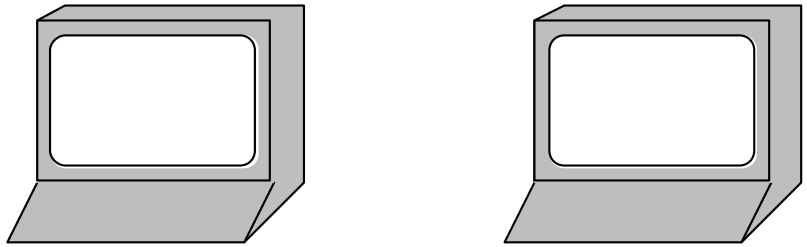
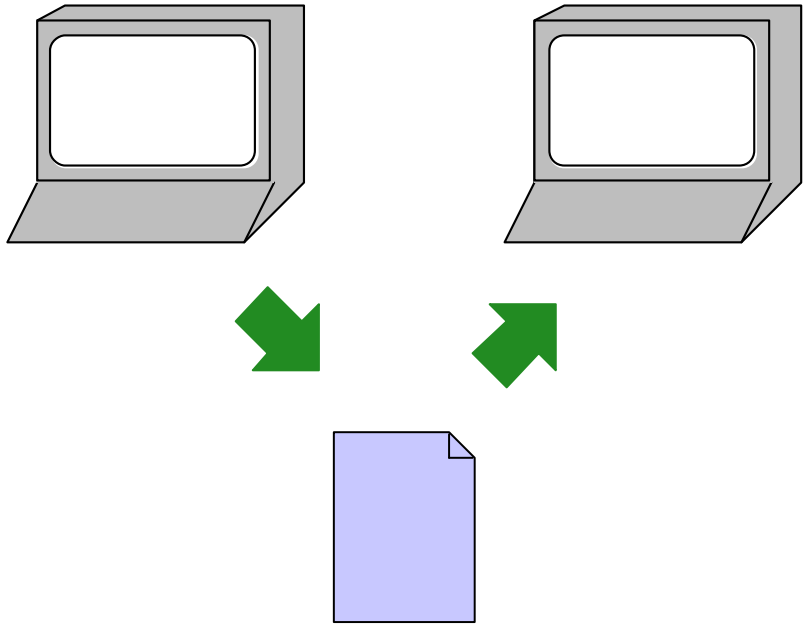
Multiple Programs

How do programs communicate?



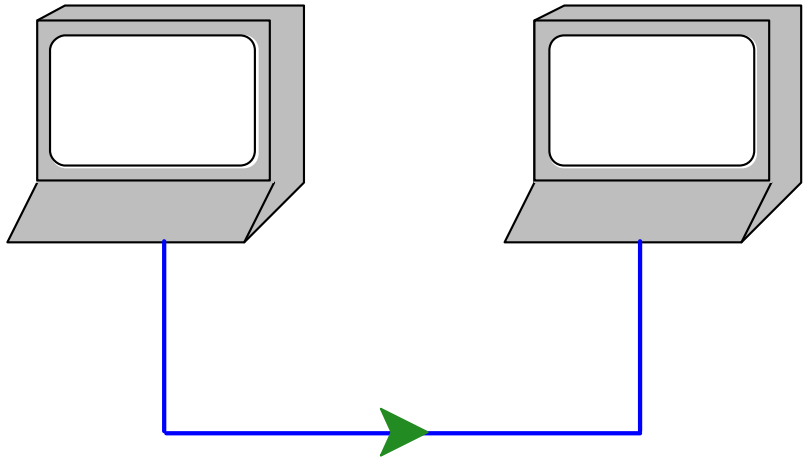
Multiple Programs

How do programs communicate? Files...



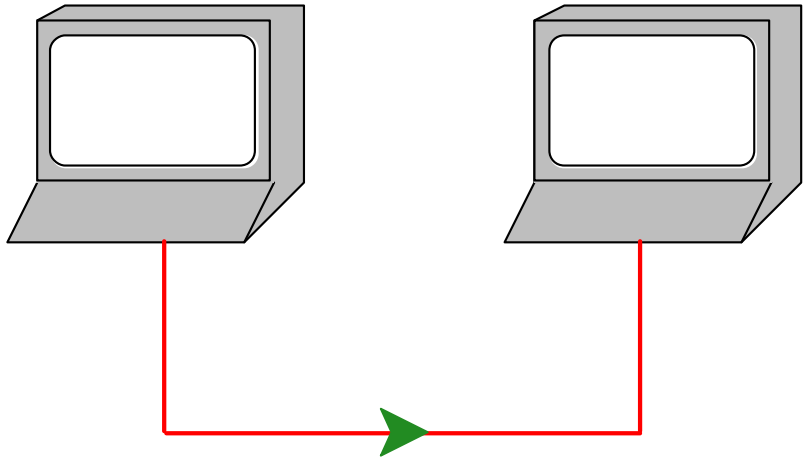
Multiple Programs

How do programs communicate? Files... Network...



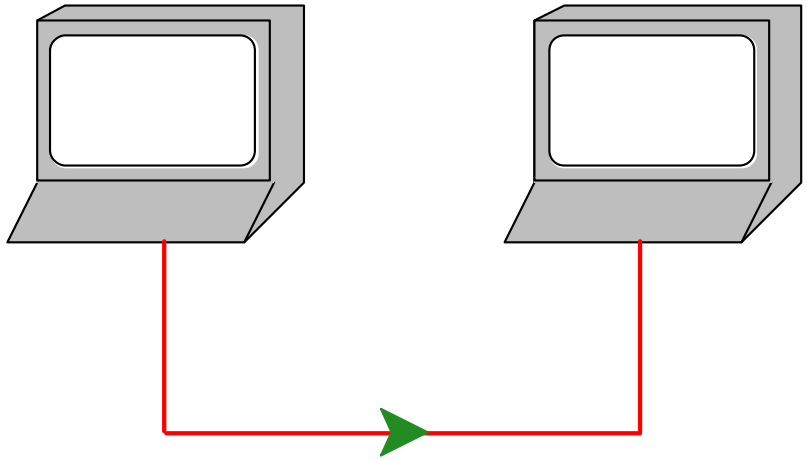
Multiple Programs

How do programs communicate? Files... Network... Stdin...



Multiple Programs

How do programs communicate? Files... Network... Stdin... Etc.



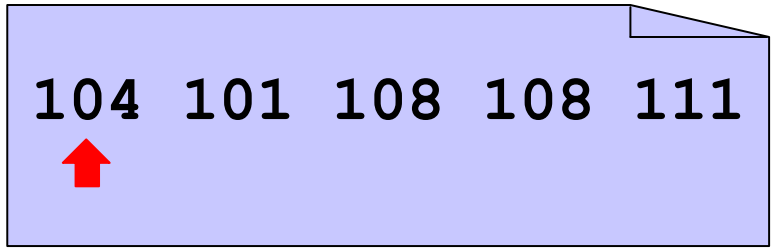
But what's in a file or sent over the network?

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



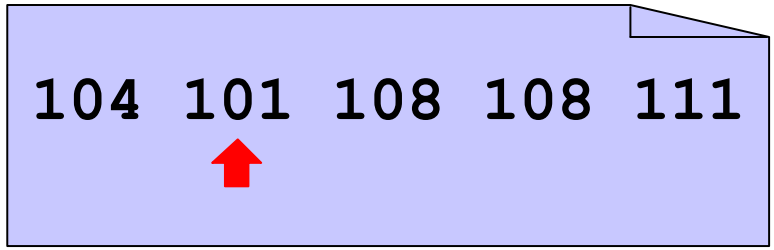

```
(read-byte in)
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



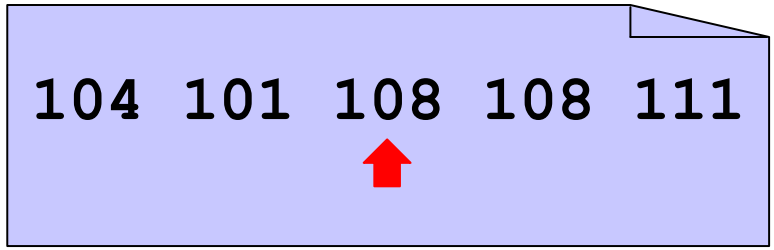
`(read-byte in)` → `104`

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



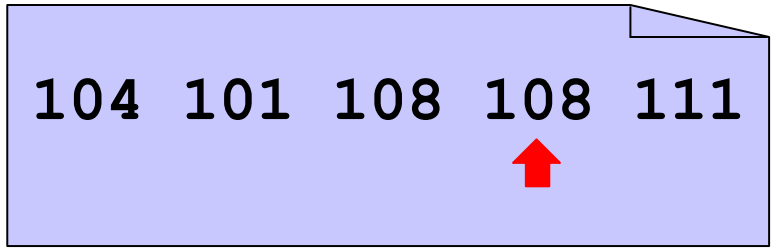

```
(read-byte in) → 104
(read-byte in) → 101
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



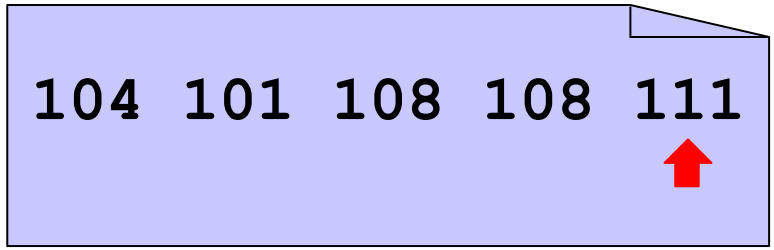

```
(read-byte in) → 104
(read-byte in) → 101
(read-byte in) → 108
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



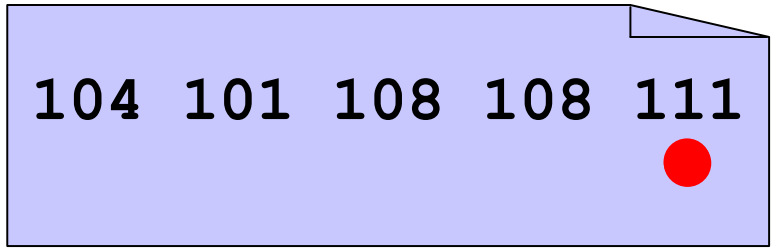

```
(read-byte in) → 104
(read-byte in) → 101
(read-byte in) → 108
(read-byte in) → 108
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



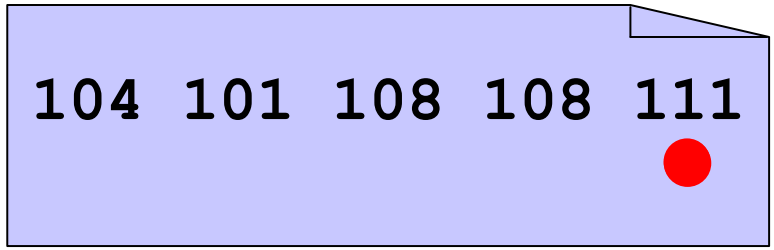

```
(read-byte in) → 104
(read-byte in) → 101
(read-byte in) → 108
(read-byte in) → 108
(read-byte in) → 111
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



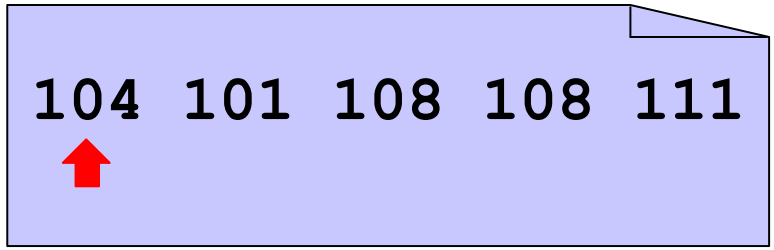

```
(read-byte in) → 104
(read-byte in) → 101
(read-byte in) → 108
(read-byte in) → 108
(read-byte in) → 111
(read-byte in) → eof-object
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



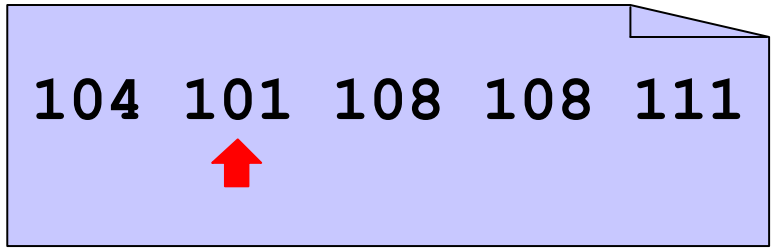
`fgetc(in)`

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



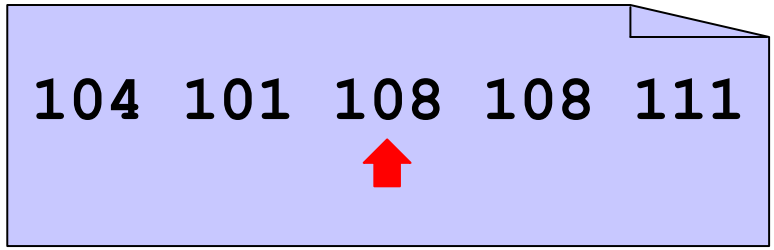
`fgetc(in)` → `104`

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



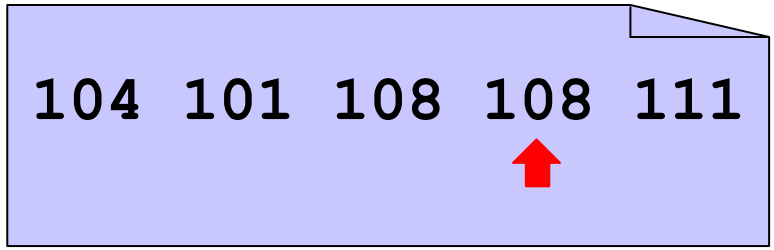

```
fgetc(in) → 104
fgetc(in) → 101
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



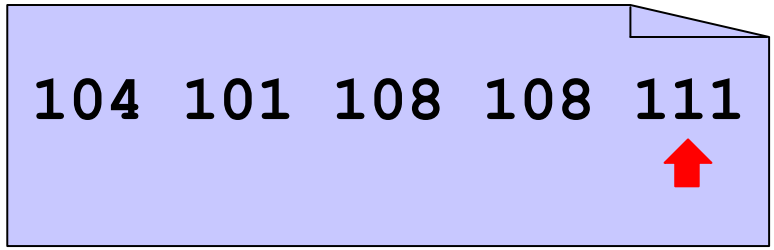

```
fgetc(in) → 104
fgetc(in) → 101
fgetc(in) → 108
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: **`read-byte`** or **`write-byte`**



`fgetc(in)` → `104`
`fgetc(in)` → `101`
`fgetc(in)` → `108`

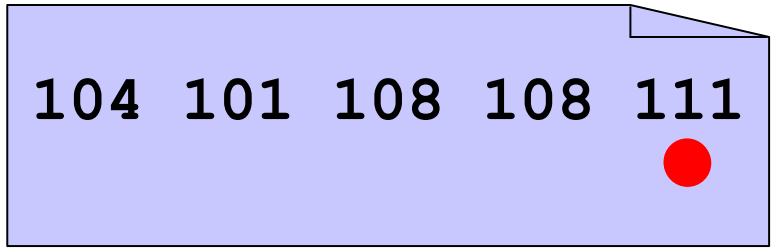
`fgetc(in)` → `108`

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



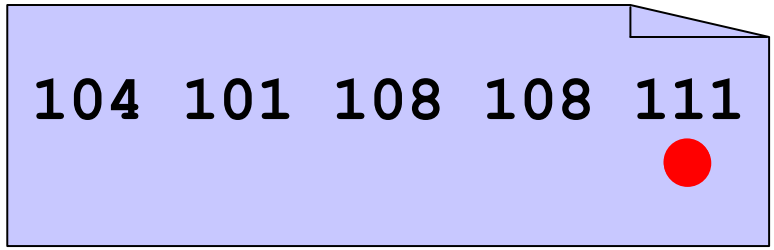

```
fgetc(in) → 104
fgetc(in) → 101
fgetc(in) → 108
fgetc(in) → 108
fgetc(in) → 111
```

Byte Streams

Operating systems provide files, network connections, etc. as ***byte stream*** objects

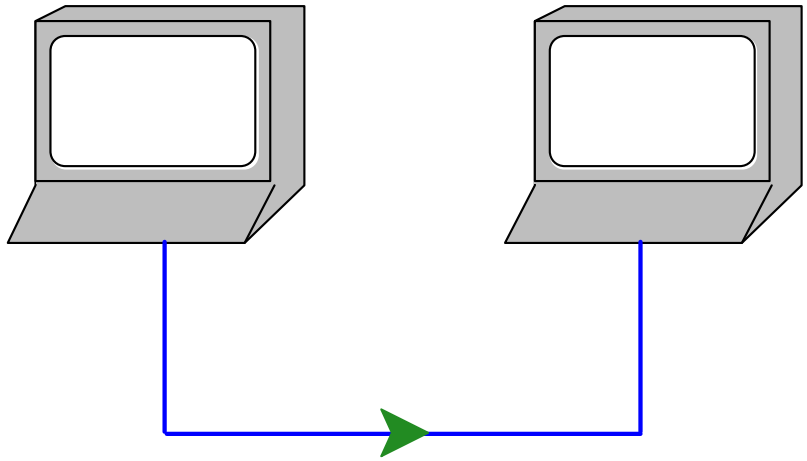
A ***byte*** is a number between `0` and `255`

A ***stream*** is a sequence with a counter and an operation: `read-byte` or `write-byte`



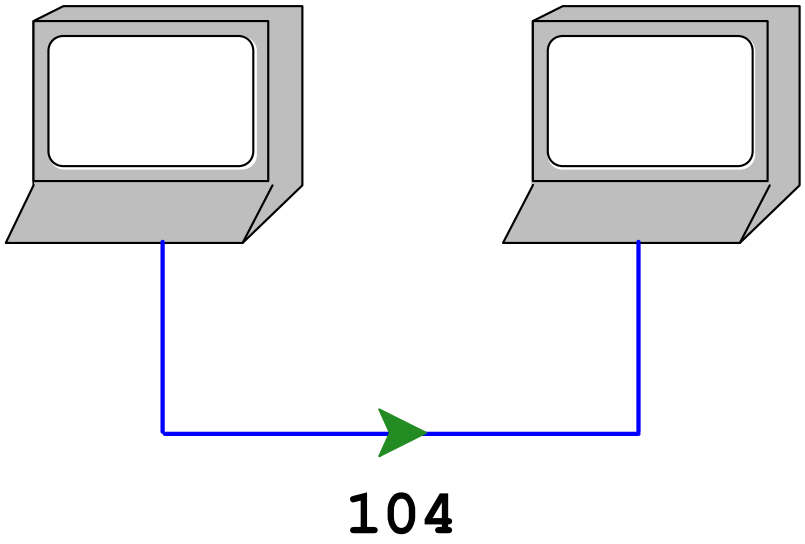

```
fgetc(in) → 104
fgetc(in) → 101
fgetc(in) → 108
fgetc(in) → 108
fgetc(in) → 111
fgetc(in) → -1
```

Byte Streams



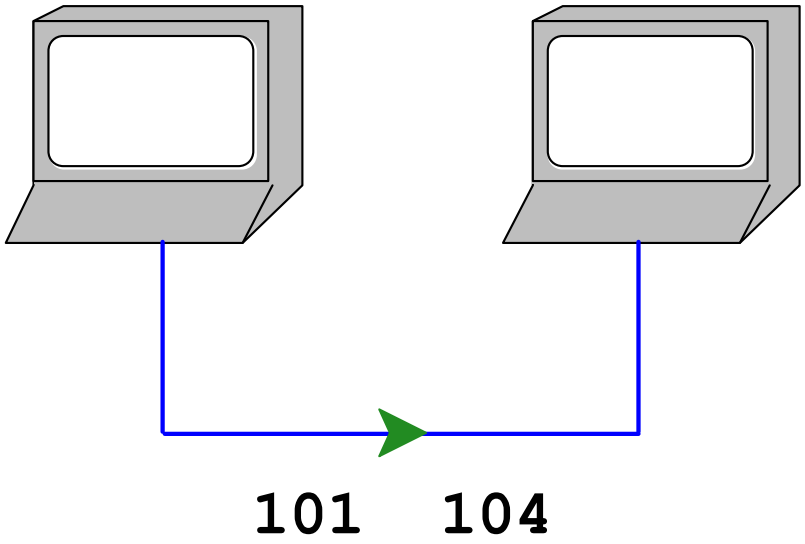
Byte Streams

```
(write-byte 104 o)
 → (void)
```



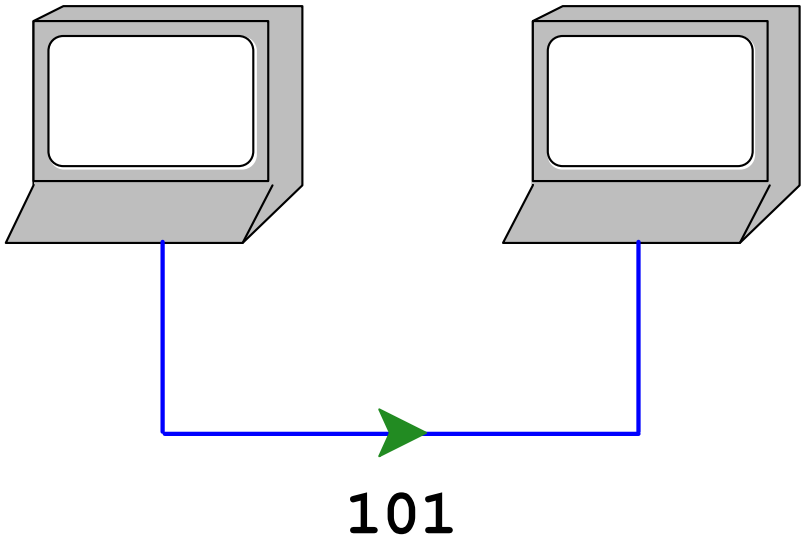
Byte Streams

```
(write-byte 104 o)
 → (void)

(write-byte 101 o)
 → (void)
```



Byte Streams

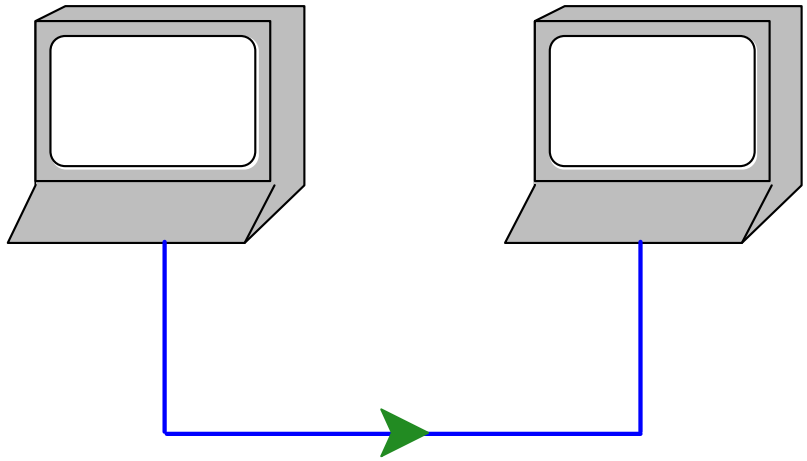
```
(write-byte 104 o)
 → (void)

(write-byte 101 o)
 → (void)
```




```
(read-byte i)
 → 104
```

Byte Streams

```
(write-byte 104 o)
 → (void)

(write-byte 101 o)
 → (void)
```



```
(read-byte i)
 → 104

(read-byte i)
 → 101
```

Encoding

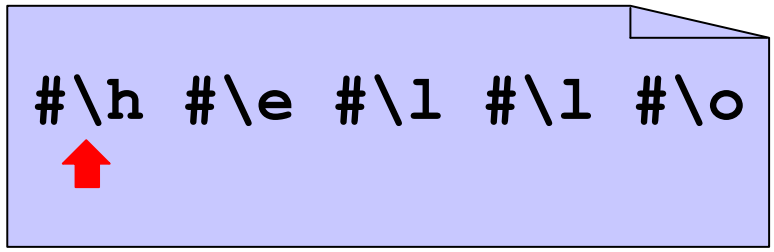
To communicate information other than small numbers, it must be ***encoded***

To encode English text, map each ***character*** to a byte

```
#\a   ⇒   97
#\b   ⇒   98
#\c   ⇒   99
      …
#\A   ⇒   65
      …
#\(   ⇒   40
#\)   ⇒   41
#\1   ⇒   48
      …
```

Character Streams

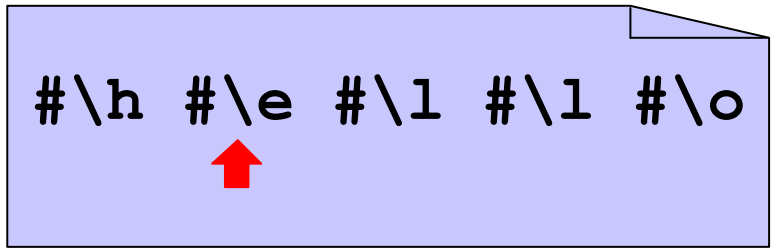
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***




```
(read-char in)
```

Character Streams

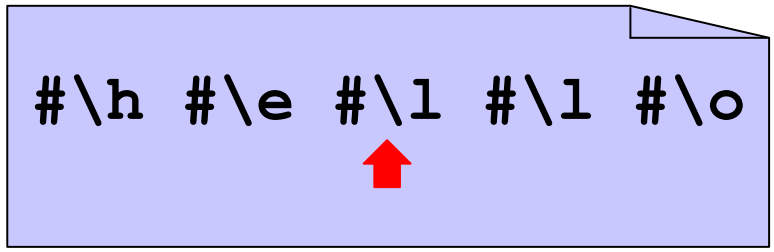
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***




```
(read-char in) → #\h
```

Character Streams

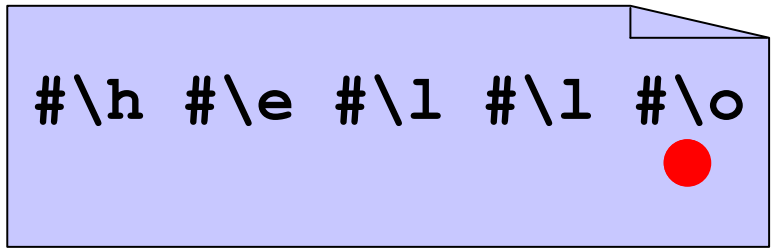
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***




```
(read-char in) → #\h

(read-char in) → #\e
```

Character Streams

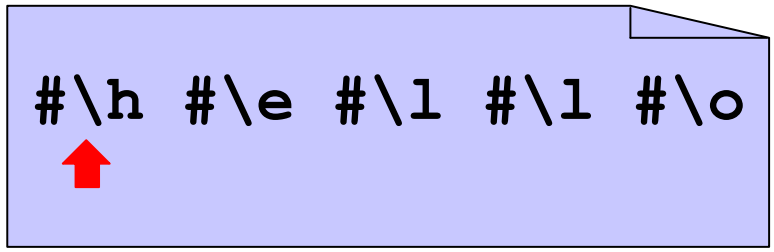
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***




```
(read-char in) → #\h

(read-char in) → #\e

...

(read-char in) → eof-object
```

Character Streams

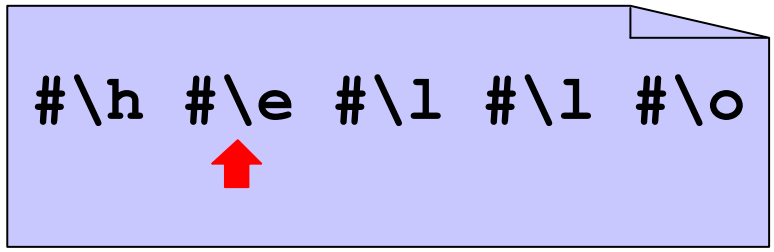
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***



`fgetc(in)`

Character Streams

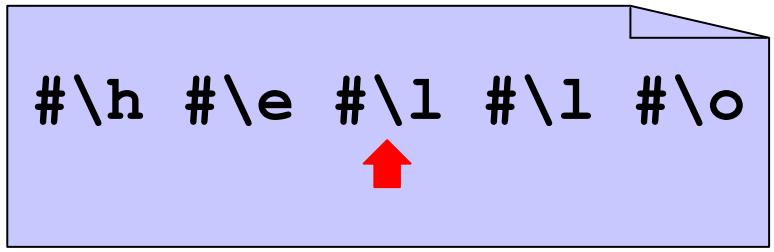
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***




```
fgetc(in) → 'h' /* = 104 */
```

Character Streams

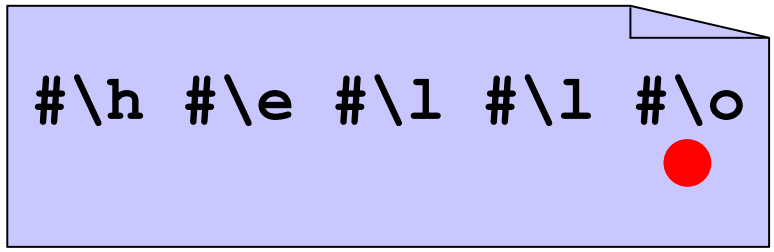
This character encoding is so popular that byte streams are sometimes viewed as ***character streams***




```
fgetc(in) → 'h' /* = 104 */

fgetc(in) → 'e' /* = 101 */
```

Character Streams

This character encoding is so popular that byte streams are sometimes viewed as ***character streams***




```
fgetc(in) → 'h' /* = 104 */

fgetc(in) → 'e' /* = 101 */

…

fgetc(in) → -1
```

Accessing Streams

Stream types:

- Racket:
  - input port
  - output port
- Java:
  - **`InputStream`**
  - **`PrintStream`**
- C:
  - **`FILE*`**

Accessing Streams

Getting standard input, output, and error-output:

- Racket:
  - `(current-input-port)`
  - `(current-output-port)`
  - `(current-error-port)`
- Java:
  - `System.out`
  - `System.in`
  - `System.err`
- C with `#include <stdio.h>`:
  - `stdin`
  - `stdout`
  - `stderr`

Accessing Streams

Reading or writing a file:

- Racket:
  - `(open-input-file filename)`
  - `(open-output-file filename)`
- Java:
  - `new BufferedReader(new FileReader(`*filename*`))`
  - `new BufferedWriter(new FileWriter(`*filename*`))`
- C with `#include <stdio.h>`:
  - `fopen(`*filename*`, "rb")`
  - `fopen(`*filename*`, "wb")`

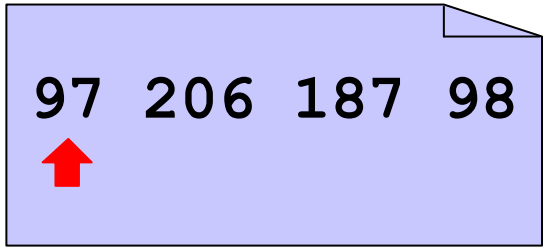
Character Streams in Racket

```
(define o (open-output-file "ex1"))
(write-char #\h o)
(write-char #\e o)
...
(close-output-port o)

(define i (open-input-file "ex1"))
(check-expect (read-char i) #\h)
(check-expect (read-char i) #\e)
...
(close-input-port i)
```

Note: Racket term for ***stream*** is ***port***

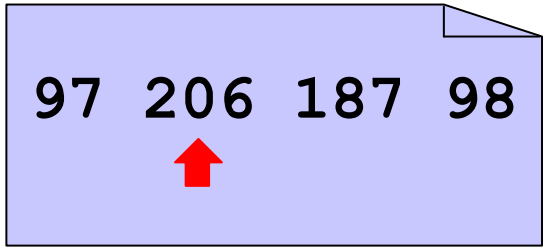
Encoding: Characters in Racket

In Racket, characters are actually encoded in multiple bytes, sometimes



Encoding: Characters in Racket

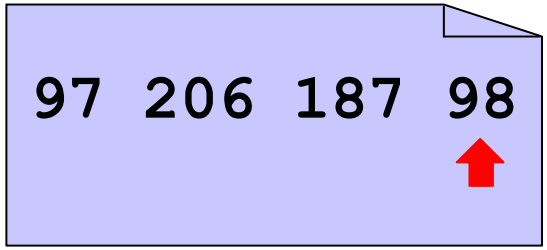
In Racket, characters are actually encoded in multiple bytes, sometimes




```
(read-char in) → #\a
```

Encoding: Characters in Racket

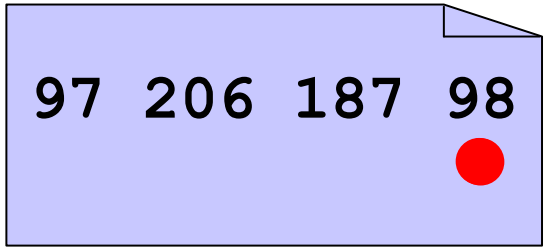
In Racket, characters are actually encoded in multiple bytes, sometimes




```
(read-char in) → #\a
(read-char in) → #\λ
```

Encoding: Characters in Racket

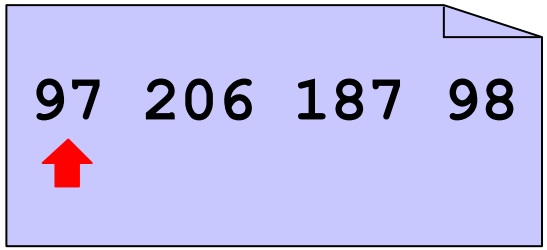
In Racket, characters are actually encoded in multiple bytes, sometimes




```
(read-char in) → #\a
(read-char in) → #\λ
(read-char in) → #\b
```

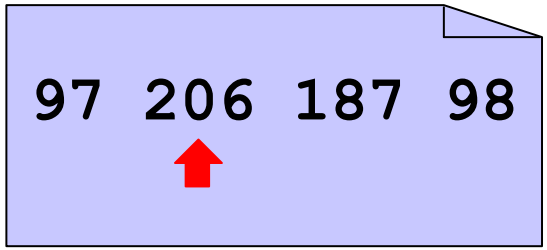
Encoding: Characters in C

In C, **`char`** just means "byte"



Encoding: Characters in C

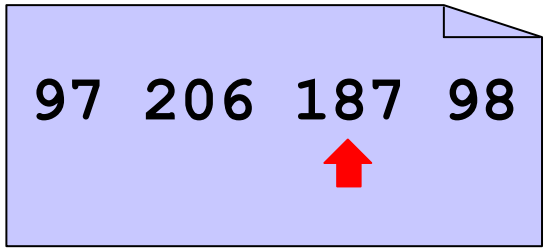
In C, **`char`** just means “byte”



`fgetc(in)` → **`'a'`**

Encoding: Characters in C

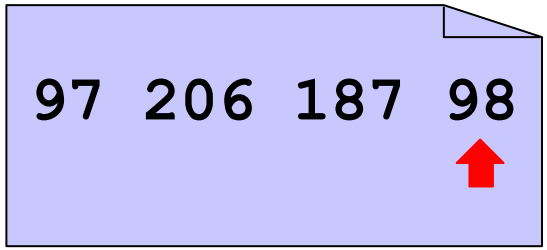
In C, **`char`** just means “byte”




```
fgetc(in) → 'a'

fgetc(in) → 'Î'
```

Encoding: Characters in C

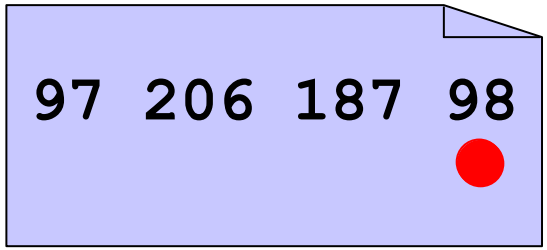
In C, **`char`** just means "byte"




```
fgetc(in) → 'a'

fgetc(in) → 'Î'

fgetc(in) → '»'
```

Encoding: Characters in C

In C, **`char`** just means "byte"




```
fgetc(in) → 'a'

fgetc(in) → 'Î'

fgetc(in) → '»'

fgetc(in) → 'b'
```

Some Character Encoding Standards

- ASCII
  - "Characters" 0 to 127
  - A kind of English plus computer creole
- Latin-1
  - "Characters" 0 to 255
  - A kind of Western Europe plus computer creole
  - A superset of ASCII
- UTF-8
  - "Characters" 0 to 917999 or so
  - Roughly covers all languages on Earth
  - A superset of ASCII
- UTF-16
  - Same coverage as UTF-8
  - Uses 2 or 4 bytes for each character
- ...

Communicating Strings

One string: encode as a sequence of characters

Multiple strings: need a way to mark the end of one string

Communicating Strings

One string: encode as a sequence of characters

Multiple strings: need a way to mark the end of one string

The most popular encoding is ***line-based***:

- Use a newline (encoded as 10) to separate strings
  - `#\newline` or `'\n'`
- Works for strings that don't contain newlines

- Racket:
  - `(read-line input-port)`
- C:
  - `fgets(buffer, len, stream)`

CRLF versus LF

Sometimes, lines are separated by two characters (***CRLF***: 13 then 10) instead of one (***LF***: 10):

`"one\ntwo\n"` versus **`"one\r\ntwo\r\n"`**

The encoding convention depends on the platform

Opening a file in "text mode" reads CRLF or LF as newline, as appropriate for a given platform

- Racket:
  - `(open-input-file #:mode 'text filename)`
  - `(open-output-file #:mode 'text filename)`
- C:
  - **`fopen`**`(`*filename*`, "r")`
  - **`fopen`**`(`*filename*`, "w")`

Communicating More Than Characters

To read and write aquariums, we need to communicate lists of (large) numbers

Communicating More Than Characters

To read and write aquariums, we need to communicate lists of (large) numbers

Again, we must encode:

```
empty       ⇒   #\.
'(10000)    ⇒   #\1 #\0 #\0 #\0 #\space #\.
'(1 2)      ⇒   #\1 #\space #\2 #\space #\.
            ...
```

Number List Serialization

A **<numlist>** is either

#\.

<num> #\space <numlist>

A **<num>** is either

<digit>

<num> <digit>

A **<digit>** is either

#\0

#\1

…

#\9

Number List Writer

```
; write-numlist : list-of-num output-port -> void
(define (write-numlist l p)
  (cond
   [(empty? l) (write-char #\. p)]
   [else (begin
           (write-num (first l) p)
           (write-char #\space p)
           (write-numlist (rest l) p))]))

; write-num : num output-port -> void
(define (write-num n p)
  (cond
   [(< n 10) (write-digit n p)]
   [else (begin
           (write-num (quotient n 10) p)
           (write-digit (remainder n 10) p))]))

; write-digit : num [0-9] output-port -> void
(define (write-digit n p)
  (cond
   [(= n 0) (write-char #\0 p)]
   ...
   [(= n 9) (write-char #\9 p)]))
```

Number List Parsing

Parse using an equivalent but more convenient form:

A `<numlist>` is either

- `#\.`
- `<num> #\space <numlist>`

A `<num>` is either

- `<digit>`
- `<num> <digit>`

A `<digit>` is either

- `#\0`
- `#\1`
- …
- `#\9`

⇒

A `<numlist>` is either

- `#\.`
- `#\0 <num> <numlist>`
- …
- `#\9 <num> <numlist>`

A `<num>` is either

- `#\space`
- `#\0 <num>`
- …
- `#\9 <num>`

Number List Reader

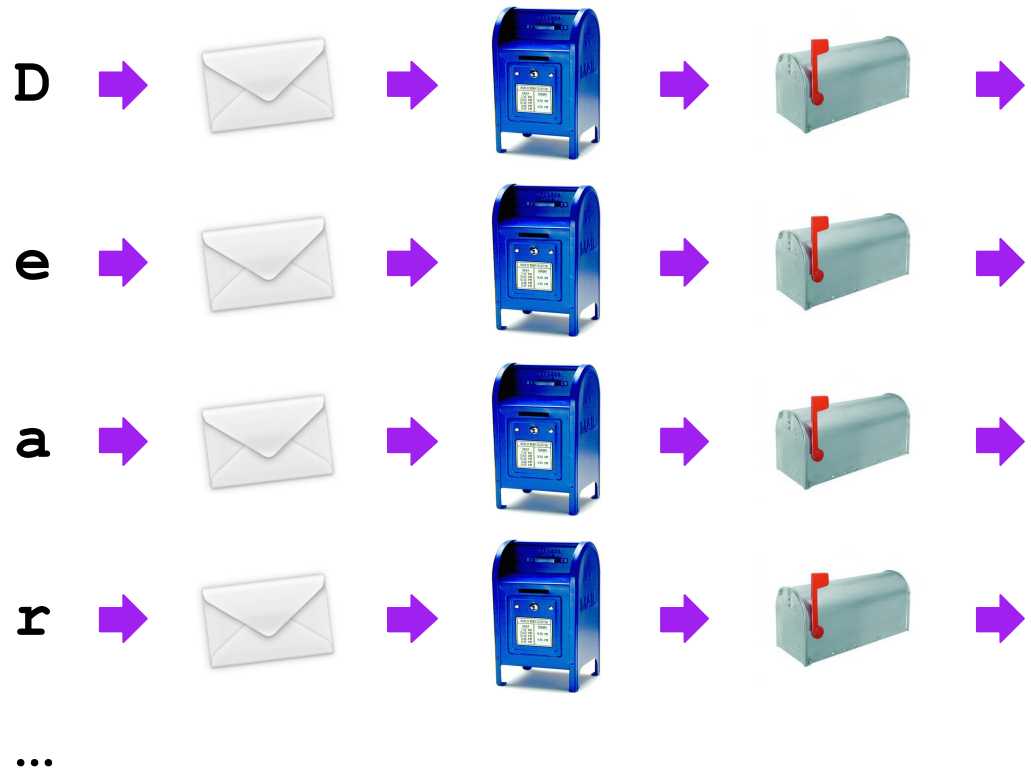
```
; read-numlist : input-port -> list-of-num
(define (read-numlist p)
  (local [(define c (read-char p))]
    (cond
     [(char=? #\. c) empty]
     [(char-digit? c) (cons (read-number p (digit-val c))
                            (read-numlist p))])))

; read-number : input-port num -> num
(define (read-number p n)
  (local [(define c (read-char p))]
    (cond
     [(char=? #\space c) n]
     [(char-digit? c)
      (read-number p (+ (* n 10) (digit-val c)))])))

; char-digit? : char -> bool
...

; digit-val : char -> num
...
```

I/O Libraries

You don't always have to start from scratch

- Racket:
  - **`read`** and **`write`**
  - **`read-line`** and **`displayln`**
  - **`read-xml`** and **`write-xml`**
  - ...
- C:
  - **`fscanf`** and **`fprintf`**
  - ...

Buffers



vs.

Buffers

A ***buffer*** is why you see no output from

```
int main() {
   printf("hello");
   crash();
}
```

Buffers

A ***buffer*** is why you see no output from

```
int main() {
    printf("hello");
    crash();
}
```

Line-buffering is why you do see output from

```
int main() {
    printf("hello\n");
    crash();
}
```

... unless you redirect to an output file

Buffers

Flushing buffers:

- Racket:
  - `(flush-output output-port)`
- C:
  - **`fflush`**`(`*stream*`)`