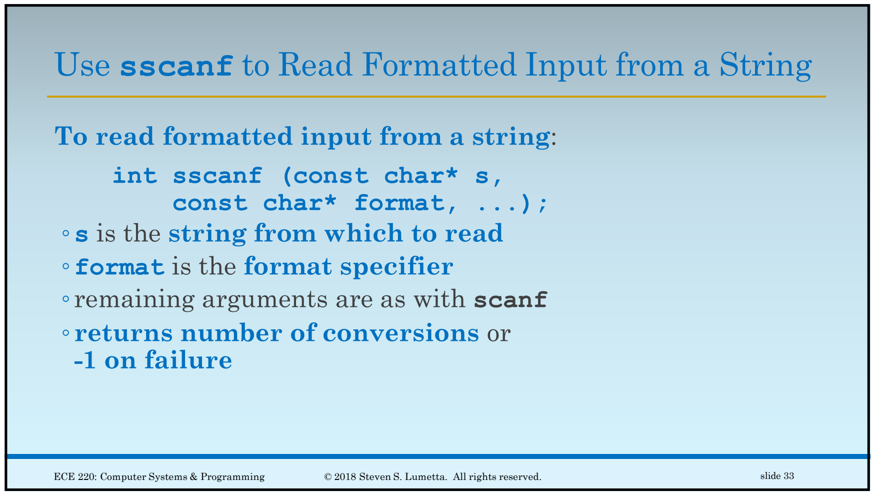


Use **fgets** & String "I/O" to Parse Human-Readable Files

Humans are error prone.

Writing **error-handling and variation-handling code**
- for human-readable files
- is **challenging with fscanf**.

Instead, one can
- **use fgets** to read each line into a string, then
- **use string "I/O"** to parse the string.
- Failures can be re-parsed in different ways,
- and failed lines can be echoed to the human.

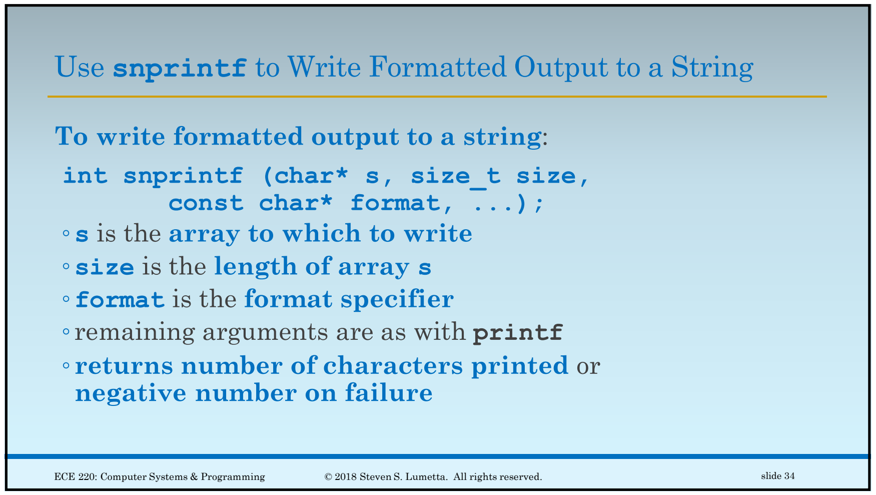


Use **`sscanf`** to Read Formatted Input from a String

To read formatted input from a string:

```
int sscanf (const char* s,
        const char* format, ...);
```

- **`s`** is the **string from which to read**
- **`format`** is the **format specifier**
- remaining arguments are as with **`scanf`**
- **returns number of conversions** or **-1 on failure**

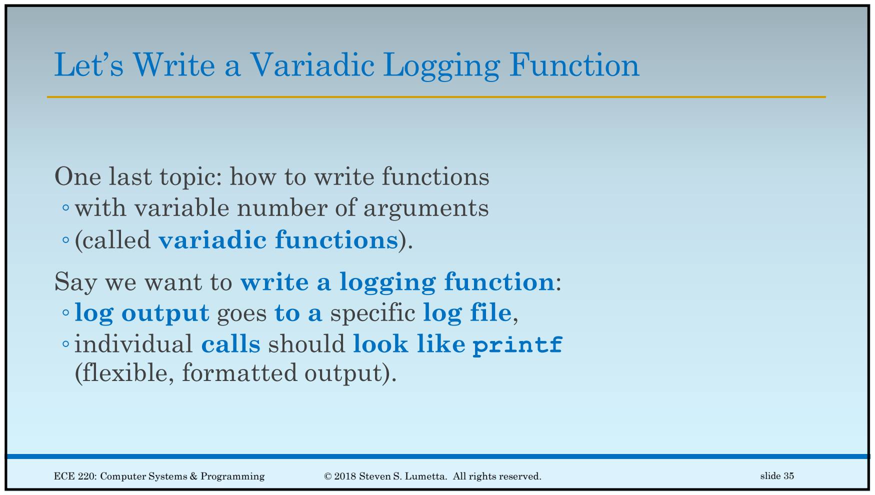


Use **`snprintf`** to Write Formatted Output to a String

To write formatted output to a string:

```
int snprintf (char* s, size_t size,
       const char* format, ...);
```

- **`s`** is the **array to which to write**
- **`size`** is the **length of array `s`**
- **`format`** is the **format specifier**
- remaining arguments are as with **`printf`**
- **returns number of characters printed** or **negative number on failure**



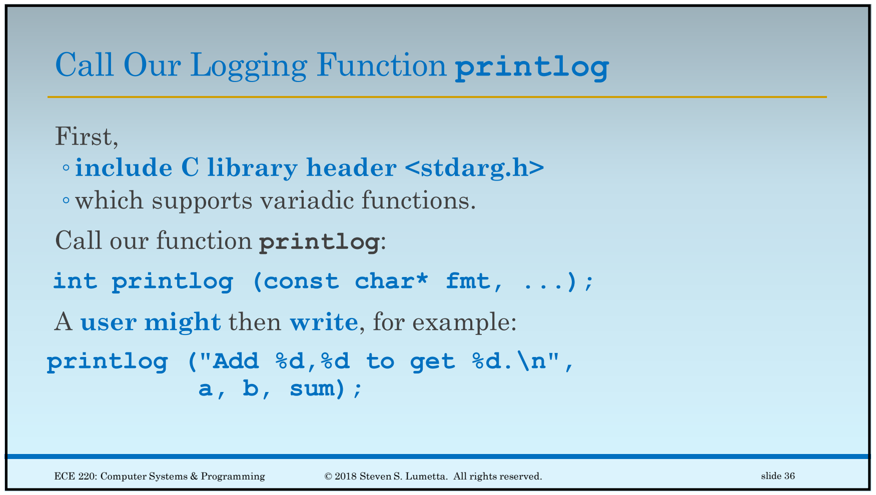
Let's Write a Variadic Logging Function

One last topic: how to write functions

- with variable number of arguments
- (called **variadic functions**).

Say we want to **write a logging function**:

- **log output** goes **to a** specific **log file**,
- individual **calls** should **look like `printf`** (flexible, formatted output).



Call Our Logging Function **`printlog`**

First,

- **include C library header <stdarg.h>**
- which supports variadic functions.

Call our function **`printlog`**:

```
int printlog (const char* fmt, ...);
```

A **user might** then **write**, for example:

```
printlog ("Add %d,%d to get %d.\n",
          a, b, sum);
```

Use a File-Scope Variable for the Log's Stream

Where is the stream for the log?

Let's put all of the log functionality in a file.

The **stream** can be **a file-scope variable**:

```
FILE* logfile = NULL;
```

Let's **initialize the stream in the first call** to **printlog**.

We should have a function to close the stream, too, but we won't write that function.

First Task: Open the Log File

Now we can start to write…

```
int printlog (const char* fmt, ...)
{
  if (NULL == logfile) {
    logfile = fopen ("the_log","a");
    if (NULL == logfile) {
      return -1;
    }
  }
```

First call?

Append to end of existing file.

Nowhere to write if **fopen** fails.

Return negative value to indicate failure.

Are You Ready to Rewrite **printf**?

Now what?

Rewrite printf?

That part is left to you.

But there's an easier way…

vfprintf

(The "v" is for variadic/variable arguments.)

Declare and Initialize a Variable Arguments List Variable

```
int printlog (const char* fmt, ...);
```

To use the arguments after fmt,
- we must **declare and initialize**
- **a "variable argument list" variable**.
- The type is **va_list**:

```
va_list args;
va_start (args, fmt);
```

va_start is a preprocessor macro that starts the variable-length list after the specified argument—in this case, **fmt**.

Add the Variable Arguments List to Our Code

```
int printlog (const char* fmt, ...)
{
  va_list args;
  if (NULL == logfile) {
    logfile = fopen ("the_log","a");
    if (NULL == logfile) {
      return -1;
    }
  }
  va_start (args, fmt);
```

Insert the blue lines as shown.

One More Line of Code: Call **`vfprintf`**

```
int printlog (const char* fmt, ...)
{
  va_list args;
  if (NULL == logfile) {
    logfile = fopen ("the_log","a");
    if (NULL == logfile) {
      return -1;
    }
  }
  va_start (args, fmt);
  return vfprintf (logfile, fmt, args);
}
```

Just call **`vfprintf`**!

What Does **`printlog`** Return?

```
int printlog (const char* fmt, ...);
```

But what does our function return?

Number of characters printed or a negative value on failure (just like printf).