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**

```
104 101 108 108 111
```

(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*

```
104 101 108 108 111
```

(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*

```
104 101 108 108 111
```

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

```
104 101 108 108 111
↑
```

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

```
104 101 108 108 111
```

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*

\[
\begin{array}{cccccc}
104 & 101 & 108 & 108 & 111 \\
\end{array}
\]

\[\text{fgetc}(\text{in}) \rightarrow 104\]
\[\text{fgetc}(\text{in}) \rightarrow 101\]
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**

```
104 101 108 108 111
```

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

\[(\text{write-byte 104 o}) \rightarrow (\text{void})\]

\[(\text{write-byte 101 o}) \rightarrow (\text{void})\]

\[(\text{read-byte i}) \rightarrow 104\]

\[(\text{read-byte i}) \rightarrow 101\]
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*

```
\#\h \#\e \#\l \#\l \#\o
```

(read-char in) → \#\h
Character Streams

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

\[
\text{(read-char in)} \rightarrow \#\text{\textbackslash h}
\]

\[
\text{(read-char in)} \rightarrow \#\text{\textbackslash e}
\]
Character Streams

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

\[
\text{(read-char in)} \rightarrow \#\h
\]

\[
\text{(read-char in)} \rightarrow \#\e
\]

... 

\[
\text{(read-char in)} \rightarrow \text{eof}
\]
Character Streams

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

```c
fgetc(in)
```

Character Streams

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

```
\#h \#e \l \l \o
```

`fgetc(in) \rightarrow 'h' /* = 104 */`
Character Streams

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

\[
\text{fgetc}(\text{in}) \rightarrow 'h' \quad /* = 104 \ */
\]

\[
\text{fgetc}(\text{in}) \rightarrow 'e' \quad /* = 101 \ */
\]
Character Streams

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

```c
fgetc(in) → 'h'  /* = 104 */
```

```c
fgetc(in) → 'e'  /* = 101 */
```

...

```c
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 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

```
97 206 187 98
```
Encoding: Characters in Racket

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

\[(\text{read-char in}) \rightarrow \#\backslash a\]
Encoding: Characters in Racket

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

```
(\read-char \in) \rightarrow #\a
(\read-char \in) \rightarrow #\l
```
Encoding: Characters in Racket

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

\[
\text{(read-char in)} \rightarrow \#\textbackslash a
\]

\[
\text{(read-char in)} \rightarrow \#\textbackslash \lambda
\]

\[
\text{(read-char in)} \rightarrow \#\textbackslash b
\]
Encoding: Characters in C

In C, `char` just means “byte”
Encoding: Characters in C

In C, `char` just means “byte”

\[ 97 \ 206 \ 187 \ 98 \]

\[ \text{fgetc}(\text{in}) \rightarrow 'a' \]
Encoding: Characters in C

In C, `char` just means “byte”

```
97 206 187 98
```

`fgetc(in) → 'a'`

`fgetc(in) → 'î'`
Encoding: Characters in C

In C, `char` just means “byte”

\[
\begin{array}{cccc}
97 & 206 & 187 & 98 \\
\end{array}
\]

\[
\text{fgetc}(\text{in}) \rightarrow 'a' \\
\text{fgetc}(\text{in}) \rightarrow 'î' \\
\text{fgetc}(\text{in}) \rightarrow '›'
\]
Encoding: Characters in C

In C, `char` just means “byte”

```
97 206 187 98

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

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:
  - (\texttt{read-line input-port})

- C:
  - \texttt{fgets(buffer, len, stream)}
CRLF versus LF

Sometimes, lines are separated by two characters (\textbf{CRLF}: 13 then 10) instead of one (\textbf{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:
  - (\begin{verbatim}open-input-file #:mode 'text filename\end{verbatim})
  - (\begin{verbatim}open-output-file #:mode 'text filename\end{verbatim})

- C:
  - \texttt{fopen(filename, "r")}
  - \texttt{fopen(filename, "w")}
Communicating More Than Characters

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

Again, we must encode:

\[
\begin{align*}
\text{empty} & \rightarrow \text{'#\ \#\ .'} \\
'(10000) & \rightarrow \text{'#\1 \#\0 \#\0 \#\0 \#\space \#\ .'} \\
'(1\ 2) & \rightarrow \text{'#\1 \#\space \#\2 \#\space \#\ .'} \\
\cdots
\end{align*}
\]
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
\[
\#ackslash .
\]
\[
<\text{num}> \#\backslash space <\text{numlist}>
\]

A `<num>` is either
\[
<\text{digit}>
\]
\[
<\text{num}> <\text{digit}>
\]

A `<digit>` is either
\[
\#\backslash 0
\]
\[
\#\backslash 1
\]
\[
\ldots
\]
\[
\#\backslash 9
\]

\[
\Rightarrow
\]

A `<numlist>` is either
\[
\#ackslash .
\]
\[
\#\backslash 0 <\text{num}> <\text{numlist}>
\]
\[
\ldots
\]
\[
\#\backslash 9 <\text{num}> <\text{numlist}>
\]

A `<num>` is either
\[
\#\backslash space
\]
\[
\#\backslash 0 <\text{num}>
\]
\[
\ldots
\]
\[
\#\backslash 9 <\text{num}>
\]
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

D → envelope → blue mailbox → green mailbox →

E → envelope → blue mailbox → green mailbox →

A → envelope → blue mailbox → green mailbox →

R → envelope → blue mailbox → green mailbox →

... →

vs.

Dear Sir: ... → envelope → blue mailbox → green mailbox →
Buffers

A buffer is why you see no output from

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

**Line-buffering** is why you do see output from

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

... unless you redirect to an output file
Buffers

Flushing buffers:

• Racket:
  ○ `(flush-output output-port)`

• C:
  ○ `fflush(stream)`