

The Internet, the Web and file processing

Finding and requesting remote resources
Keeping data beyond a program run

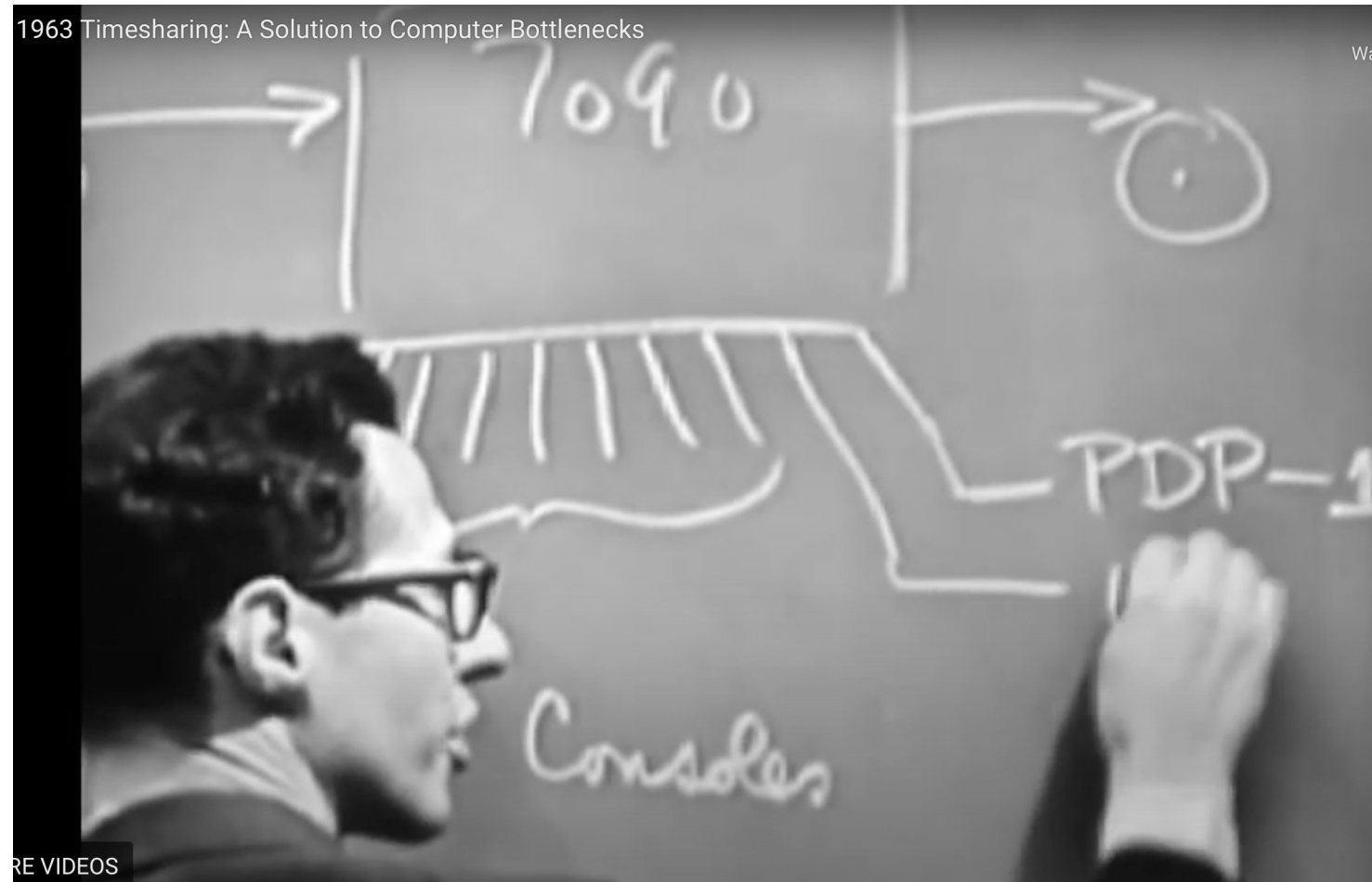
Prof. Dr. Mehmet Göktürk · Gebze Technical University

The Internet connects networks. Applications give those connections useful meaning.

The Web combines resource addresses, requests and linked documents.

Files let a program save data for a later run or another program.

Early computer communication experiments



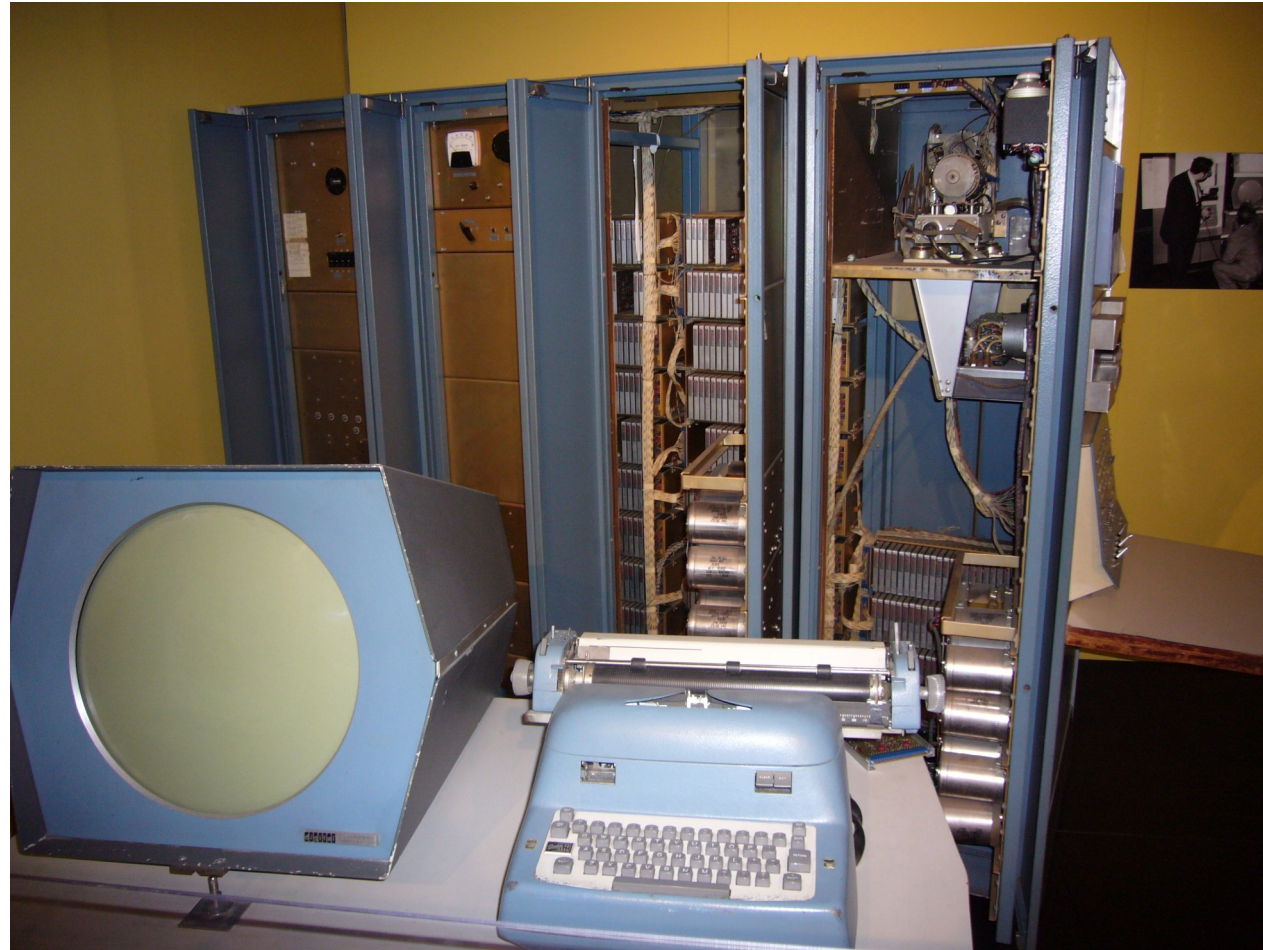
Remote terminals made access to a shared computer possible.

IBM 7090



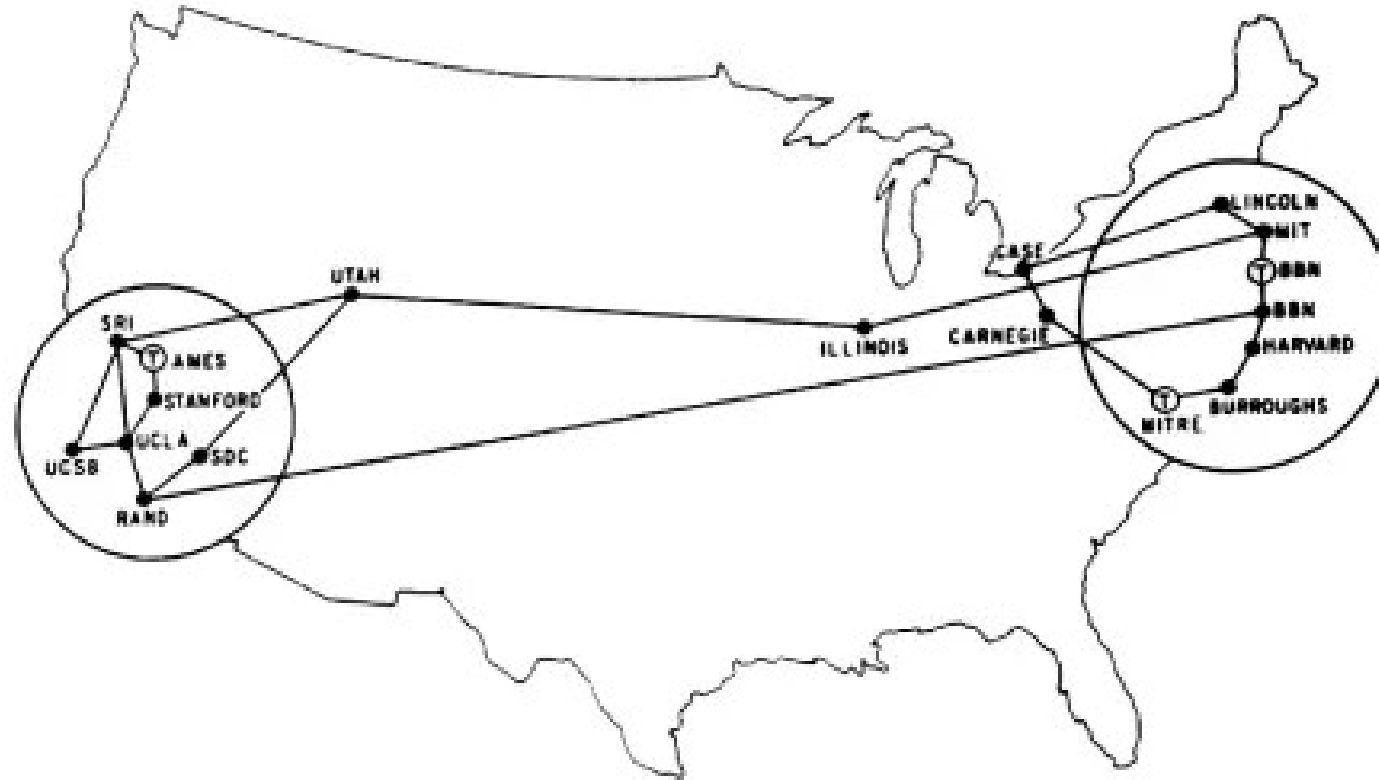
Large computers made remote access and resource sharing valuable.

DEC PDP-1



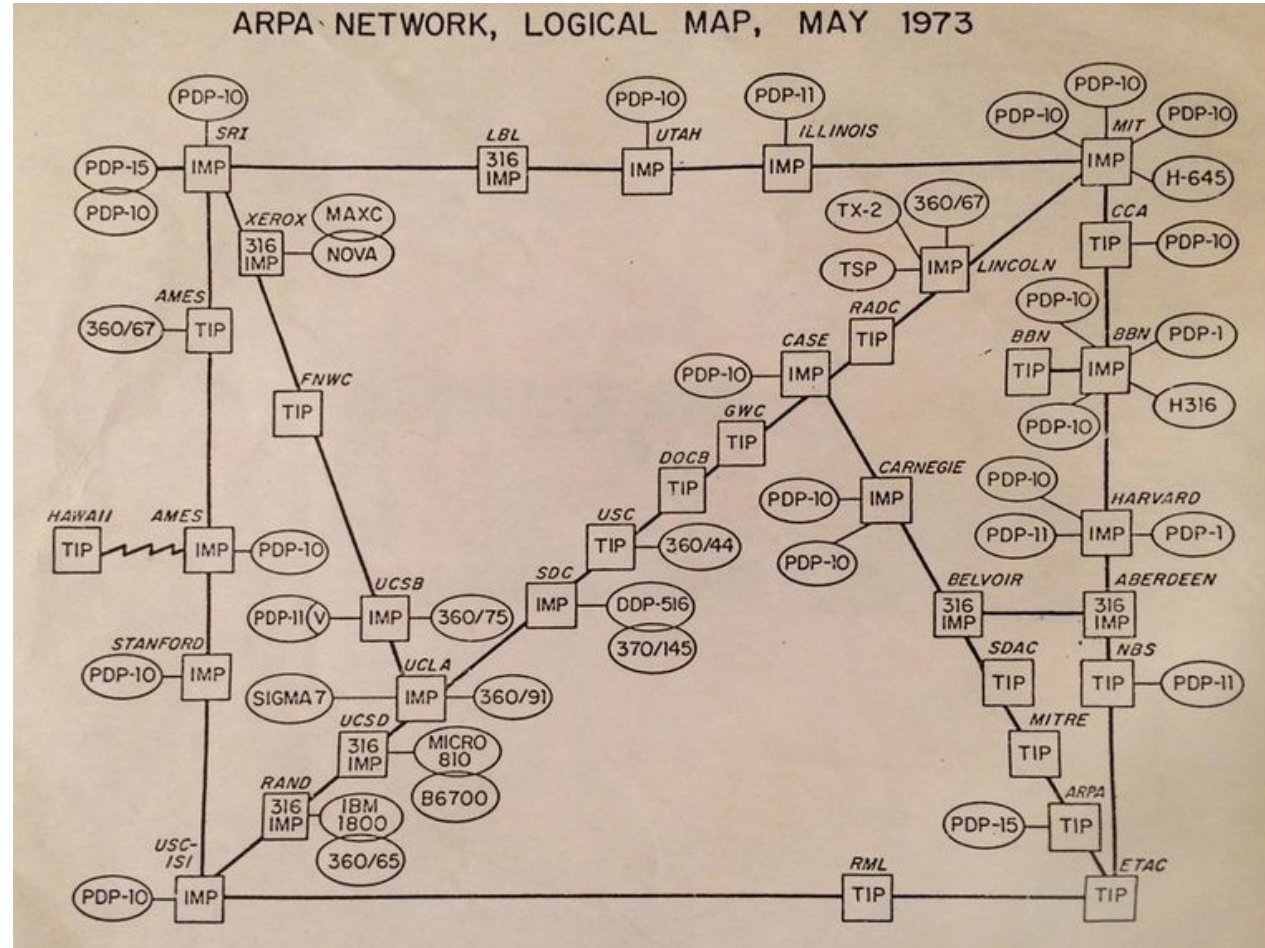
Historical computer photograph retained from the original collection.

An early network map



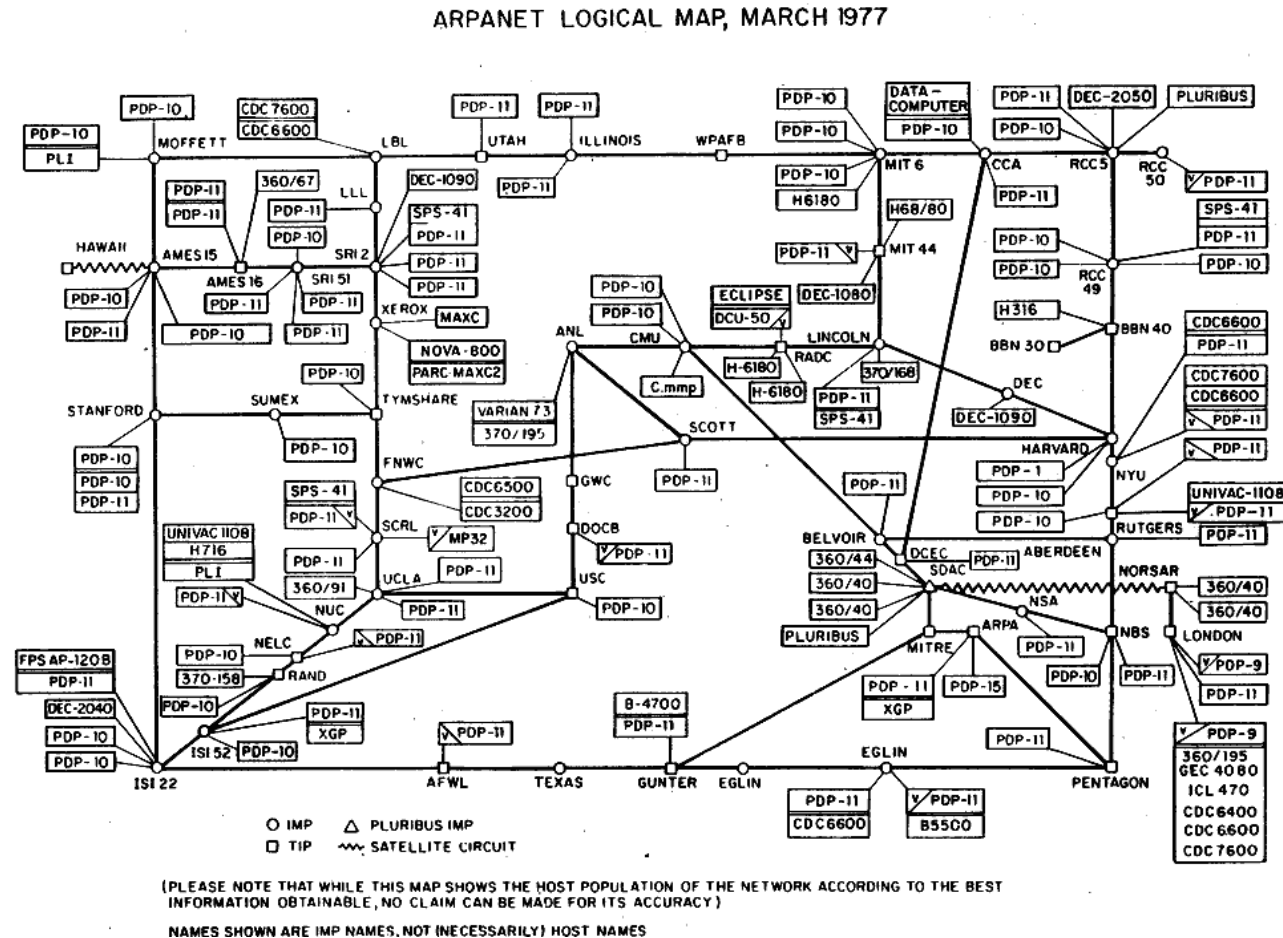
Early networks connected geographically separated research sites.

ARPANET logical map, May 1973



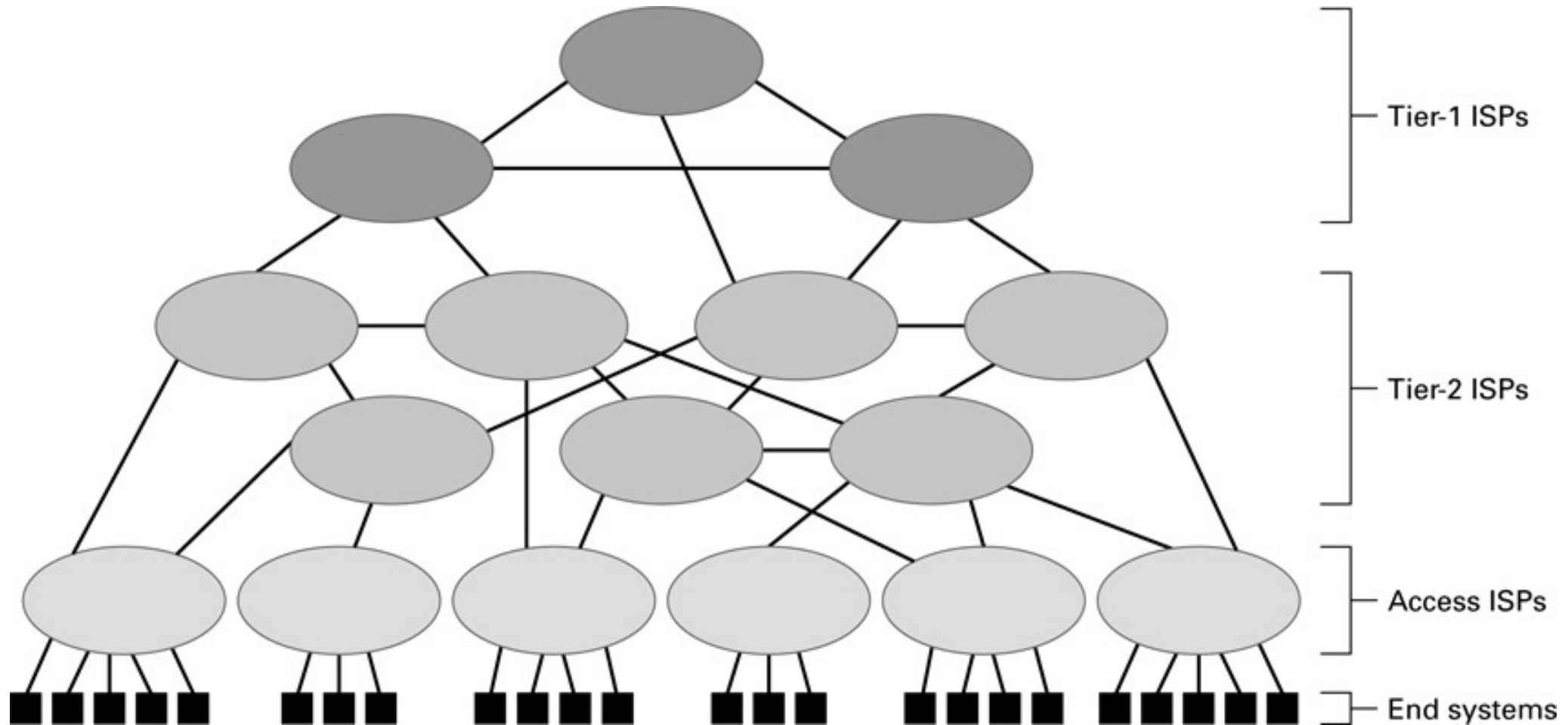
The date comes from the original map heading. This is a historical network snapshot.

ARPANET logical map, March 1977



A later historical snapshot shows the expanding network.

The Internet as interconnected networks



A simplified ISP model: real networks also peer directly and connect through exchange points.

ARPANET, Internet and the Web

Term	Role
ARPANET	An early packet-switched network; a precursor to the Internet
Internet	Network infrastructure connecting networks
World Wide Web	Resources linked and accessed using Web technologies
Browser	Client software that retrieves and presents resources
Search engine	A service that indexes content and helps find resources

Opening a search page is one use of a browser; the browser can also open a known URL.

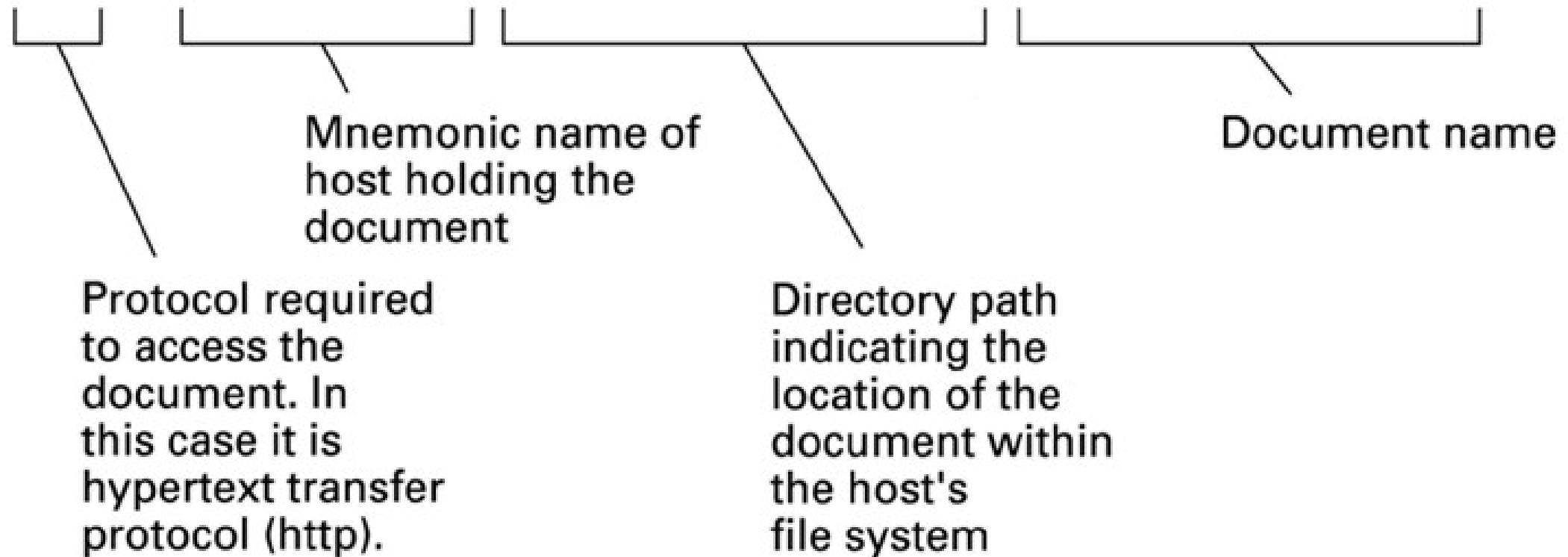
A service can use several computers while presenting one name to its users.

Different computers may handle requests, store data or keep copies close to users.

Distribution adds capacity and resilience, but communication delays and partial failures remain.

A resource address: URL

`http://eagle.mu.edu/authors/Shakespeare/Julius_Cesar.html`



A URL path identifies a resource. It need not match a physical directory or file.

Parts of a Web address

`https://example.org:443/readings/week10?lang=en#files`

scheme:	https
host:	example.org
port:	443
path:	/readings/week10
query:	lang=en
fragment:	files

The host is used in locating the service.

The fragment is handled by the client; it is not sent as part of the HTTP request target.

DNS stores records associated with domain names, including IP addresses.

A name can have several addresses. Several names can lead to the same server.

DNS locates information about a name; it does not fetch the webpage or search its contents.

DNS hierarchy

Level	Example	Responsibility
Root	.	Refers resolvers toward top-level domains
Top-level domain	org	Delegates names below it
Authoritative zone	example.org	Publishes records for names in its zone
Host name	www.example.org	Can have address or alias records

Administrative delegation divides responsibility; no one server stores every name.

An uncached DNS lookup

Step	Question or answer
Browser / OS → resolver	Find an address for www.example.org
Resolver → root	Which servers handle org?
Resolver → org server	Which servers handle example.org?
Resolver → authoritative server	What address records exist for www.example.org?
Resolver → client	Return the result, or report failure

A simplified walk. Cached records can skip steps; aliases can add steps.

A resolver can reuse a cached answer during its permitted lifetime, expressed by a time to live (TTL).

After an address change, some clients may temporarily use older cached information.

Name resolution and service availability are separate: a correct address does not guarantee a working server.

Opening an HTTPS page

Stage	What the browser needs
Resolve the host	An address for the service, unless already known
Establish secure communication	A suitable connection and server authentication
Request the resource	HTTP method, target and relevant headers
Process the response	Status, metadata and content
Display and continue	Parse content; fetch additional resources as needed

Connections and cached resources may already exist, so not every visit repeats all steps.

An HTTP request

```
GET /readings/week10?lang=en HTTP/1.1  
Host: example.org  
Accept: text/html
```

GET requests a representation of the resource.

This is readable HTTP/1.1 notation. HTTP/2 and HTTP/3 use different wire framing.

An HTTP response

```
HTTP/1.1 200 OK  
Content-Type: text/html; charset=utf-8  
Content-Length: 20
```

```
<h1>Week 10</h1>  
OK!
```

Status reports the request outcome.

Headers describe the content.

The body here has 20 UTF-8 bytes, with no final newline.

Common HTTP outcomes

Status	Meaning in this example
200 OK	A successful response with the requested content
301 / 302	Look at another location, normally given in a header
404 Not Found	The server did not find a current representation
500 Internal Server Error	The server encountered an unexpected condition

A 404 response means an HTTP server answered. A DNS failure happens before this exchange.

HTTPS protects HTTP communication in transit and authenticates the server name through TLS.

It does not prove that every statement on a page is true or that the application has no defects.

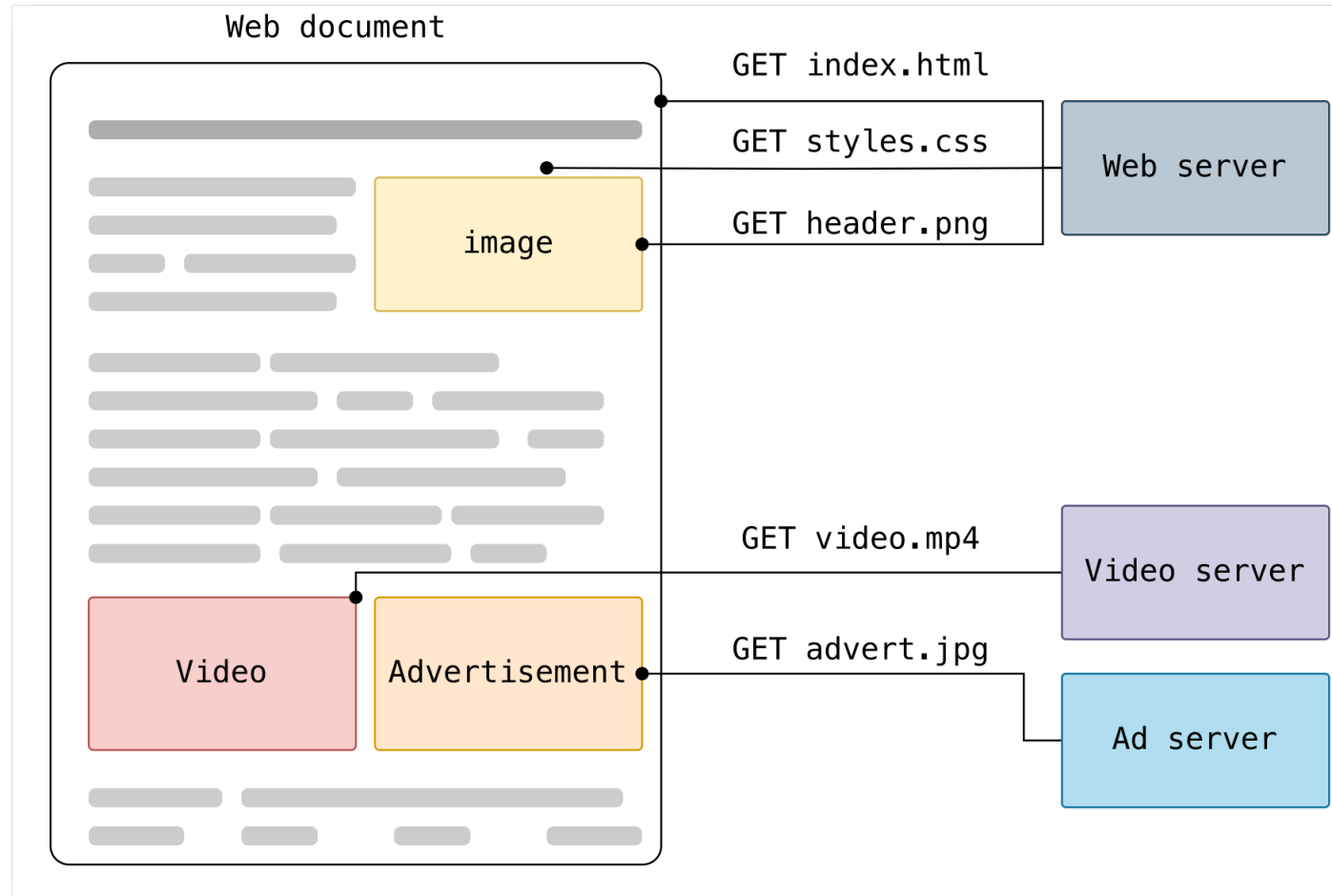
The browser and server still need to interpret the exchanged data correctly.

HTTP versions and transport

Version	How messages travel
HTTP/1.1	Textual message syntax over TCP; connections can be reused
HTTP/2	Binary framing with multiplexed streams over TCP
HTTP/3	HTTP over QUIC, which uses UDP underneath

HTTP/3 is not raw unreliable UDP: QUIC supplies reliable streams and secure transport.

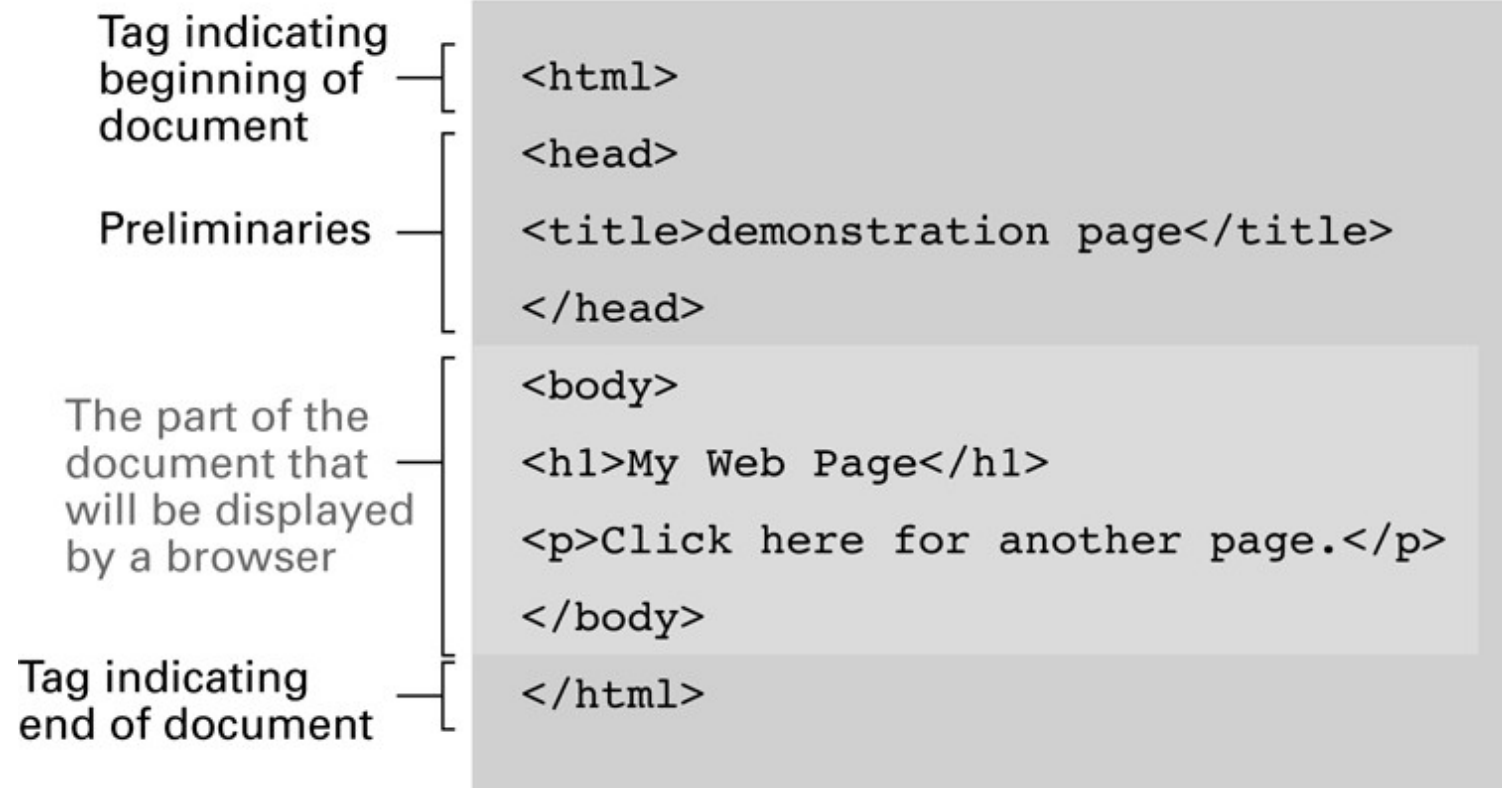
One page can require many resources



The HTML document can refer to images, styles and scripts, each obtained as a resource.

HTML describes document structure

a. The page encoded using HTML.



Elements identify content such as a heading and a paragraph; the browser interprets the markup.

The browser presents the document

b. The page as it would appear on a computer screen.



The displayed page differs from its source text. The sentence alone does not create a link.

A hyperlink in the source

a. The page encoded using HTML.

Anchor tag
containing
parameter

Closing
anchor tag

```
<html>
<head>
<title>demonstration page</title>
</head>
<body>
<h1>My Web Page</h1>
<p>Click
  <a href="http://crafty.com/demo.html">
    here
  </a>
  for another page.</p>
</body>
</html>
```

An anchor element associates visible link text with a target address.

b. The page as it would appear on a computer screen.



Following a link asks the browser to navigate to another resource.

HTML, CSS and JavaScript

Technology	Primary responsibility	Example
HTML	Content and document structure	A heading, paragraph or link
CSS	Presentation rules	Layout, colors and spacing
JavaScript	Programmed behavior	Respond to an action or request new data

These cooperate in a page. HTML is markup; it is not a C-style programming language.

A static resource can be served from stored content, such as a course reading page.

A server can also compute a response using a request and stored data, such as a timetable query.

Browser-side code can update a page after its initial load. A generated page can still contain ordinary HTML.

A service can return data

```
GET /api/readings/latest HTTP/1.1  
Host: example.org  
Accept: application/json
```

```
Example response body:  
{ "sensor": "light", "value": 512 }
```

An API defines how programs request and interpret data.

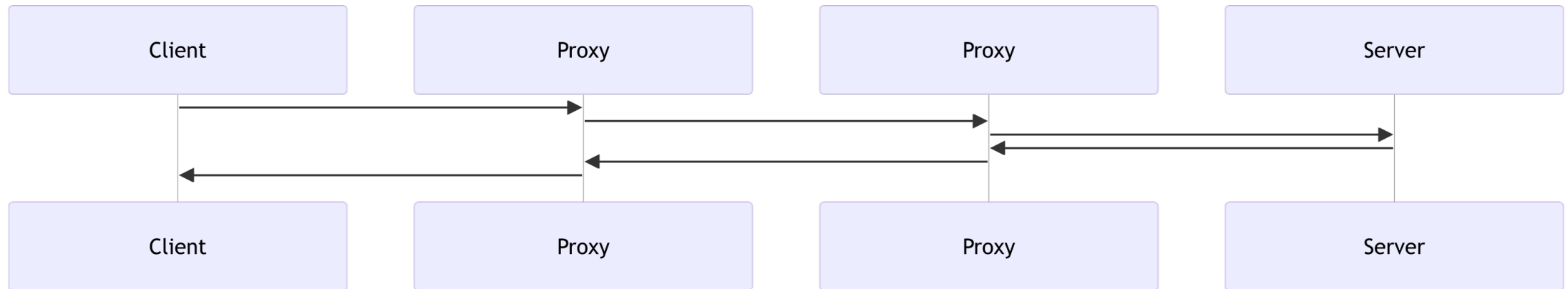
The JSON value 512 has meaning only under an agreed format and unit.

HTTP requests are individually described. Applications can use a session identifier to associate requests with one user interaction.

A server can set a cookie that the browser sends with later eligible requests.

The session data may remain on the server. A cookie is not inherently the user's password.

Intermediaries in Web communication



Proxies can relay requests and responses; some also cache content or distribute work.

A cache reuses a stored response when the applicable rules allow it, avoiding some work and delay.

A content delivery network places copies or services at multiple locations to serve users efficiently.

Reuse needs rules: a public diagram and a private account page cannot be treated identically.

Tracing a stale course page

Observation	A possible explanation
The name leads to an old server	A DNS answer may still be cached
The right server has the old HTML	The new file may not have been deployed
The server has new HTML, browser shows old	A browser or intermediary may be reusing a response
HTML is new, one image is old	Resources can have separate cache lifetimes

These are diagnostic possibilities, not proof. Inspect the relevant request and response.

Failure belongs to a stage

Symptom	Reasoning
Name cannot be resolved	Investigate the name and DNS result
Connection times out	Investigate reachability or service availability
Certificate warning	Investigate server identity and secure connection
HTTP 404	Investigate the requested resource path
Page appears, data is wrong	Investigate application logic and source data

The visible symptom narrows the investigation; it does not uniquely identify the cause.

Web concepts to carry forward

Separate the Internet infrastructure, DNS naming and HTTP resource exchange.

A displayed page can depend on multiple resources, services and cached copies.

Next week: databases organize the persistent records behind many applications.

Streams and persistent data

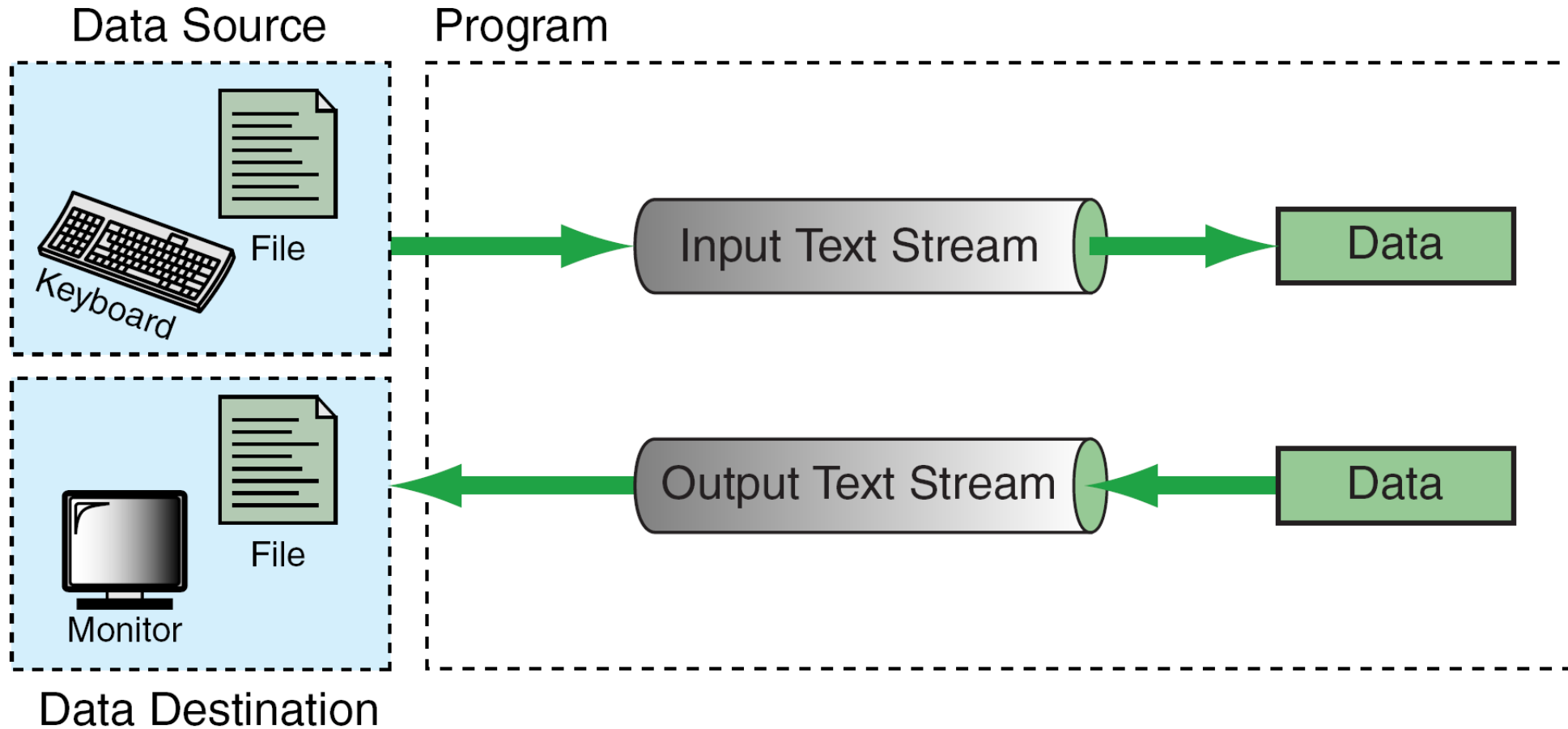
Write a small log, reopen it and verify the contents

An array belongs to a running program. A file can remain available after that program terminates.

Saving requires a representation: which values, what order, what separators and what units?

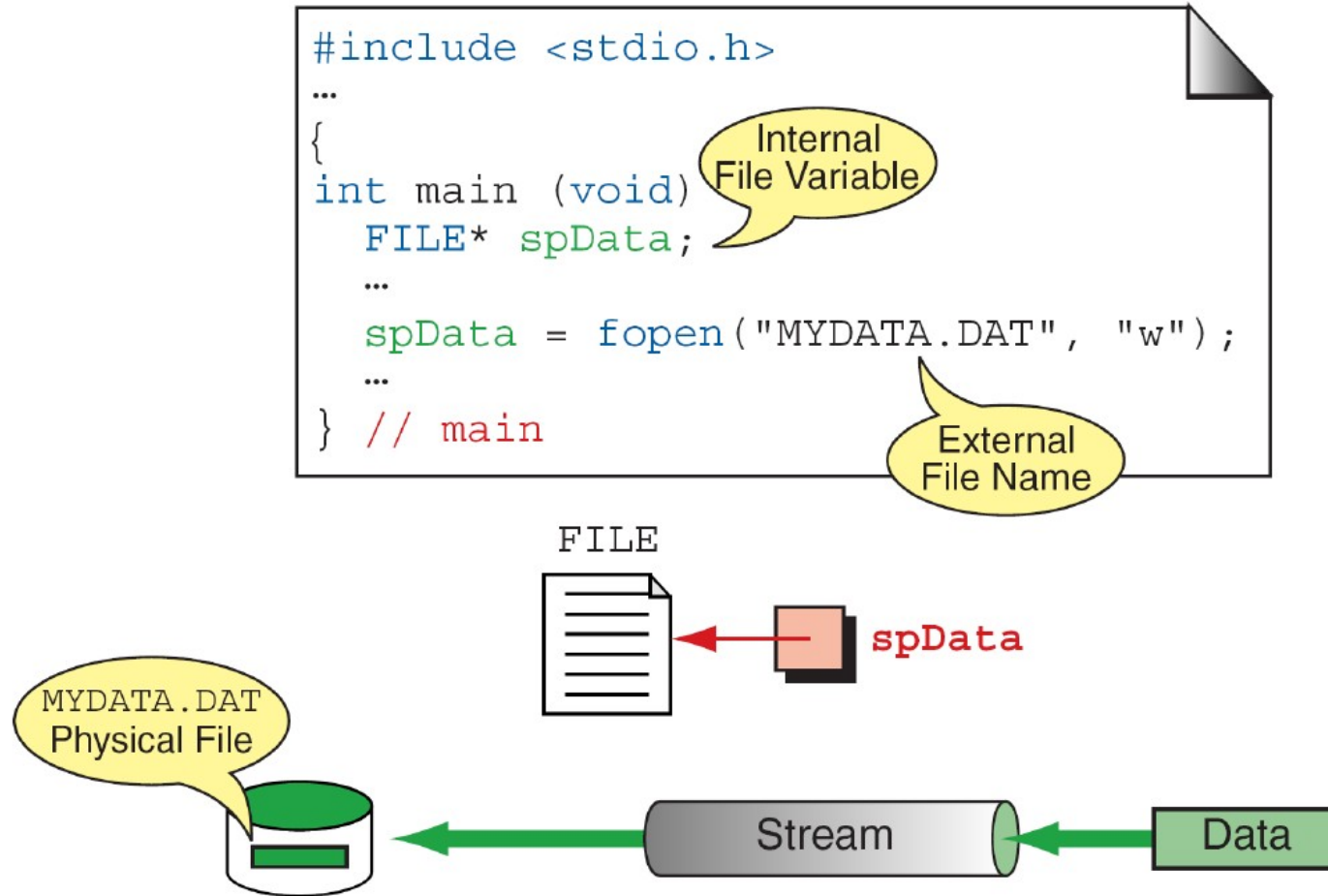
Persistence does not mean indestructibility. Files can be overwritten, truncated or damaged.

Streams connect a program to data



The same stream idea supports a file or a terminal; the direction is relative to the program.

A stream pointer and an external file



FILE * refers to the library's stream state. It does not point directly to all the bytes of the disk file.

Standard streams

Name	Usual role	Can be redirected?
stdin	Input supplied to the program	Yes
stdout	Ordinary program output	Yes
stderr	Diagnostics	Yes

printf writes to stdout; it is not restricted to the physical monitor.

A relative path such as readings.txt is interpreted from the program's current working directory.

The directory containing the source code is not automatically the working directory.

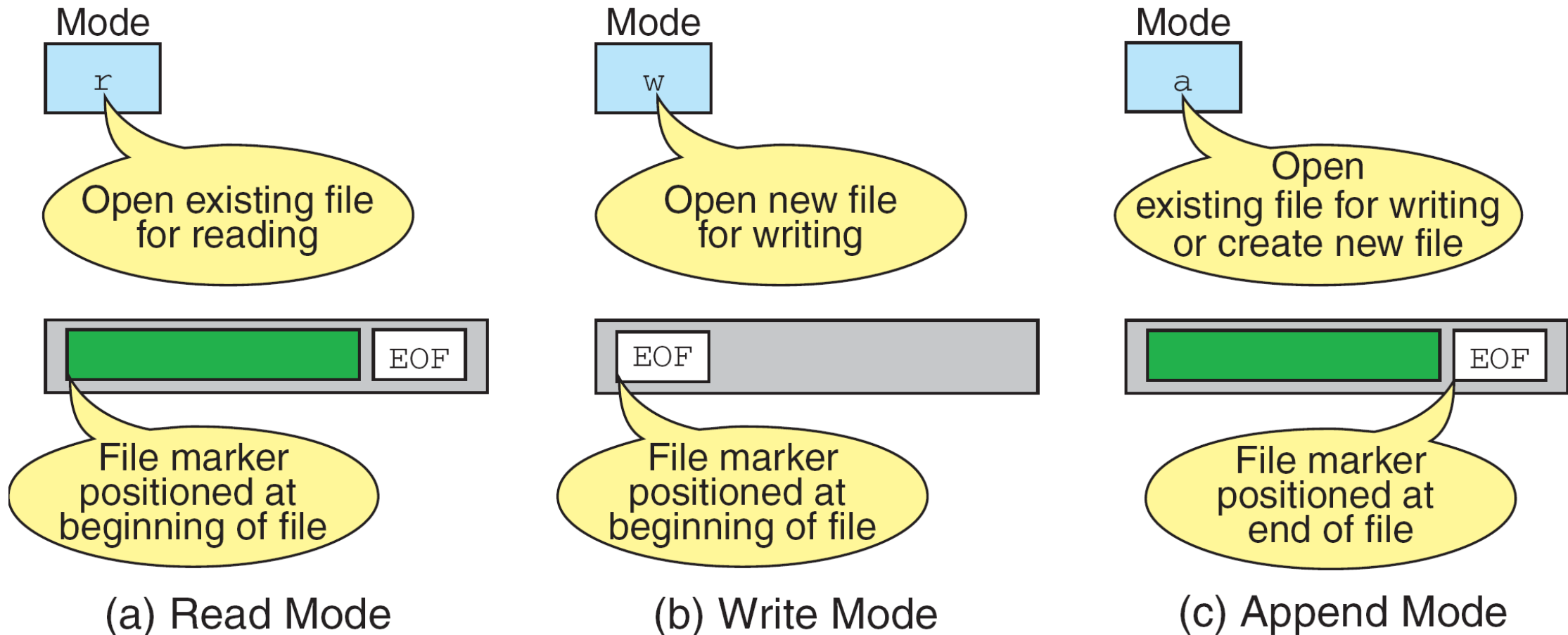
Opening can fail because the file is missing, the path is wrong, access is denied or resources are unavailable.

Choosing the open mode

Mode	Existing file	Missing file
"r"	Read from the beginning	Opening fails
"w"	Truncate to zero length, then write	Create the file
"a"	Write at the end	Create the file

Use w for a deliberate fresh output and a for extending a log. Check before discarding old data.

File position in read, write and append modes



The marker is a conceptual stream position. EOF is a condition, not a required byte stored after the data.

Opening and checking a stream

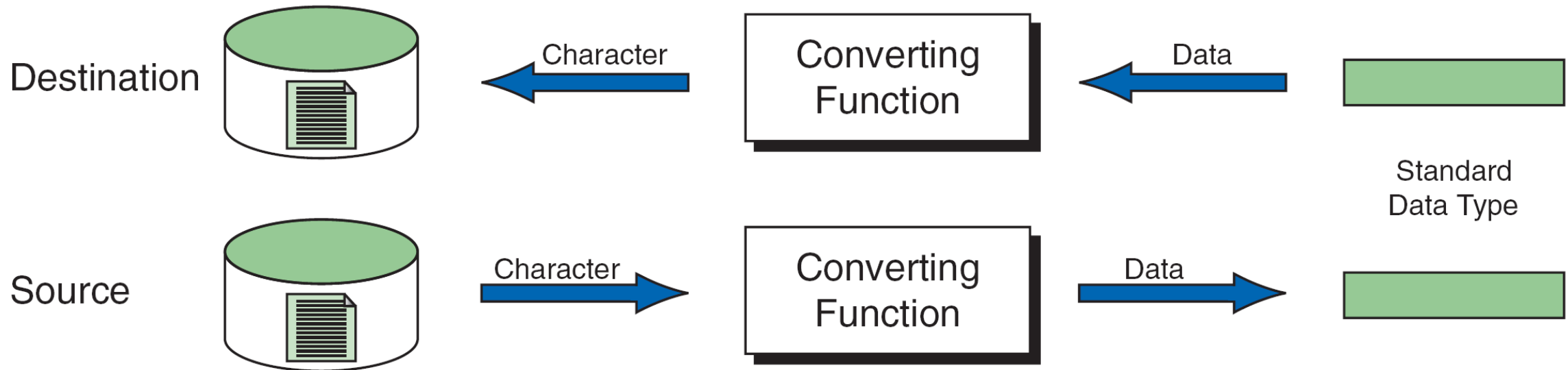
```
FILE *fp = fopen("readings.txt", "r");  
if (fp == NULL) {  
    perror("readings.txt");  
    return 1;  
}  
  
/* Use fp only after the check. */
```

Include `<stdio.h>`.

A failed open returns NULL.

This fragment belongs inside main.

Formatting values as text



Writing converts a value to characters. Reading interprets those characters under an agreed format.

Writing one reading

```
int value = 512;

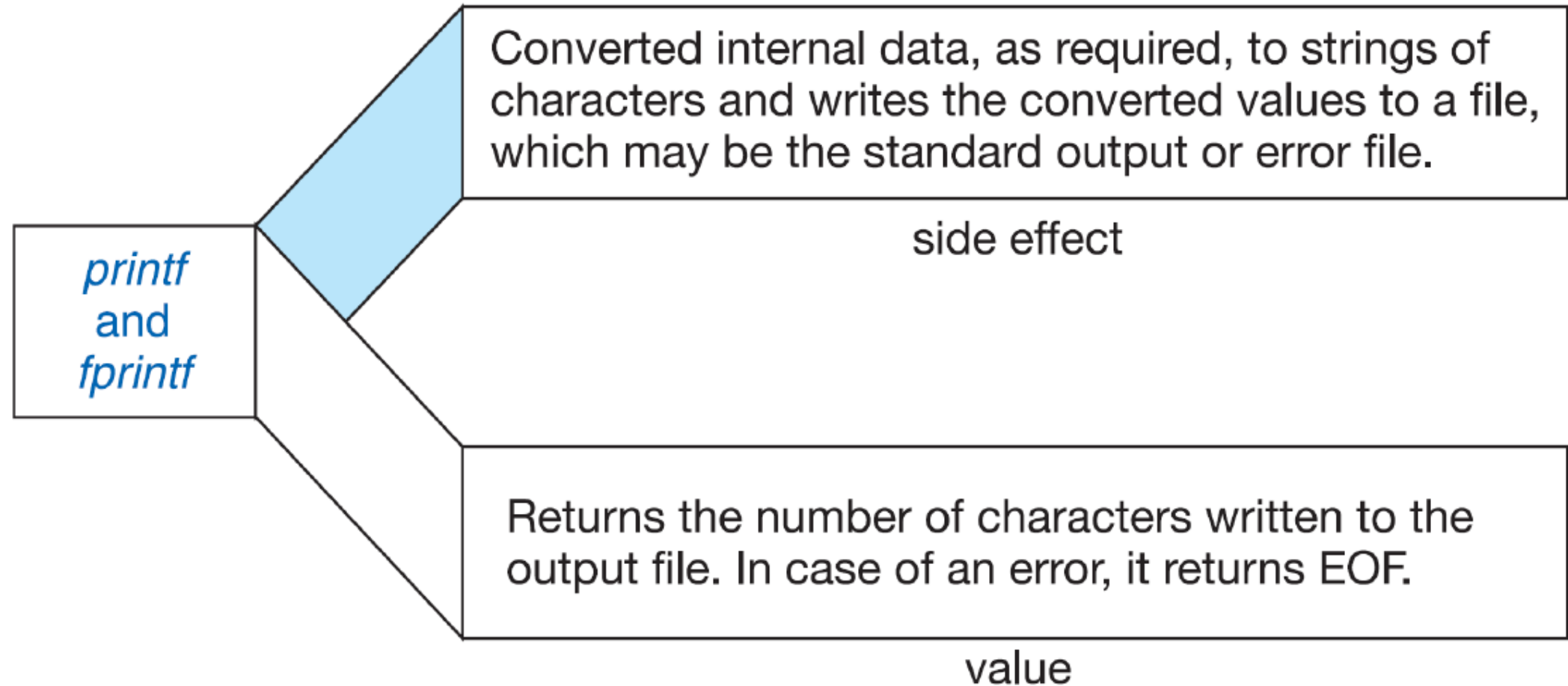
if (fprintf(fp, "%d\n", value) < 0) {
    perror("write");
    fclose(fp);
    return 1;
}
```

Assume fp is open for output.

The stored text is 512
followed by a newline.

The integer's in-memory
bytes are not copied
verbatim.

Output has a result as well as an effect



Check `fprintf` for a negative return value. Do not assume every output failure equals EOF.

Closing and reporting failure

```
if (fclose(fp) == EOF) {  
    perror("close");  
    return 1;  
}  
  
/* Do not use fp after fclose. */
```

Buffered output may fail when it is flushed during close.

Successful fclose is not a general guarantee against a later power loss.

A write and append trace

Operation	Contents after successful close
Open with w; write 512 and 640	512\n640\n
Open with a; write 700	512\n640\n700\n
Open with w again; write 900	900\n

Here \n denotes a newline, not two literal characters in the file.

Reading text into a bounded buffer

```
char line[16];  
  
while (fgets(line, sizeof line, fp) != NULL) {  
    printf("%s", line);  
}  
if (ferror(fp)) {  
    perror("read");  
}
```

Assume fp is open for input.

fgets stores at most 15 characters, then a null terminator on success.

A long line can arrive in several pieces.

A small fgets trace

Input remaining	Result in char line[5]	Next input position
512\n640\n	"512\n" plus \0	Before 6
640\n	"640\n" plus \0	At end of data
No more data	Returns NULL; no new string	End-of-file indicator set

Test the return value before using the buffer. The null terminator is added in memory.

When a line exceeds the buffer

Call with char line[5]	Characters obtained
Input: 123456\n	The complete line has seven characters
First successful call	1234 plus \0
Second successful call	56\n plus \0
Next call	NULL at end of file

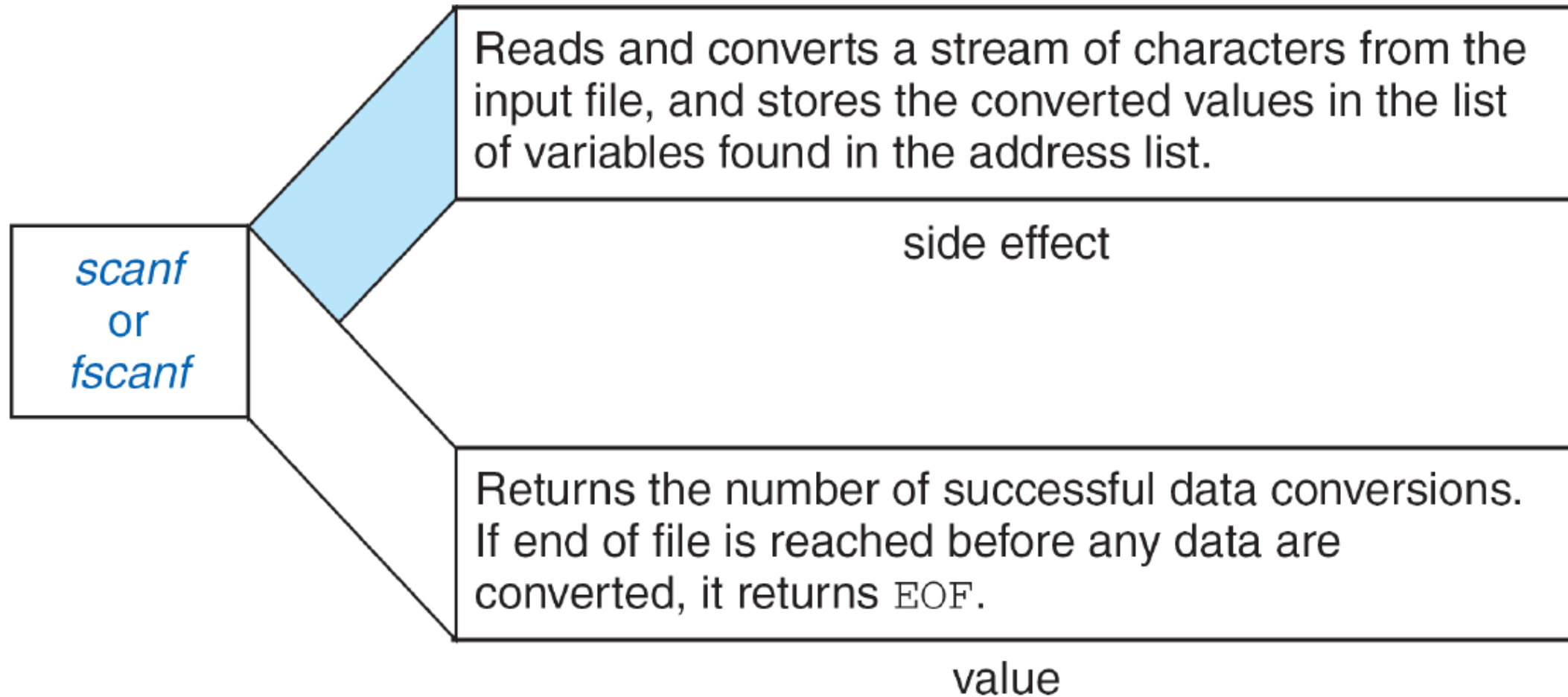
One successful fgets call does not necessarily equal one complete line.

An input operation tells us whether it obtained new data. Test that result first.

After input stops, ferror distinguishes a read error from normal exhaustion in these loops.

`while (!feof(fp))` is the wrong prediction: the end-of-file indicator is set only after a read encounters the end.

Formatted input: value and return count



For `fscanf`, count successful assignments. EOF means an input failure before the first conversion completes.

Partial conversion leaves work unfinished

```
...  
printf("\nEnter price and amount: ");  
result = scanf("%d%d", &a, &b);  
...
```

scanf aborted.
Value of a is 23!

Input stream
before

2	3		r	\n
---	---	--	---	----

Should have
been a digit

23	?	1
a	b	result

Here only the first integer is assigned. The second destination must not be treated as newly read data.

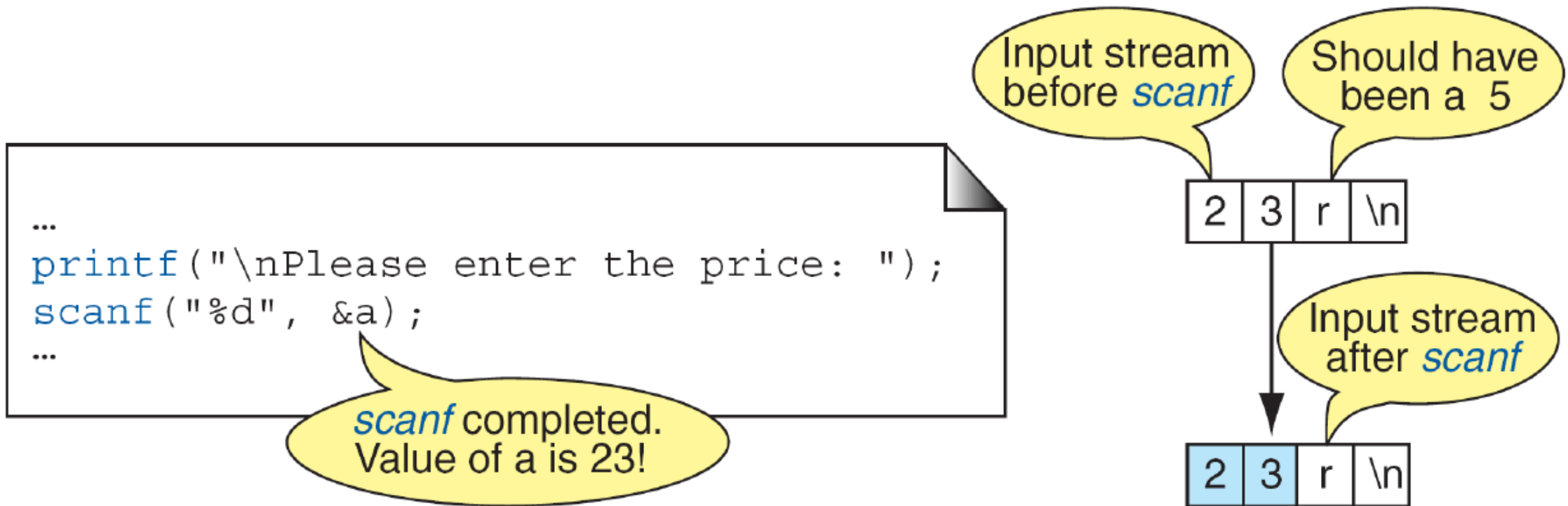
An input conversion needs an address

```
...  
printf( "\nPlease enter number of dependents: " );  
scanf ( "%4d", a );  
...
```

A missing address operator will cause your program to fail.

For an int value, %d requires int *. Omitting & causes undefined behavior, not a guaranteed diagnostic.

A numeric prefix is not a valid whole record



The prefix 23 converts successfully; the letter remains. A successful conversion need not validate the whole line.

Different fscanf outcomes

n = 2

```
n = fscanf (fp, " %d %d", &a, &b);
```

Input
Stream

1234 752 ↵

n = 1

```
n = fscanf (fp, " %d %d", &a, &b);
```

1234 F ↵

n = EOF

```
n = fscanf (fp, " %d %d", &a, &b);
```

<EOF>

Two assignments, one assignment and no completed assignment require different handling.

A controlled numeric-file loop

```
int value;  
int rc;  
while ((rc = fscanf(fp, "%d", &value)) == 1) {  
    printf("%d\n", value);  
}  
if (ferror(fp))  
    puts("Read error");  
else if (rc == 0)  
    puts("Non-integer input");
```

Assume fp is open for input.

Fixture values must fit int.

For arbitrary external input,
read bounded text and
validate the complete record.

Choosing how to read

Approach	Useful for	Important condition
fgets	Text fragments or bounded lines	Detect incomplete lines before treating them as records
fscanf	Controlled formatted tokens	Check assignment count and representable values
fgetc	Character-by-character processing	Store result in int; distinguish EOF from data

Choose the data contract first, then the function.

A small log format

Our example stores one decimal reading per line: 512, then 640, then 700.

The producer and reader agree that values are raw readings. The format does not silently invent physical units.

For a more durable interchange format, document encoding, separators, units, missing values and version changes. Structured records come next week.

Tests for a file program

Case	Expected behavior
Missing input file	Report open failure; do not read through NULL
Empty input file	Produce no readings; finish normally
Last line lacks a newline	Still process the data that was read
Long input line	Process fragments or explicitly reject a long record
Write / close failure	Report failure rather than claiming the log was saved

Also verify that append preserves old contents and w deliberately replaces them.

Lab 9: save and verify readings

Use a small fixed batch first. Write the readings, close the file, reopen it and compare the recovered content.

Arduino can supply readings through provided acquisition and transport support. The file remains on the PC.

Students implement the basic file operations themselves. No SD-card hardware or Web-server programming is required.

Check your reasoning

Why does a valid DNS answer not guarantee a webpage? The connection, HTTP request or application can still fail.

Why can a second run erase yesterday's log? Opening with w truncates an existing file.

Why must a read loop test the read result? A failed read does not supply a fresh value.

Brookshear & Brylow: Chapter 4, Internet and World Wide Web. Original Sets 5 and 14 provide the course figures.

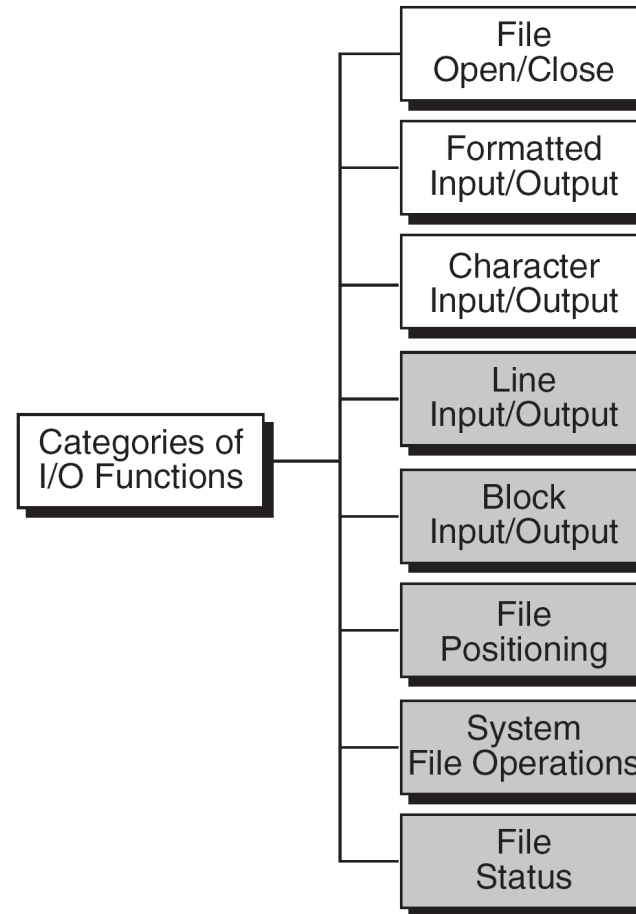
MDN: HTTP overview and cookies. IETF RFC 1034 (DNS), 3986 (URIs), 9110/9111 (HTTP and caching), 9114 (HTTP/3).

C library references: fopen, fprintf, fgets, fscanf and fclose. Follow the standard C forms used here; platform extensions are separate.

Original file-I/O illustrations

Detailed formats and longer programs for restudy

Standard I/O function families



This week uses opening/closing, formatted I/O, line input and basic error checks.

Text-file mode reference

Mode	Meaning
r	Open text file in read mode • If file exists, the marker is positioned at beginning. • If file doesn't exist, error returned.
w	Open text file in write mode • If file exists, it is erased. • If file doesn't exist, it is created.
a	Open text file in append mode • If file exists, the marker is positioned at end. • If file doesn't exist, it is created.

Read preserves data; write truncates; append writes at the end. Always check the open result.

Open and close errors: opening

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  ...
4
5  int main (void)
6  {
7  // Local Declarations
8  FILE* spTemps;
9
10 // Statements
11 ...
12
13 if ((spTemps = fopen("TEMPS.DAT", "r")) == NULL)
14 {
15     printf("\aERROR opening TEMPS.DAT\n");
16     exit (100);
17 } // if open
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Open and close errors: closing

```
18     ...
19
20     if (fclose(spTemps) == EOF)
21     {
22         printf( "\aERROR closing TEMPS.DAT\n" );
23         exit (102);
24     } // if close
25     ...
26
27 } // main
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Terminal Input/Output

```
scanf ("control string", ...);  
printf("control string", ...);
```

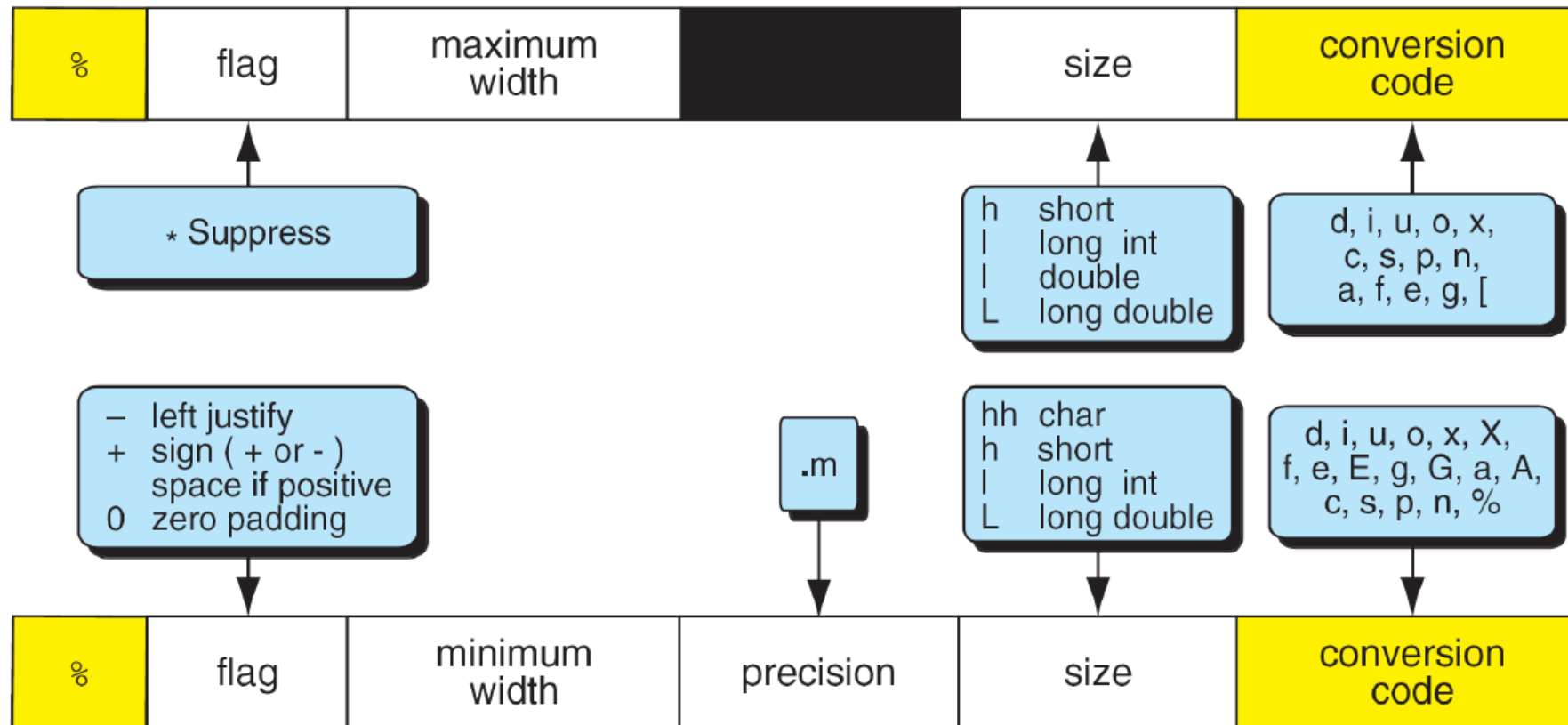
General Input/Output

```
fscanf (stream_pointer, "control string", ...);  
fprintf(stream_pointer, "control string", ...);
```

stdin and stdout can be redirected; “terminal” describes a common use, not a restriction.

Conversion specification syntax

scanf/fscanf



printf/fprintf

Reference only: use the documented conversion and argument type for the operation.

Input conversion types

Argument Type	Size Specifier	Code
integral	hh (char), h (short), none (int), l (long), ll (long long)	i
integer	h (short), none (int), l (long), ll (long long)	d
unsigned int	hh (char), h (short), none (int), l (long), ll (long long)	u
character octal	hh (unsigned char)	o
integer hexadecimal	h (short), none (int), l (long), ll (long long)	x
real	none (double), l (double), L (double)	f
real (scientific)	none (double), l (double), L (double)	e
real (scientific)	none (double), l (double), L (double)	g
real (hexadecimal)	none (double), l (double), L (double)	a

For scanf: %f takes float *, %lf double *, %Lf long double *. Input destinations are pointers.

Input conversions for characters and strings

Argument Type	Size Specifier	Code
character	none (char), l (wchar_t)	c
string	none (char string), l (wchar_t string)	s
pointer		p
integer (for count)	none (int), hh (char), h (short), l (long), ll (long long)	n
set	none (char), l (wchar_t)	[

Input destinations are pointers. For %s, limit the width and reserve space for the null terminator.

Checking input: original example

```
1  #define FLUSH while (getchar() != '\n')
2  #define ERR1 "\aPrice incorrect. Re-enter both fields\n"
3  #define ERR2 "\aAmount incorrect. Re-enter both fields\n"
4
5  // Read price and amount
6  do
7  {
8      printf("\nEnter amount and price: ");
9      ioResult = scanf("%d%f", &amount, &price);
10
11     if (ioResult != 2)
12     {
13         FLUSH;
14         if (ioResult == 1)
15             printf(ERR1);
16         else
```

The original FLUSH macro can loop forever at EOF. Stop discarding at newline OR EOF.

Checking input: continuation

```
17         printf(ERR2);  
18     } // if  
19 } while (ioResult != 2);
```

Results:

Enter amount and price: ? 15.25
Amount incorrect. Re-enter both fields

Enter amount and price: 100 ?
Price incorrect. Re-enter both fields

Enter amount and price: 100 15.25

The original FLUSH macro can loop forever at EOF. Stop discarding at newline OR EOF.

Output conversion types

Argument Type	Flag	Size Specifier	code
integer	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	d, i
unsigned integer	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	u
integer (octal)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	o
integer (hex)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	x, X
real	-, +, 0, #, space	none (double), l (double), L (double)	f
real (scientific)	-, +, 0, #, space	none (double), l (double), L (double)	e, E
real (scientific)	-, +, 0, #, space	none (double), l (double), L (double)	g, G

For printf: %f takes double; %Lf takes long double. Sign flags + and space do not apply to unsigned conversions.

Output conversions: continuation

Argument Type	Flag	Size Specifier	code
real (hexadecimal)	-, +, 0, #, space	none (double), l (double), L (double)	a, A
character	-	none (char), l (w-char)	c
string	-	none (char string), l (w-char string)	s
pointer			p
integer (for count)		none (int), h (short), l (long)	n
to print %			%

For printf: %La takes long double; %c takes int; %p takes void *. The %n destination is a pointer.

Output formatting flags

Flag Type	Flag Code	Formatting
Justification	none	right justified
	–	left justified
Padding	none	space padding
	0	zero padding
Sign	none	positive value: no sign negative value: –
	+	positive value: + negative value: –
	space	positive value: space negative value: –
Alternate	#	print alternative format for scientific, hexadecimal, and octal.

Reference only: use the documented conversion and argument type for the operation.

Reading integers: declarations and open

```
1  /* Read a text file of integers, and print the contents.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main (void)
9  {
10     // Local Declarations
11     FILE* spIn;
12     int    numIn;
13
14     // Statements
15     spIn = fopen("P07-03.DAT", "r");
16     if (!spIn)
17     {
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Reading integers: loop and output

```
18     printf("Could not open file\a\n");
19     exit  (101);
20 } // if open fail
21
22 while ((fscanf(spIn, "%d", &numIn)) == 1)
23     printf("%d ", numIn);
24
25 return 0;
26 } // main
```

Results:

1 2 3 4 5 6 7 8 9 10

A stopped conversion can mean malformed input. Check ferror and the assignment count; check writes too.

Copying integers: declarations

```
1  /* Copy a text file of integers.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main (void)
9  {
10     // Local Declarations
11     FILE* spIn;
12     FILE* spOut;
13     int    numIn;
14     int    closeResult;
15
16     // Statements
17     printf("Running file copy\n");
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Copying integers: input and output

```
19     if (!spIn)
20     {
21         printf("Could not open input file\n");
22         exit (101);
23     } // if open fail
24
25     spOut = fopen("P07-04.DAT", "w");
26     if (!spOut)
27     {
28         printf("Could not open output file\n");
29         exit (102);
30     } // if open fail
31
32     while ((fscanf(spIn, "%d", &numIn)) == 1)
33         fprintf(spOut, "%d\n", numIn);
34
35     closeResult = fclose(spOut);
```

A stopped conversion can mean malformed input. Check ferror and the assignment count; check writes too.

Copying integers: completion

```
36     if (closeResult == EOF)
37     {
38         printf("Could not close output file\n");
39         exit  (201);
40     } // if close fail
41
42     printf("File copy complete\n");
43     return 0;
44 } // main
```

Results:

```
Running file copy
File copy complete
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Appending integers: opening

```
1  /* Copy a text file of integers.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main (void)
9  {
10     // Local Declarations
11     FILE* spAppnd;
12     int    numIn;
13     int    closeResult;
14
15     // Statements
16     printf("This program appends data to a file\n");
17     spAppnd = fopen("P07-05.DAT", "a");
18     if (!spAppnd)
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Appending integers: loop

```
19     {
20         printf("Could not open input file\a\n");
21         exit  (101);
22     } // if open fail
23
24     printf("Please enter first number: ");
25     while ((scanf("%d", &numIn)) != EOF)
26     {
27         fprintf(spAppnd, "%d\n", numIn);
28         printf("Enter next number or <EOF>: ");
29     } // while
30
31     closeResult = fclose(spAppnd);
32     if (closeResult == EOF)
33     {
34         printf("Could not close output file\a\n");
35         exit  (201);
36     } // if close fail
```

Replace != EOF with == 1: a nonnumeric token returns 0 and otherwise makes this loop repeat forever.

Appending integers: output

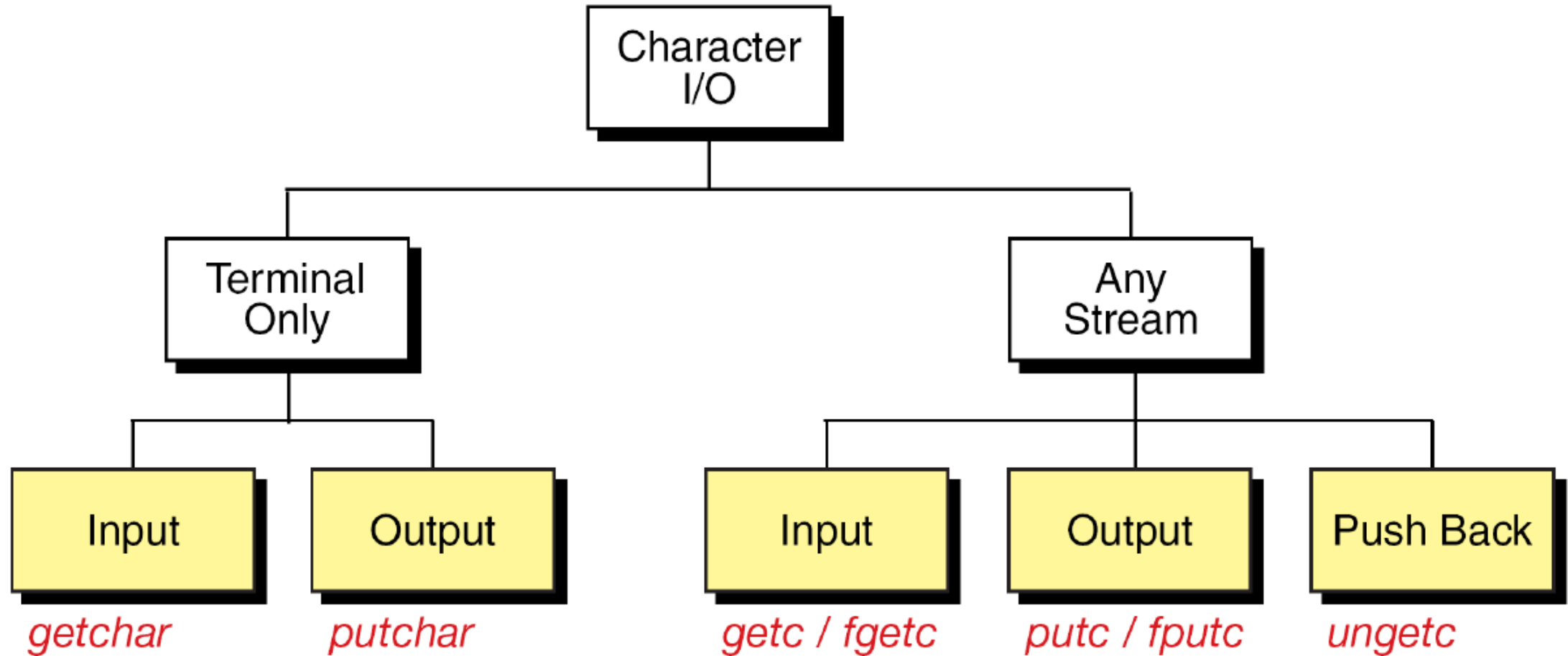
```
37  
38     printf("\nFile append complete\n");  
39     return 0;  
40 } // main
```

Results:

```
This program appends data to a file  
Please enter first number: 1  
Enter next number or <EOF>: 2  
Enter next number or <EOF>: 3  
Enter next number or <EOF>: ^d  
File append complete
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Character I/O function families



stdin and stdout can be redirected; “terminal” describes a common use, not a restriction.

Creating a text file: opening

```
1  /* This program creates a text file.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  int main (void)
7  {
8      // Local Declarations
9      FILE* spText;
10     int    c;
11     int    closeStatus;
12
13     // Statements
14     printf("This program copies input to a file.\n");
15     printf("When you are through, enter <EOF>.\n\n");
16
17     if (!(spText = fopen("P07-07.DAT", "w")))
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Creating a text file: character loop

```
18     {
19         printf("Error opening P07.07.DAT for writing");
20         return (1);
21     } // if open
22
23     while ((c = getchar()) != EOF)
24         fputc(c, spText);
25
26     closeStatus = fclose(spText);
27     if (closeStatus == EOF)
28     {
```

Store fgetc/getchar results in int. Check write failure and distinguish read error from normal EOF.

Results:

```
This program copies input to a file.  
When you are through, enter <EOF>.
```

```
Now is the time for all good students  
To come to the aid of their college.^d
```

```
Your file is complete
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Copying text: opening

```
1  /* This program copies one text file into another.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  int main (void)
7  {
8      // Local Declarations
9      int    c;
10     int    closeStatus;
11     FILE*  sp1;
12     FILE*  sp2;
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Copying text: loop and close

```
21         } // if open input
22     if (!(sp2 = fopen ("P07-08.DAT", "w")))
23     {
24         printf("Error opening P07-08.DAT for writing");
25         return (2);
26     } // if open output
27
28     while ((c = fgetc(sp1)) != EOF)
29         fputc(c, sp2);
30
31     fclose(sp1);
32     closeStatus = fclose(sp2);
33     if (closeStatus == EOF)
34     {
35         printf("File close error.\a\n");
36         return 201;
37     } // if close error
38     printf("File successfully created\n");
39     return 0;
40 } // main
```

Store fgetc/getchar results in int. Check write failure and distinguish read error from normal EOF.

Counting characters and newlines: setup

```
1  /* This program counts characters and lines in a program.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  int main (void)
7  {
8      // Local Declarations
9      int    curCh;
10     int    preCh;
11     int    countLn = 0;
12     int    countCh = 0;
13     FILE*  sp1;
14
15     // Statements
16     if (!(sp1 = fopen("P07-07.DAT", "r")))
17     {
18         printf("Error opening P07-07.DAT for reading");
19         return (1);
20     } // if open error
21
```

Reference example. Apply the open, read, write and close checks developed in this lecture.

Counting characters and newlines: loop

```
22     while ((curCh = fgetc(sp1)) != EOF)
23     {
24         if (curCh != '\n')
25             countCh++;
26         else
27             countLn++;
28         preCh = curCh;
29     } // while
30
31     if (preCh != '\n')
32         countLn++;
33
34     printf("Number of characters: %d\n", countCh);
35     printf("Number of lines      : %d\n", countLn);
36     fclose(sp1);
37     return 0;
38 } // main
```

Results:

Number of characters: 74
Number of lines: 2

Initialize preCh and track whether any character was read: the final test otherwise uses an unset value for an empty file.

Counting words: setup

```
1  /* Count number of words in file. Words are separated by
2     whitespace characters: space, tab, and newline.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7
8  #define WHT_SPC\
9      (cur == ' ' || cur == '\n' || cur == '\t')
10 int main (void)
11 {
12     //Local Declarations
13     int    cur;
14     int    countWd = 0;
15     char   word = '0';          // 0 out of word: 1 in word
16     FILE*  sp1;
17
18     // Statements
19     if (!(sp1 = fopen("P07-07.DAT", "r")))
20     {
21         printf("Error opening P07-07.DAT for reading");
```

The example defines a word using spaces, tabs and newlines; it is not a linguistic word counter.

Counting words: state changes

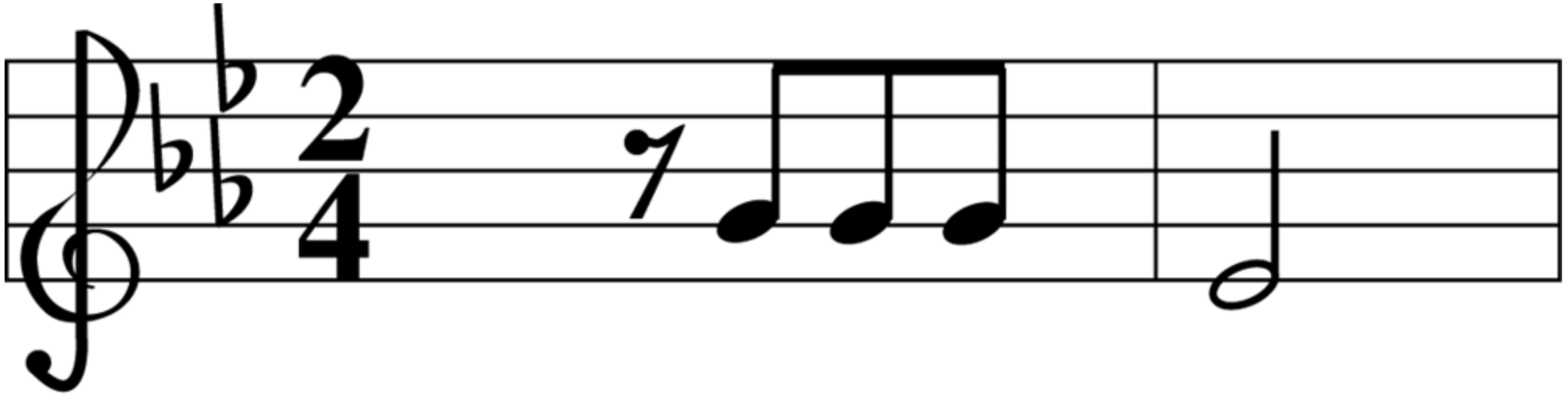
```
22         return (1);
23     } // if file open error
24     while ((cur = fgetc(sp1)) != EOF)
25     {
26         if (WHT_SPC)
27             word = 'O';
28         else
29             if (word == 'O')
30             {
31                 countWd++;
32                 word = 'I';
33             } // else
34     } // while
35     printf("The number of words is: %d\n", countWd);
36
37     fclose(sp1);
38     return 0;
39 } // main
```

Results:

The number of words is: 15

The example defines a word using spaces, tabs and newlines; it is not a linguistic word counter.

Symbols and their interpretation



Musical notation describes an action to perform, just as a data format requires an agreed interpretation.