

Databases and records in C

Organizing shared information

C: structs, record arrays and persistence

Prof. Dr. Mehmet Göktürk · Gebze Technical University

Who has the sensor kit?



Which kit is available?

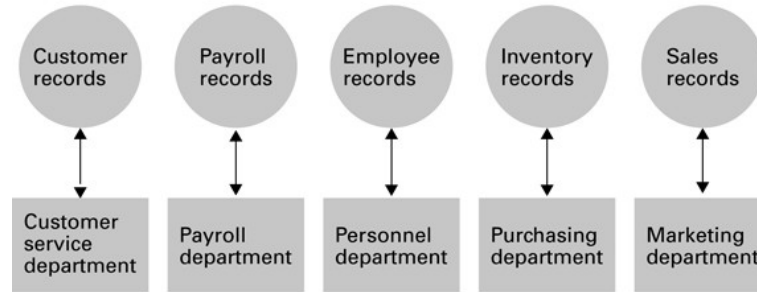
Who has borrowed it?

What happens if two people reserve it together?

A fictional equipment desk. Several people need a consistent view of the same facts.

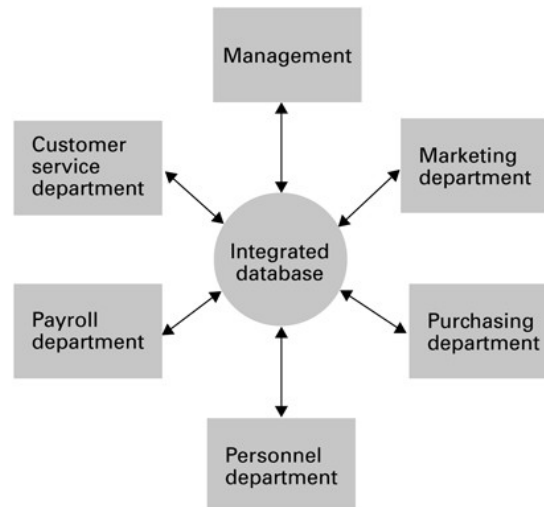
Separate files and shared information

a. File-oriented information system



Separate copies can disagree.

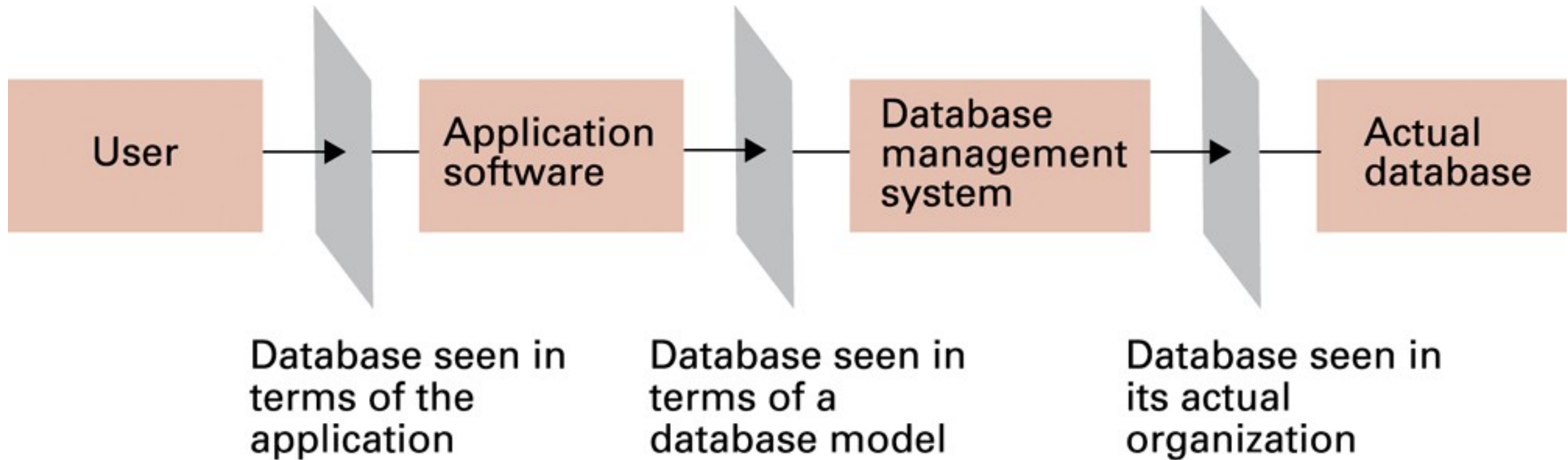
b. Database-oriented information system



Shared data needs rules for changes and access.

A database can support several views of related information. Its storage still uses files.

The database management system



The DBMS interprets requests and manages storage. Applications do not need to know every disk location.

What a DBMS adds

A shared description of records, their relationships and valid values.

Queries retrieve information while transactions coordinate changes.

Permissions, recovery and indexes support dependable access. A CSV file alone does not provide these services.

A small measurement database

Device	Room
7	Lab A
9	Lab B

One row describes one device. All example records in this lesson are fictional.

Readings belong to devices

Reading ID	Device ID	Raw value
101	7	512
102	9	640
103	7	700

One device can have many readings. Here raw values use an assumed 10-bit range of 0–1023.

Rows, columns and a schema

Term	Meaning in this example
Row / tuple	One reading, such as (101, 7, 512)
Column / attribute	A named property such as raw value
Schema	Field names, types, keys and constraints
Instance	The rows currently stored

Changing a value changes the instance. Adding a field changes the schema.

A relation in the original material

Empl Id	Name	Address	SSN
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555
•	•	•	•
•	•	•	•
•	•	•	•

Illustrative textbook records. Row order is not an identity and SQL does not guarantee an order without ORDER BY.

Keys identify records

Key	Example	Reason
Primary key	Reading ID = 101	Identifies one reading
Foreign key	Device ID = 7	Refers to an existing device
Candidate key	A minimal set of unique fields	May be chosen as the primary key

A primary key is unique and non-null. A foreign key can repeat in the readings table.

Device 7 can be referenced by readings 101 and 103. This is a one-to-many relationship.

A many-to-many relationship needs an association table, such as students borrowing different kits over time.

A key identifies an entity. An array index identifies a current position. Moving a row should not change its identity.

Repeated facts can disagree

Empl Id	Name	Address	SSN	Job Id	Job Title	Skill Code	Dept	Start Date	Term Date
25X15	Joe E. Baker	33 Nowhere St.	111223333	F5	Floor manager	FM3	Sales	9-1-2007	9-30-2008
25X15	Joe E. Baker	33 Nowhere St.	111223333	D7	Dept. head	K2	Sales	10-1-2008	*
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999	F5	Floor manager	FM3	Sales	10-1-2007	*
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555	S25X	Secretary	T5	Personnel	3-1-1999	4-30-2006
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555	S26Z	Secretary	T6	Accounting	5-1-2006	*
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.

The same employee details occur in several assignment rows. An address update must reach every copy.

Separating entities and assignments

EMPLOYEE relation

Empl Id	Name	Address	SSN
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555

JOB relation

Job Id	JobTitle	Skill Code	Dept
S25X	Secretary	T5	Personnel
S26Z	Secretary	T6	Accounting
F5	Floor manager	FM3	Sales
.	.	.	.
.	.	.	.
.	.	.	.

ASSIGNMENT relation

Empl Id	Job Id	Start Date	Term Date
23Y34	S25X	3-1-1999	4-30-2006
34Y70	F5	10-1-2007	*
23Y34	S26Z	5-1-2006	*
.	.	.	.
.	.	.	.
.	.	.	.

Employee details, job details and assignments have different identities. The keys reconnect the tables.

Following a relationship

EMPLOYEE relation

Empl Id	Name	Address	SSN
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cheryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555
.	.	.	.
.	.	.	.
.	.	.	.

JOB relation

Job Id	Job Title	Skill Code	Dept
S25X	Secretary	T5	Personnel
S26Z	Secretary	T6	Accounting
F5	Floor manager	FM3	Sales
.	.	.	.
.	.	.	.
.	.	.	.

are contained
in the personnel
and accounting
departments.

ASSIGNMENT relation

Empl Id	Job Id	Start Date	Term Date
23Y34	S25X	3-1-1999	4-30-2006
34Y70	F5	10-1-2007	*
23Y34	S26Z	5-1-2006	*
.	.	.	.
.	.	.	.
.	.	.	.

The jobs held
by employee
23Y34

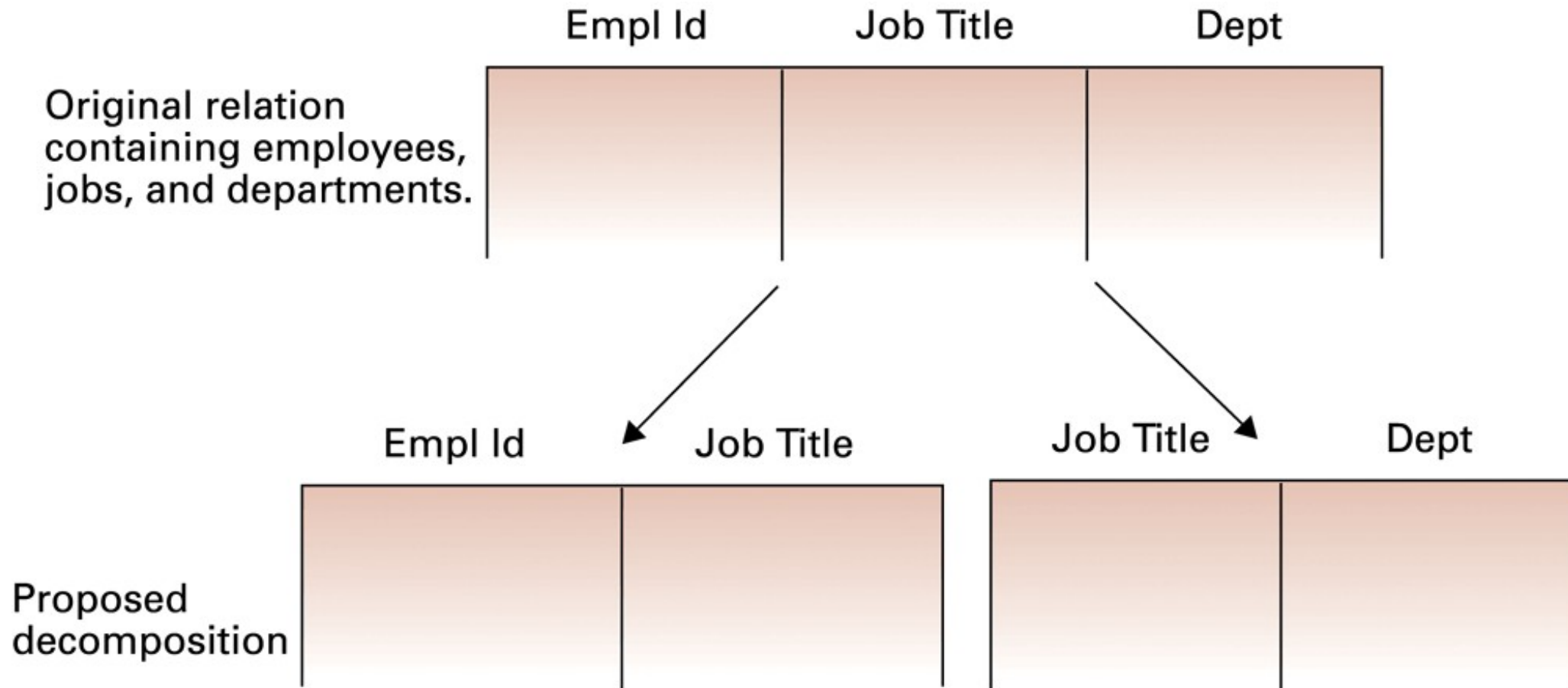
Follow employee 23Y34 through ASSIGNMENT to JOB to find the departments. Historical “*” marks a continuing assignment.

Why the split helps

Problem	Combined table	Separate device table
Update	Change a room in every reading	Change one device row
Insert	A device may need a fake reading	Store the device before any reading
Delete	Last reading may erase device facts	Deleting readings preserves the device

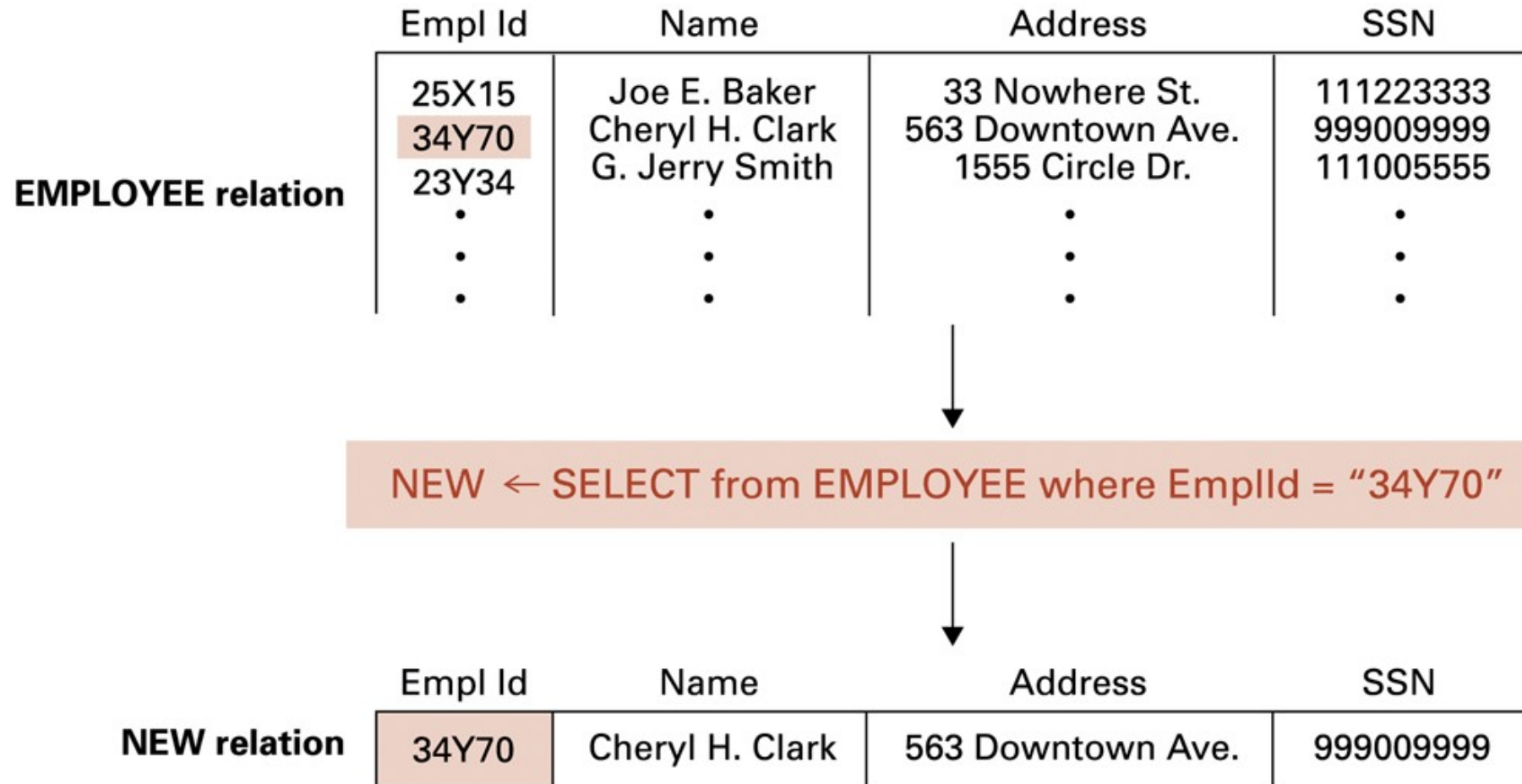
Decomposition must preserve the relationships needed to reconstruct the intended information.

A split can lose information



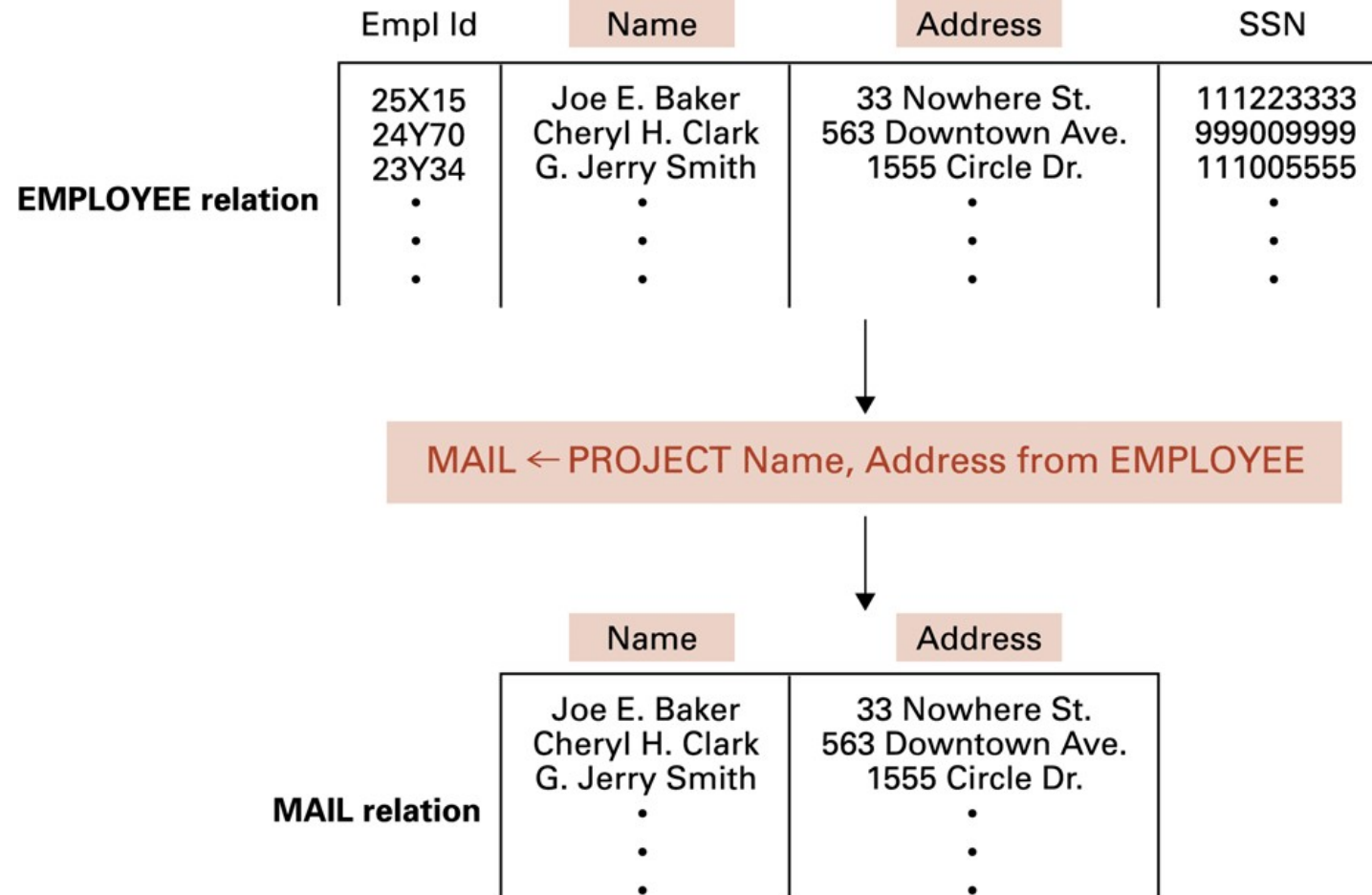
If a job title occurs in several departments, joining on title alone can invent employee–department pairs.

Selection chooses rows



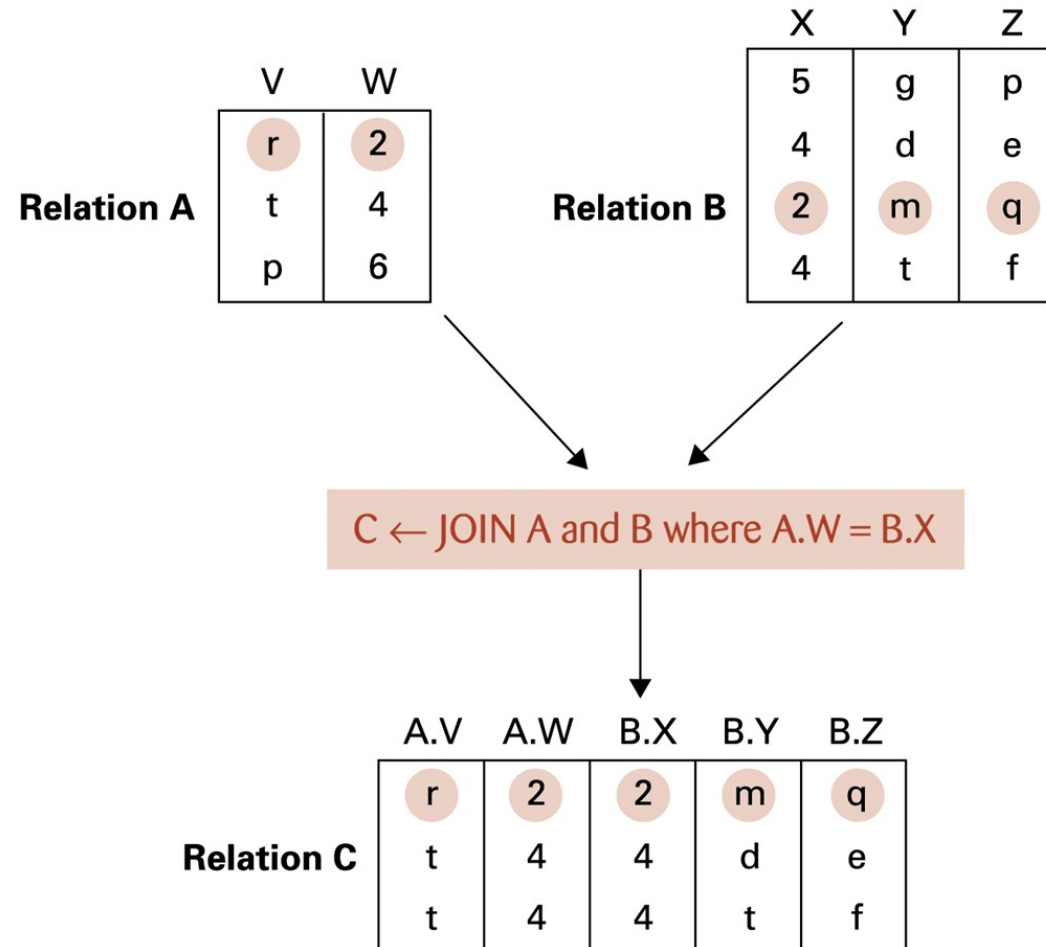
Relational selection filters rows. SQL expresses this condition with WHERE.

Projection chooses columns



Relational projection chooses attributes. SQL SELECT lists columns; DISTINCT is needed when duplicate removal is intended.

A join matches rows across tables



Here $A.W = B.X$. Two matches for key 4 produce two result rows. Joins need a stated condition.

A query describes the wanted result

```
SELECT reading_id, raw  
FROM readings  
WHERE raw >= 600  
ORDER BY reading_id;
```

Result: (102, 640) and (103, 700).

The DBMS chooses how to retrieve the rows.

SQL is a declarative query language.

Joining devices and readings

```
SELECT r.reading_id, d.room, r.raw  
FROM readings AS r  
JOIN devices AS d  
    ON r.device_id = d.device_id  
WHERE r.raw >= 600  
ORDER BY r.reading_id;
```

102 Lab B 640

103 Lab A 700

Each reading receives the
room of its matching device.

Rules protect the data

Rule	Accepted	Rejected
Unique reading ID	A new ID, 104	A second row with ID 101
Existing device	Device ID 7	Device ID 99, if absent
Value range	Raw = 640	Raw = 1200
Required value	A measured integer	NULL where NOT NULL is required

A type alone cannot express every rule. Constraints encode additional conditions.

Missing is different from zero

NULL represents missing or unknown information in SQL. A zero measurement is a known value.

Use IS NULL to test for missing data. Comparing with = NULL does not behave like an ordinary equality test.

A device that has never reported should not acquire a fabricated zero reading.

A transaction groups related changes

Stage	Kit record	Loan record
Before	Available	No open loan
Incomplete change	Borrowed	Still no open loan
Successful transaction	Borrowed	Open loan stored
Rollback	Available	No open loan

Atomicity means all intended changes commit together or none take effect.

Concurrent updates can lose work

Step	Clerk A	Clerk B
Read stock	Reads 5	Reads 5
Prepare change	Subtracts 1: writes 4	Subtracts 1: writes 4
Incorrect final value	4	Expected after two loans: 3

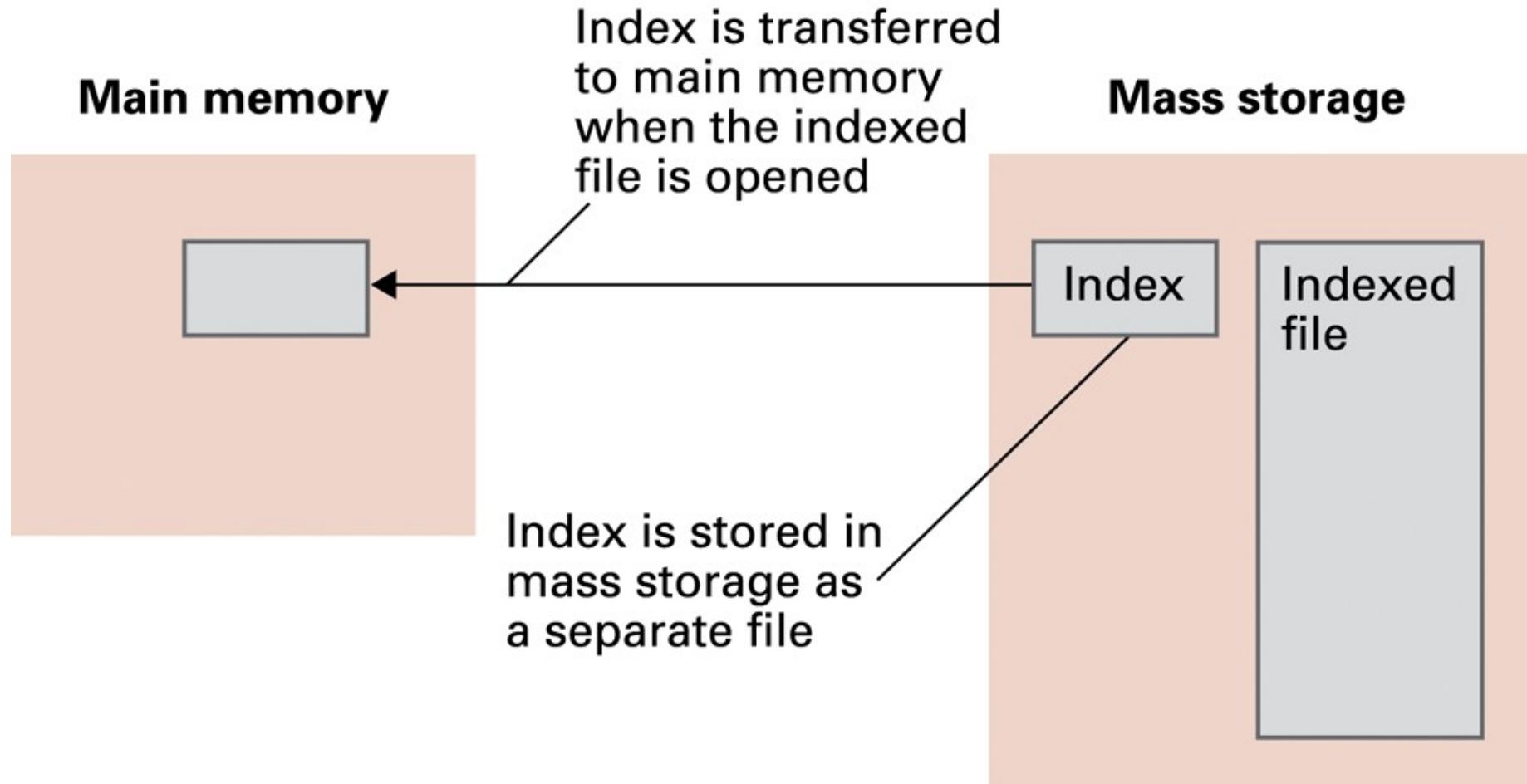
A suitable transaction/isolation strategy must coordinate competing changes. BEGIN alone does not fix every race.

A failure may happen between two writes or just after an application receives a success response.

Transaction logging helps a DBMS recover committed changes and discard incomplete work.

Backups serve a different purpose. They help recover from lost files or mistaken changes beyond the current transaction log.

Indexes locate records



This is a simple historical model. Modern DBMSs can cache parts of an index rather than loading the whole index at open.

Indexes have a cost

An index can reduce the search needed for some queries. It also occupies space and must follow changes to the data.

A suitable index may support a key lookup or a range query. The database can still choose a scan.

This connects to Week 8: the structure and the question determine the work required.

Views and access permissions

A view can expose selected columns or rows for a particular purpose.

Permissions must control access to the underlying data as well. A filtered screen alone is not a security boundary.

A device operator may need live measurements without access to borrower contact details.

Combining data changes what it reveals

A room reading can become sensitive when combined with a device location and a timetable.

A useful query can also create an unexpected profile of someone's behavior.

Choose what to collect, who can access it and how long it is needed. Technical access and appropriate use are separate questions.

Check: what should be rejected?

A new reading has ID 101, device 99 and raw value 1200. Existing devices are 7 and 9.

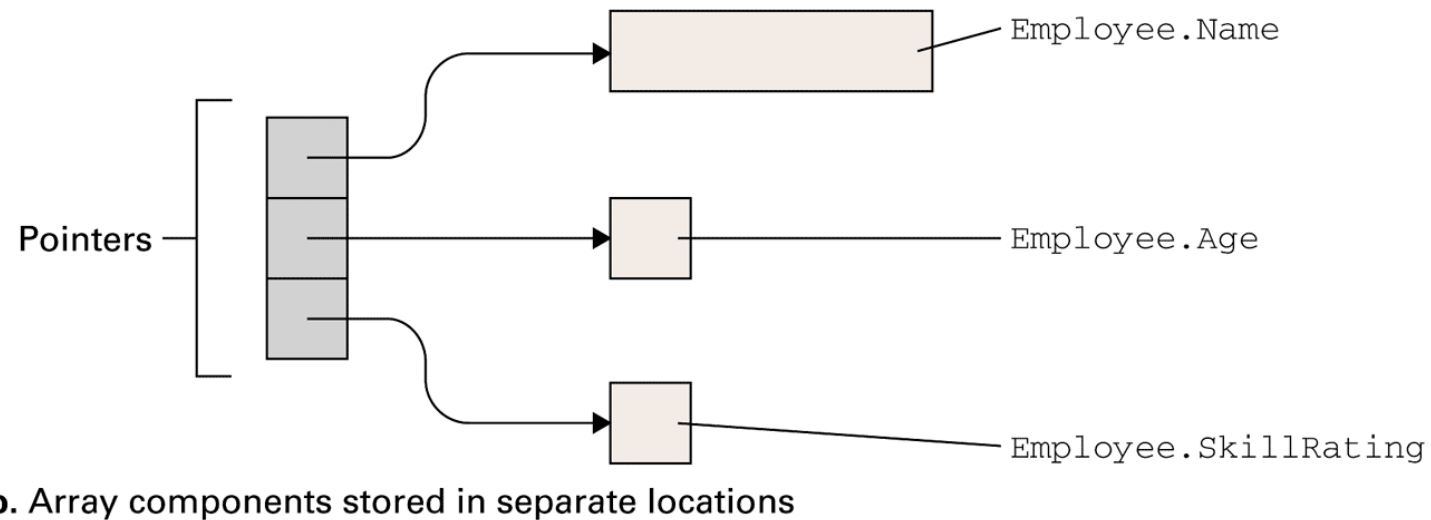
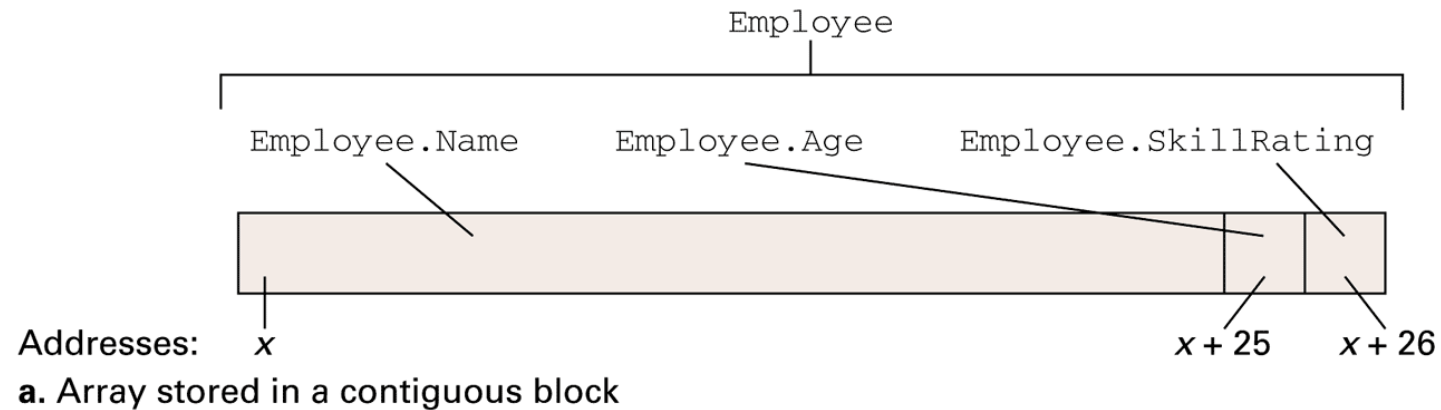
Find each violated rule independently. Correcting just one field does not make the record valid.

Answer: duplicate primary key, missing referenced device and out-of-range raw value.

One object, several fields

The program keeps a bounded array of records
The file preserves their values between runs

A record groups different fields



Brookshear's "heterogeneous array" is a general record concept. A C array has one element type; a struct can mix member types.

A C structure type

```
struct Reading {  
    int id;  
    int device_id;  
    int raw;  
};
```

```
struct Reading r = {101, 7, 512};
```

The declaration defines a type.

r is one object of that type.

Each field has a name and a type.

Accessing a field

```
printf("%d\n", r.raw);  
r.raw = 600;  
printf("%d %d\n", r.id, r.raw);
```

The dot selects a member of an object.

Output: 512, then 101 600.

Changing raw leaves id unchanged.

A typedef is an alternative name

```
typedef struct {  
    int id;  
    int device_id;  
    int raw;  
} Reading;
```

```
Reading r = {101, 7, 512};
```

Later examples use Reading.

This names the same kind of record.

The alias does not allocate an object by itself.

An array of records

```
Reading rows[4] = {  
    {101, 7, 512},  
    {102, 9, 640},  
    {103, 7, 700}  
};  
size_t count = 3;
```

Capacity is 4. Three records
are active.

rows[1].raw is 640.

Use indices below count for
active records.

Identity and position

Array index	Record ID	Raw
0	101	512
1	102	640
2	103	700

Searching for ID 102 returns index 1. ID 102 is not a valid array index here.

A bounded search

```
size_t find_id(const Reading a[],  
               size_t n, int id)  
{  
    for (size_t i = 0; i < n; ++i)  
        if (a[i].id == id)  
            return i;  
    return n;  
}
```

n means “not found”.

The caller must check before indexing.

Requires n accessible records.

Updating one matching record

```
size_t i = find_id(rows, count, 102);  
if (i < count) {  
    rows[i].raw = 650;  
}  
else {  
    puts("ID not found");  
}
```

Only the matching row changes.

Range validation belongs before the assignment.

The ID stays 102.

A pointer to a record

```
Reading *p = &rows[1];  
p->raw = 650;
```

```
/* Equivalent member access: */  
(*p).raw = 650;
```

The arrow combines
dereference and member
selection.

p points to a live record.

Parentheses matter in
(*p).raw.

Copying and sharing

```
Reading copy = rows[1];  
copy.raw = 900;
```

```
Reading *alias = &rows[1];  
alias->raw = 650;
```

The struct assignment copies member values.

Changing copy does not change rows[1].

An alias accesses the original object.

Array members and pointer members

A `char name[16]` member stores its characters inside the struct. Struct assignment copies that array too.

A `char *name` member stores a pointer. Struct assignment copies the address, so both records refer to the same characters.

The pointed-to object needs a valid lifetime. Saving the address to a file does not save the referenced text.

Memory layout is implementation-dependent

Property	What C guarantees / requires
Member order	Non-bit-field members follow declaration order
Padding	Extra bytes may occur between members and at the end
Object size	Use sizeof(Reading), not an assumed sum
Equality	Compare the meaningful fields, not padding bytes

A memory representation is not automatically a portable storage format.

Searching sorted records: a brief recap

Low...high indices	Middle	Compare target ID 103
0...2	1: ID 102	103 is larger; keep index 2
2...2	2: ID 103	Found

Binary search requires records sorted by the same key. For a small unsorted array, a bounded scan is simpler.

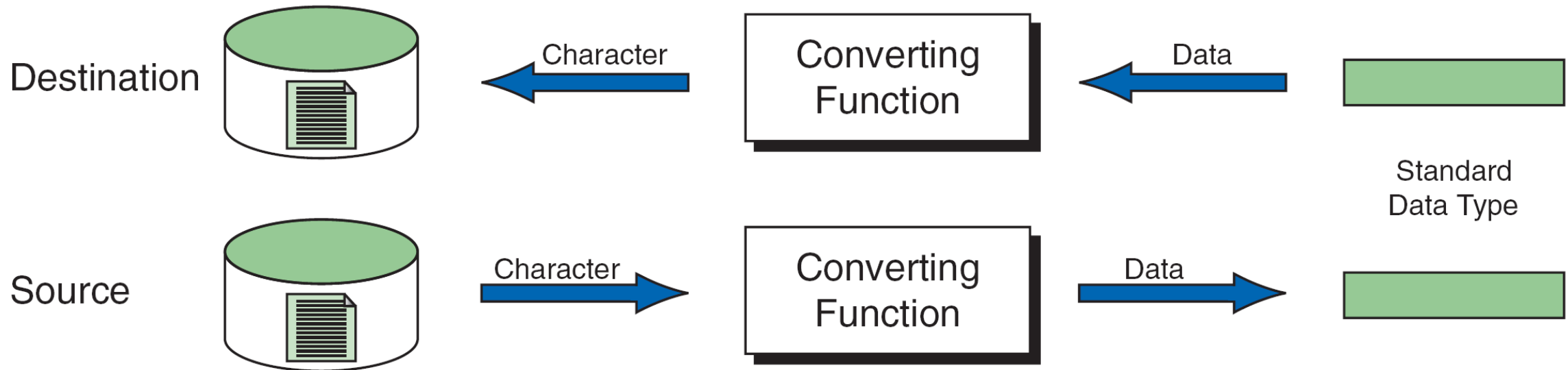
Inserting a valid record

Before inserting: check capacity, require a new ID, check the device and validate the value.

Store the complete record at `rows[count]`, then increase count. A rejected record must not increase count.

If later code relies on sorted order, the insertion must preserve that order or explicitly re-sort.

Text storage converts values



A text format specifies field order, separators and number syntax. Both reader and writer must follow it.

A small text-record format

```
101 7 512
102 9 650
103 7 700
```

Each line: reading ID, device ID, raw value.

Our example permits only three decimal integers.

Names, quoting and arbitrary CSV are outside this format.

Writing fields explicitly

```
int write_one(FILE *fp,  
              const Reading *r)  
{  
    return fprintf(fp, "%d %d %d\n",  
                  r->id,  
                  r->device_id,  
                  r->raw) >= 0;  
}
```

The result reports an output failure.

The caller must also check fclose.

Opening with "w" replaces old content.

Loading a record safely

Read a complete bounded line, parse all three fields and reject extra or malformed data.

Check integer range, raw-value range, known device, duplicate ID and array capacity.

Commit the candidate to the array only after every check succeeds. Distinguish bad input from clean end-of-file.

A load trace

Input line	Result	Staged count
101 7 512	Accept first record	1
102 9 640	Accept second record	2
102 7 700	Reject duplicate ID	Discard staged data
103 7 700	Not read after previous error	—

The companion loader stops on an error and leaves its caller's destination unchanged.

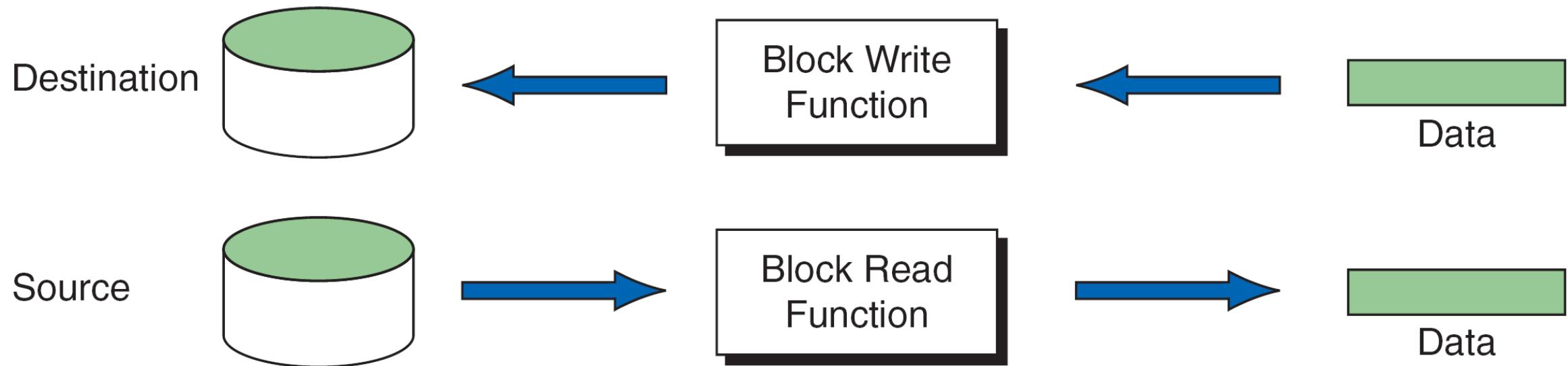
Updating memory does not update the file

Loading creates records in memory. The file still contains the earlier values.

After a valid change, write a new output file and check writing and closing. Then reload it to confirm the result.

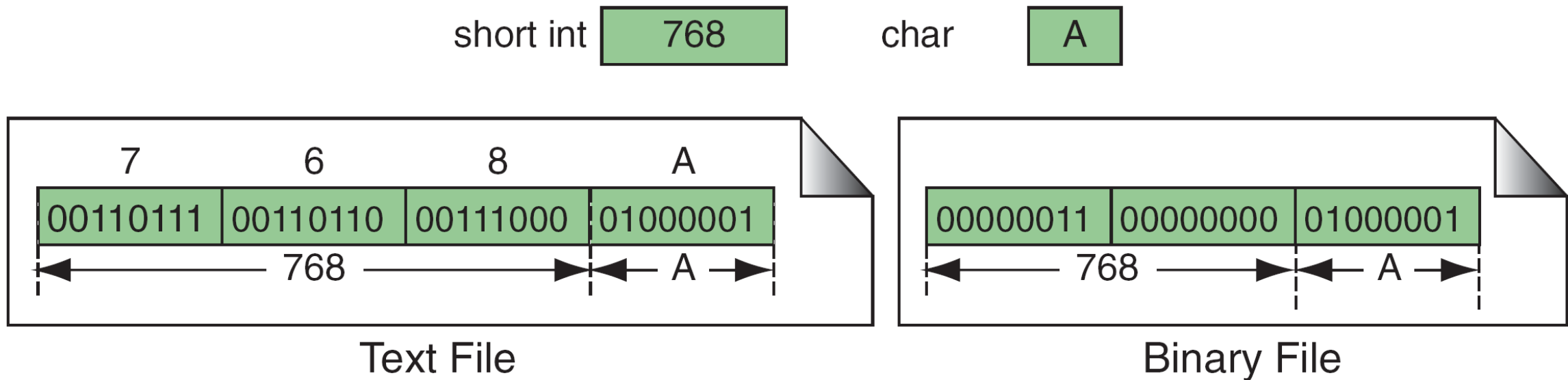
This lesson uses distinct input and output paths. Crash-safe replacement and multi-user updates require additional design.

Block I/O transfers object bytes



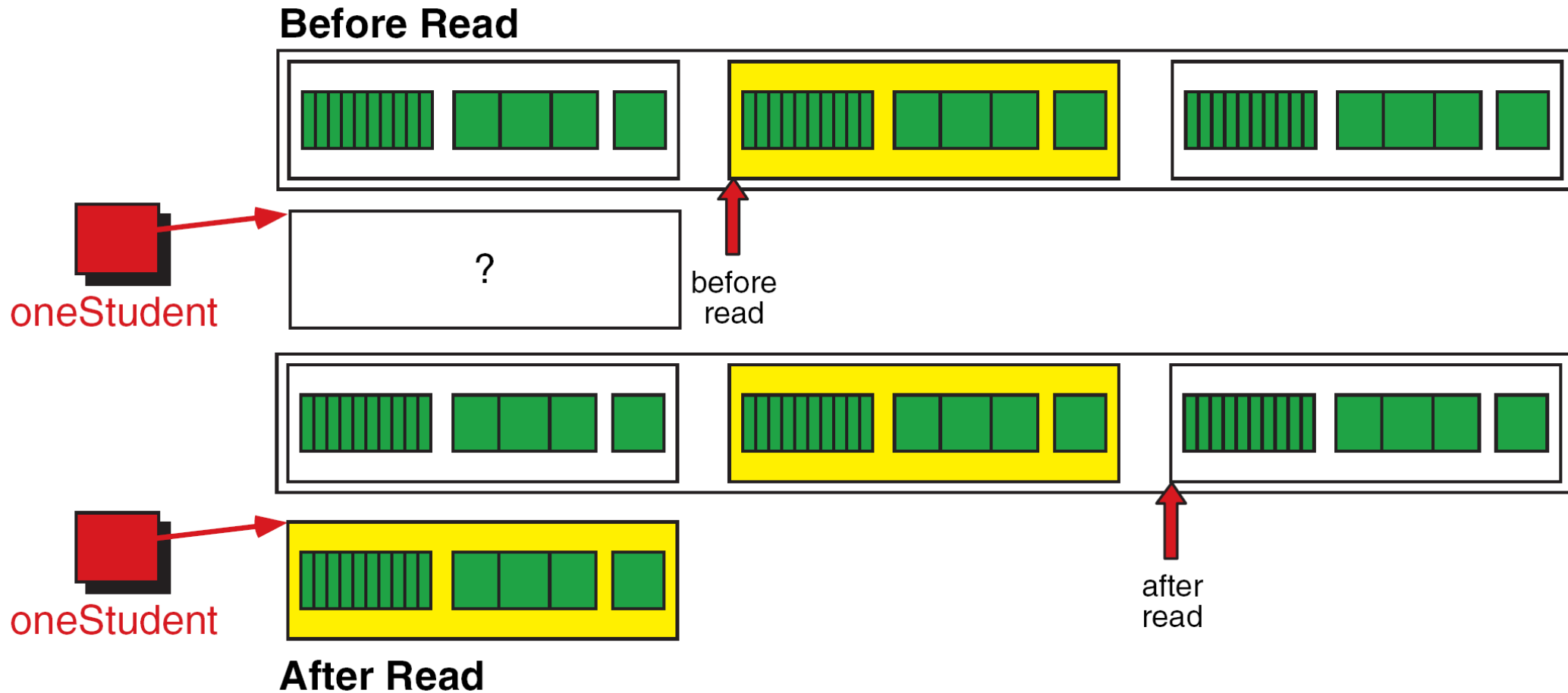
fread and fwrite transfer bytes in blocks. Binary mode does not define an interoperable record format.

Text and binary represent values differently



This figure assumes ASCII, a two-byte short and big-endian byte order. These sizes and representations are not universal.

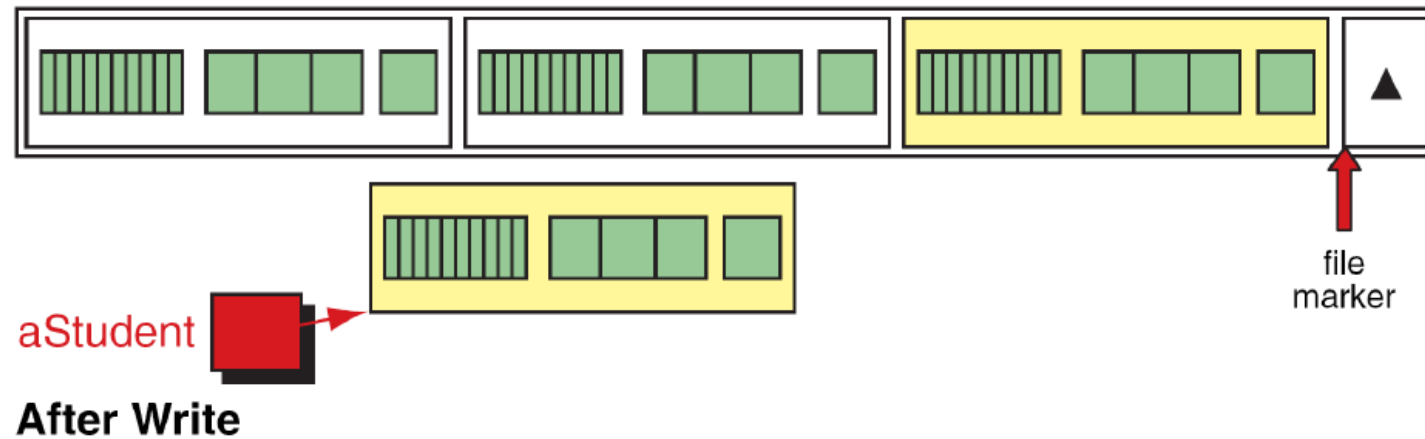
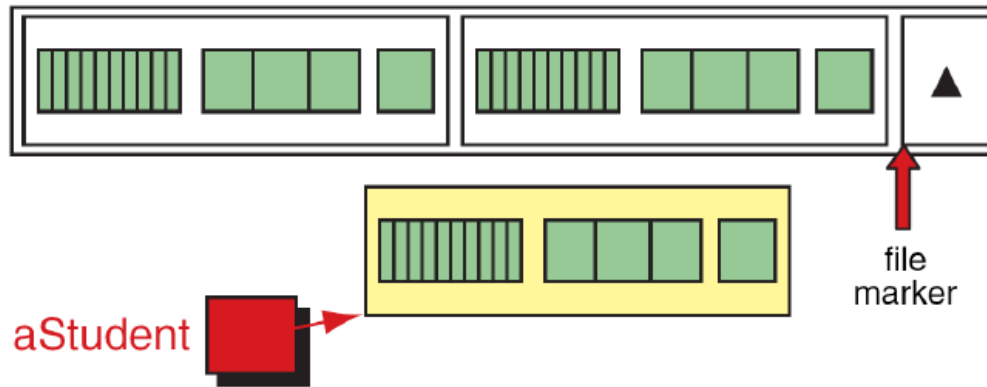
Reading a whole structure



Only use a record after a complete successful read. A raw struct file requires a compatible layout and valid stored values.

Writing a whole structure

Before Write



Padding, integer representation and pointer fields make raw struct dumps unsuitable as a general exchange format.

Struct versus union

Property	struct	union
Storage	Separate members	Overlapping member storage
Values retained	All members together	Use the selected variant
Size	Members plus possible padding	At least the largest member; may include padding
Typical purpose	One record with several fields	Alternative representations

A union does not remember which member your program last selected. Keep a separate tag.

A tagged variant

```
enum Kind { RAW, VOLTS };
```

```
struct Sample {  
    enum Kind kind;  
    union {  
        int raw;  
        double volts;  
    } value;  
};
```

The tag states how to interpret value.

Store and read the member selected by kind.

This is a brief comparison, not a new lab requirement.

Using the selected member

```
struct Sample s = {  
    .kind = RAW,  
    .value.raw = 512  
};  
if (s.kind == RAW)  
    printf("%d\n", s.value.raw);
```

Output: 512.

Writing value.volts would
reuse the storage.

It would not convert raw into
volts.

Trace member access and bounded searches before extending the code.

Load, search/update, save and reload a small record array on the PC. Prepared records or supplied Arduino events can provide input.

Reuse the provided transport and existing hardware. No DBMS installation, SQL application or binary-format project is required.

Check: where is the changed value?

A program loads a file, changes rows[1].raw and exits without saving. What does the next run read?

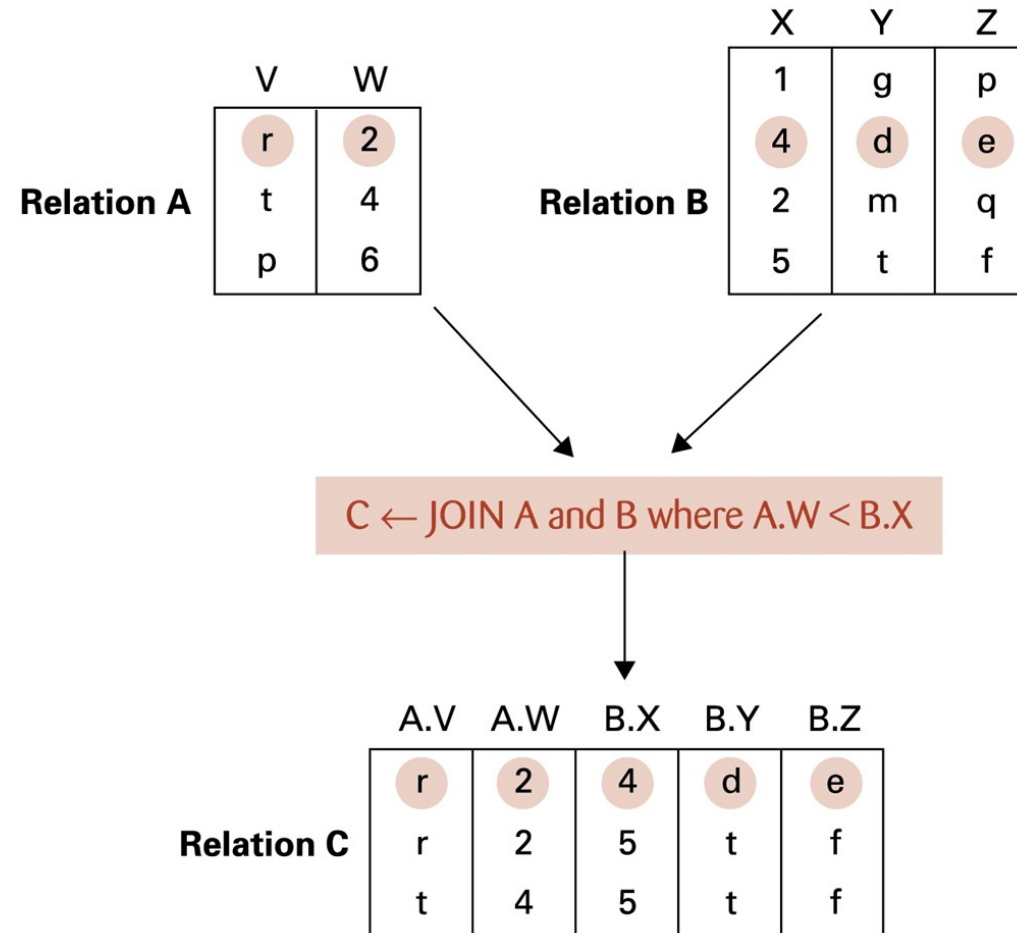
Answer: the previous file contents. Memory changes disappear when the process ends.

Now ask what happens if writing succeeds but closing reports an error. The program must report failure rather than claim the save succeeded.

Original examples for further study

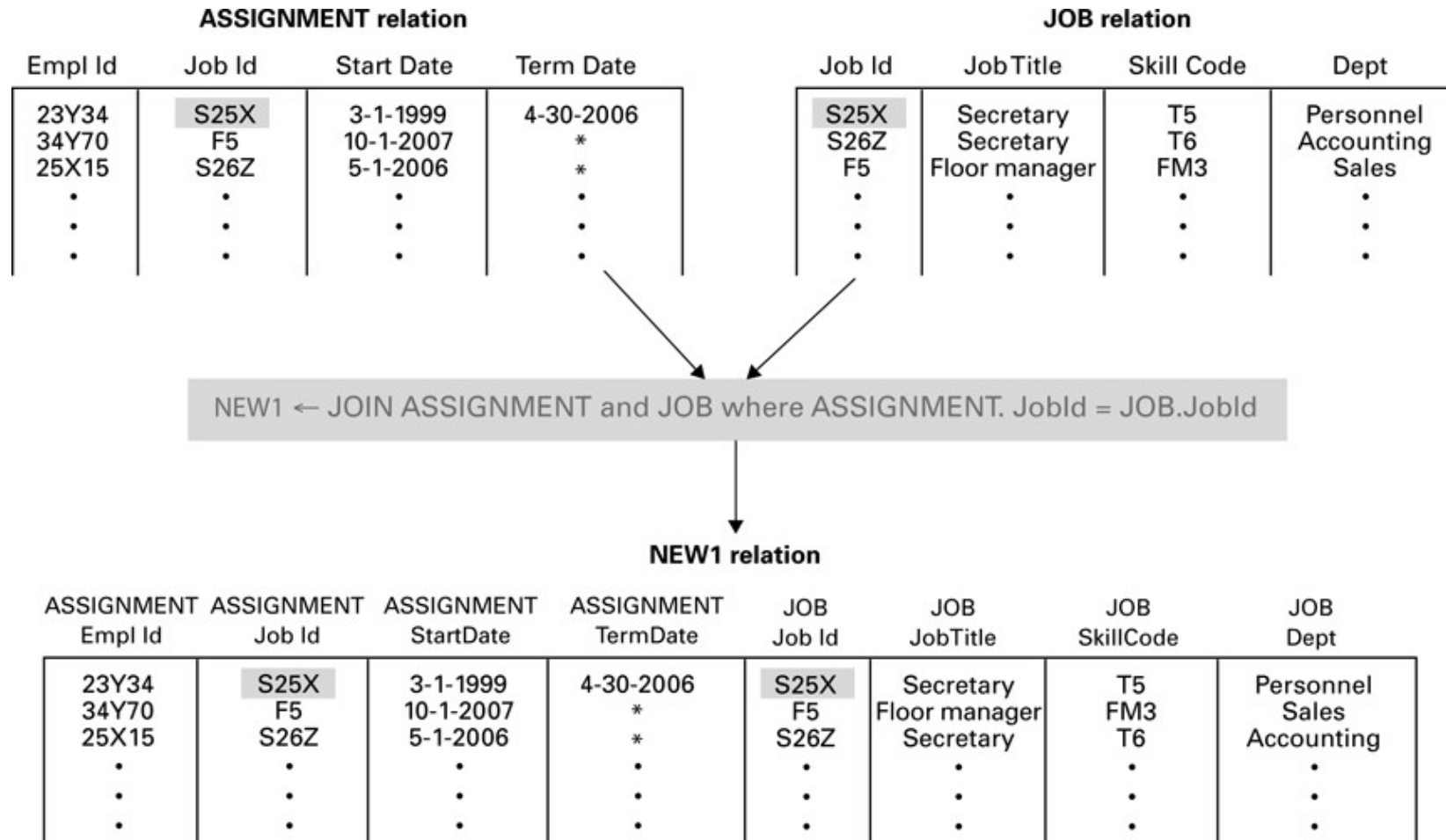
Database mechanisms and longer file-I/O examples
Use selectively after the core concepts

A join can use another condition



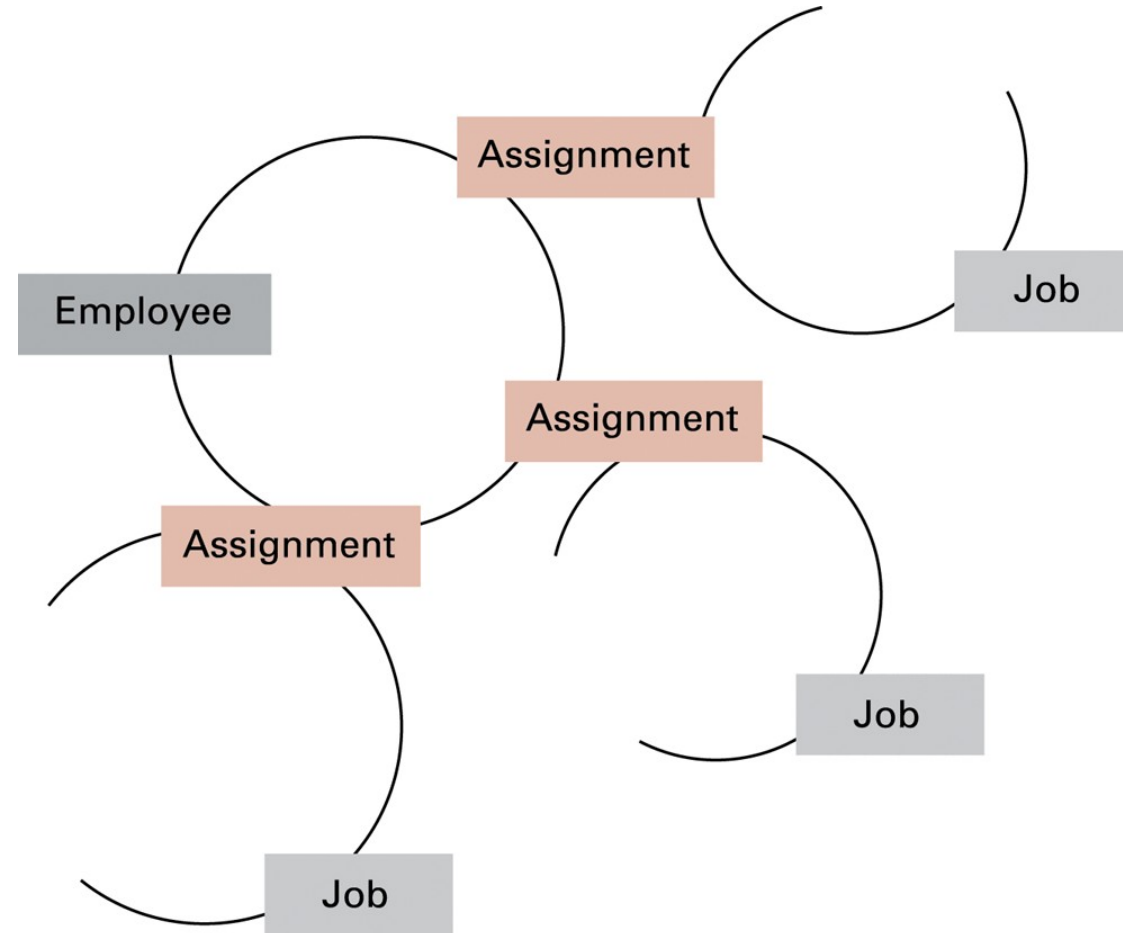
A less-than condition creates all matching pairs, not just equal keys. Predict the three result rows.

Joining assignments with job details



The shared JobId connects each assignment to its job. A join can have more rows than either input subset.

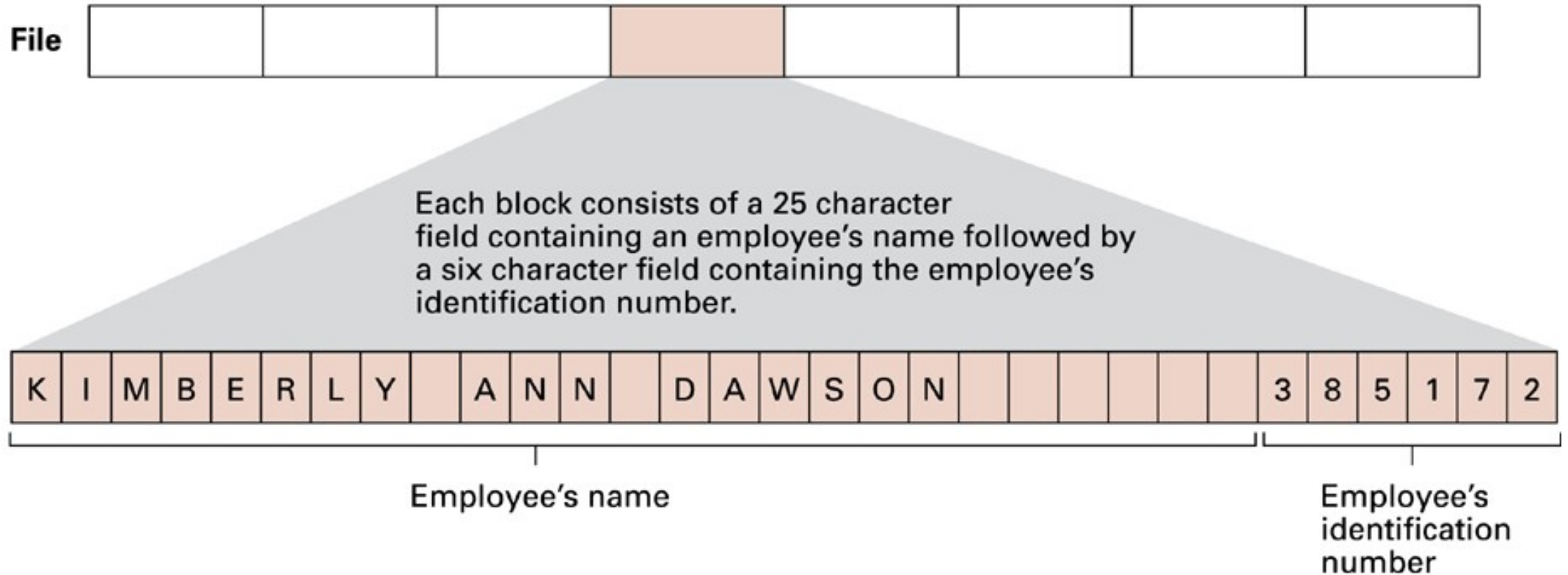
An alternative database model



Object databases preserve objects and associations. This is recognition-level context, not a C++ programming requirement.

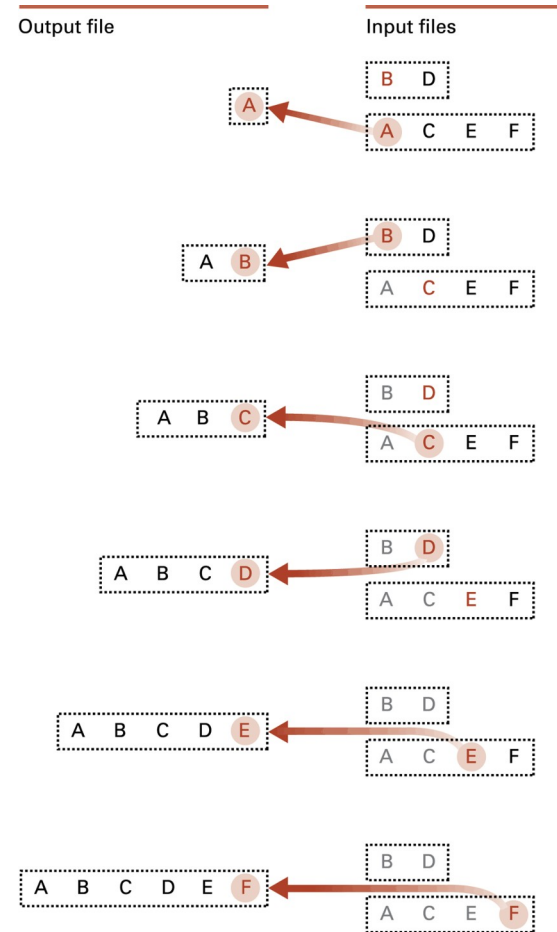
A fixed-width text record

File consists of a sequence of blocks each containing 31 characters.



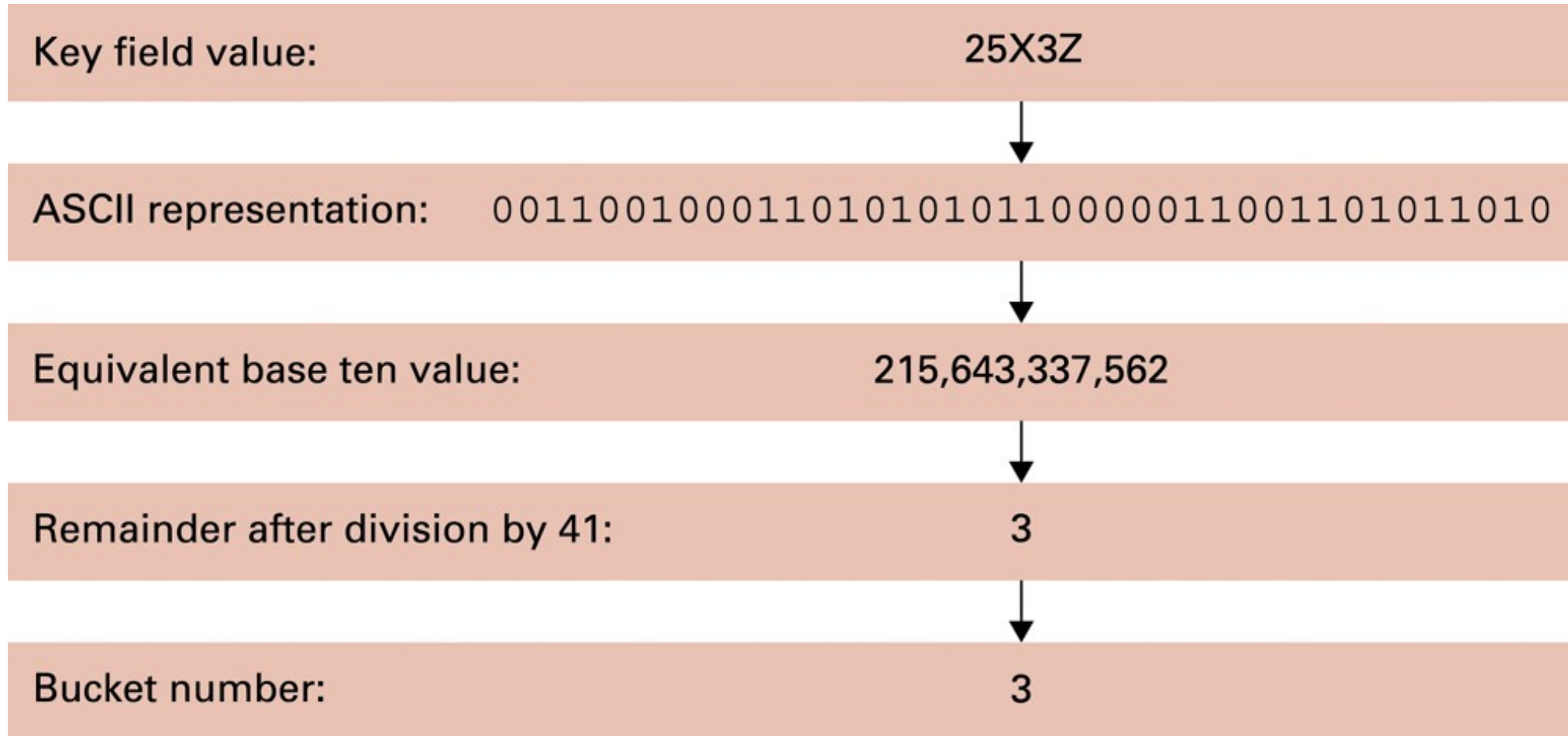
This historical format uses 25 character positions for a name and 6 for an ID. A format must define encoding and field limits.

Merging sorted record files



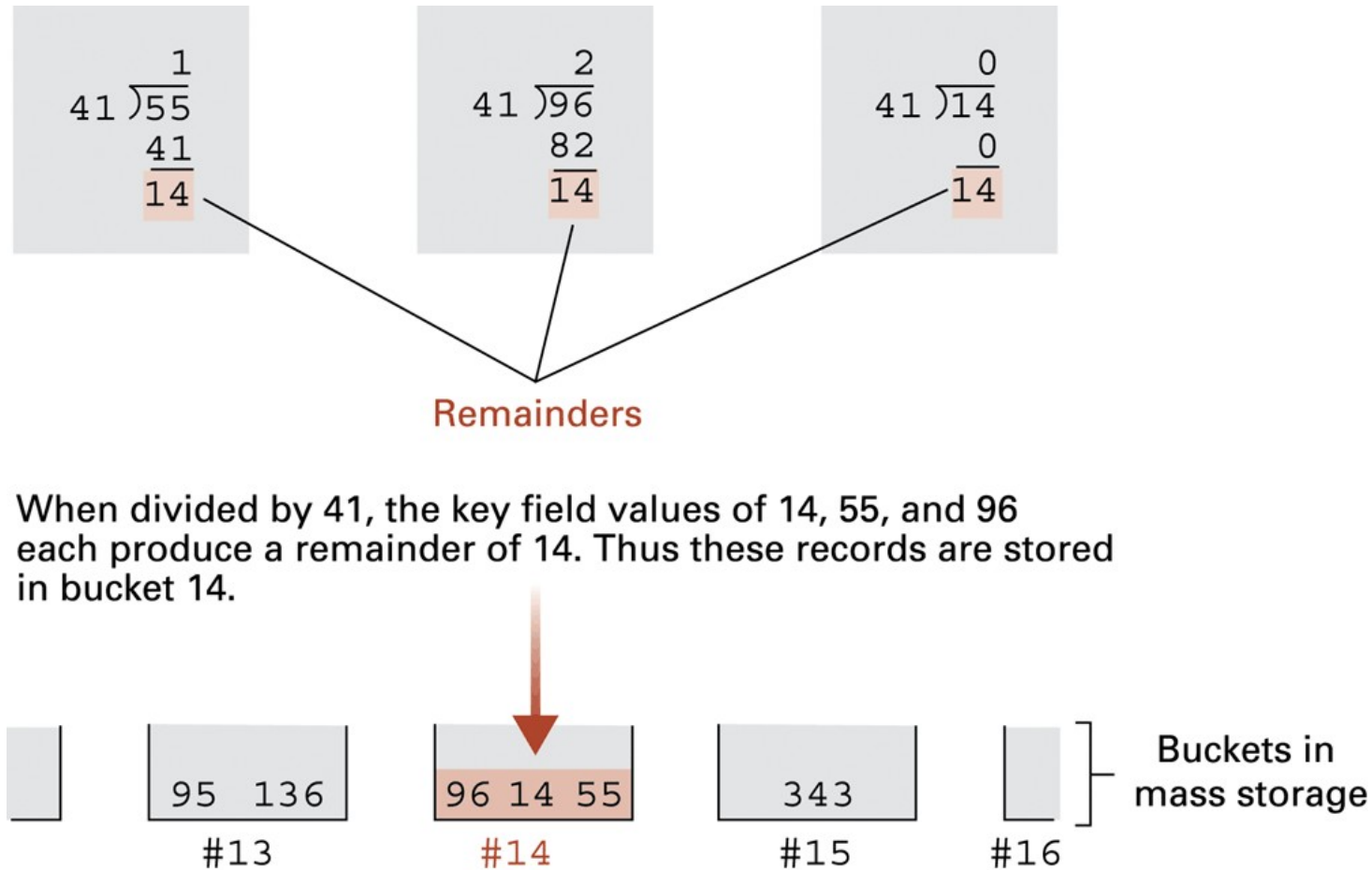
Compare the next available keys. Output the smaller record and advance that input; then copy the remaining input.

A hash chooses a bucket



This example converts the key bytes to an integer and takes remainder 41. It illustrates storage hashing, not encryption.

Different keys can share a bucket



14, 55 and 96 all have remainder 14 modulo 41. Collision handling must still compare the full keys.

Grade-file example: declarations

```
1  /* Create a grades file for transmission to Registrar.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6
7  // Function Declarations
8  int getStu      (FILE* spStu,
9                  int* stuID,  int* exam1,
10                 int* exam2, int* final);
11 int writeStu     (FILE* spGrades,
12                 int  stuID, int  avrg,  char grade);
13 void calcGrade  (int  exam1, int  exam2, int  final,
14                 int* avrg,  char* grade);
15
16 int main (void)
17 {
18     // Local Declarations
19     FILE* spStu;
20     FILE* spGrades;
```

Historical textbook program. Its example grading thresholds are not CSE101 assessment policy.

Grade-file example: opening files

```
21
22     int stuID;
23     int exam1;
24     int exam2;
25     int final;
26     int avrg;
27
28     char grade;
29
30     // Statements
31     printf("Begin student grading\n");
32     if (!(spStu = fopen ("P07-06.DAT", "r")))
33     {
34         printf("\aError opening student file\n");
35         return 100;
36     } // if open input
37
38     if (!(spGrades = fopen ("P07-06Gr.DAT", "w")))
39     {
```

A complete implementation must close an already-opened input if opening the output fails.

Grade-file example: processing records

```
40         printf("\aError opening grades file\n");
41         return 102;
42     } // if open output
43
44     while (getStu
45           (spStu, &stuID, &exam1, &exam2, &final))
46     {
47         calcGrade (exam1, exam2, final, &avrg, &grade);
48         writeStu  (spGrades, stuID, avrg, grade);
49     } // while
50
51     fclose (spStu);
52     fclose (spGrades);
53
54     printf("End student grading\n");
55     return 0;
56 } // main
57
```

The helper reads values, calculates a result and writes it. Output and close failures need explicit checks.

Grade-file example: reading fields

```
58  /*=====getStu=====
59      Reads data from student file.
60      Pre    spStu is an open file.
61             stuID, exam1, exam2, final pass by address
62      Post   reads student ID and exam scores
63             if data read    --returns 1
64             if EOF or error--returns 0
65  */
66  int getStu (FILE* spStu, int* stuID, int* exam1,
67             int* exam2, int* final)
68  {
69      // Local Declarations
70      int ioResult;
71
72      // Statements
73      ioResult = fscanf(spStu, "%d%d%d%d", stuID,
74                      exam1, exam2, final);
75      if (ioResult == EOF)
76          return 0;
77      else if (ioResult != 4)
```

Require all four successful conversions. Distinguish clean EOF, malformed input and a read error.

Grade-file example: validation and calculation

```
78     {
79         printf("\aError reading data\n");
80         return 0;
81     } // if
82 else
83     return 1;
84 } // getStu
85
86 /* ===== calcGrade =====
87 Determine student grade based on absolute scale.
88 Pre   exam1, exam2, and final contain scores
89       avrg and grade are addresses of variables
90 Post  Average and grade copied to addresses
91 */
92 void calcGrade (int exam1, int exam2, int final,
93                int* avrg, char* grade)
94 {
95     // Statements
96     *avrg = (exam1 + exam2 + final) / 3;
```

Check ranges before arithmetic. Integer division truncates the average in this original example.

Grade-file example: selecting a category

```

97     if (*avrg >= 90)
98         *grade = 'A';
99     else if (*avrg >= 80)
100         *grade = 'B';
101     else if (*avrg >= 70)
102         *grade = 'C';
103     else if (*avrg >= 60)
104         *grade = 'D';
105     else
106         *grade = 'F';
107     return;
108 } // calcGrade
109
110 /* ===== writeStu =====
111 Writes student grade data to output file.
112     Pre    spGrades is an open file
113           stuID, avrg, and grade have values to write
114     Post   Data written to file
115 */
```

These thresholds illustrate selection only. They do not define course grades.

Grade-file example: writing results

```
116 int writeStu (FILE* spGrades, int stuID,  
117             int avrg, char grade)  
118 {  
119     // Statements  
120     fprintf(spGrades, "%04d %d %c\n",  
121             stuID, avrg, grade);  
122     return 0;  
123 } // writeStu
```

Results:

Begin student grading
End student grading

Input-----

0090 90 90 90
0089 88 90 89
0081 80 82 81
0079 79 79 79
0070 70 70 70
0069 69 69 69
0060 60 60 60
0059 59 59 59

The original writer always returns zero. A revised writer should report fprintf failure to its caller.

Grade-file example: sample output

```
Output----  
0090 90 A \n  
0089 89 B \n  
0081 81 B \n  
0079 79 C \n  
0070 70 C \n  
0069 69 D \n  
0060 60 D \n  
0059 59 F \n
```

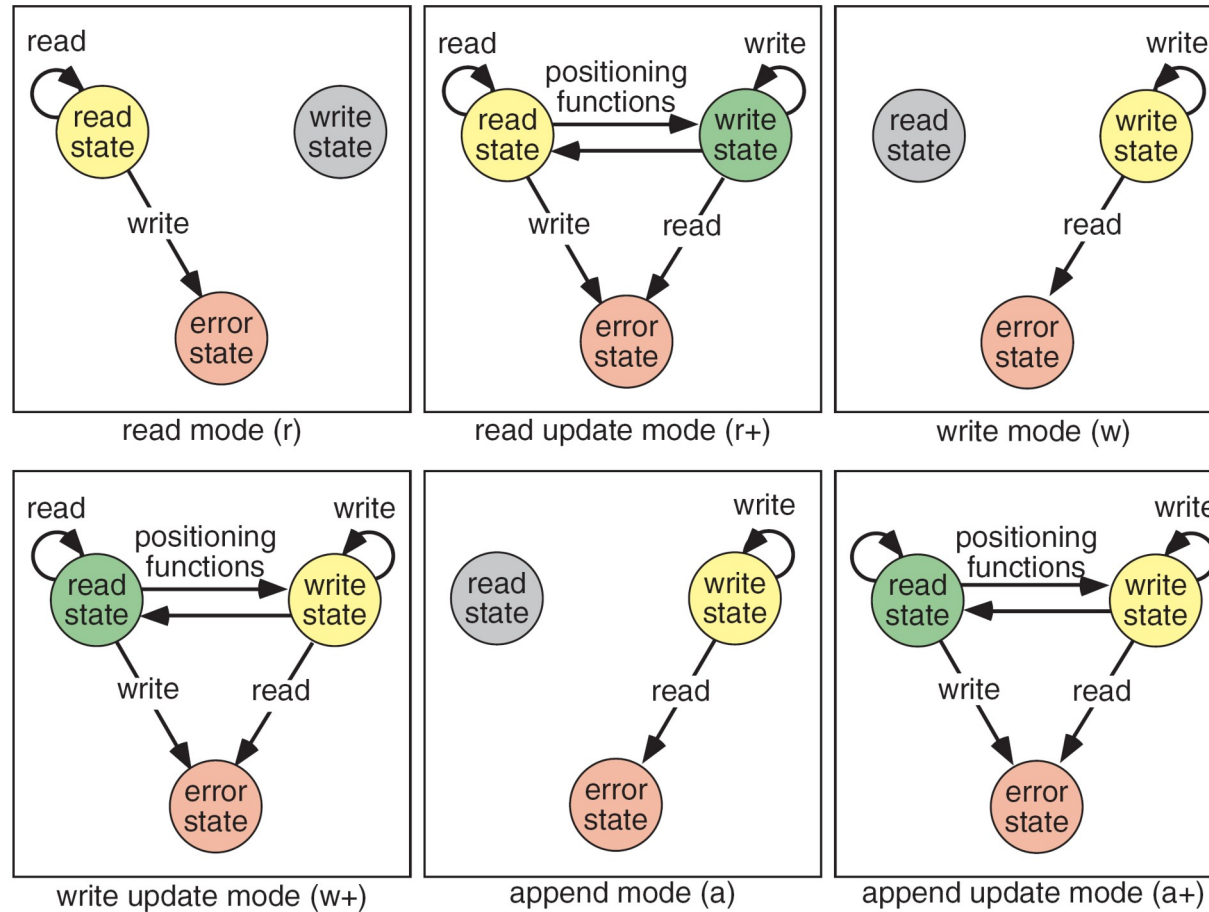
These are synthetic textbook records. The displayed \n represents the newline ending each output record.

File modes: original reference

Mode	r	w	a	r+	w+	a+
Open State	read	write	write	read	write	write
Read Allowed	yes	no	no	yes	yes	yes
Write Allowed	no	yes	yes	yes	yes	yes
Append Allowed	no	no	yes	no	no	yes
File Must Exist	yes	no	no	yes	no	no
Contents of Existing File Lost	no	yes	no	no	yes	no

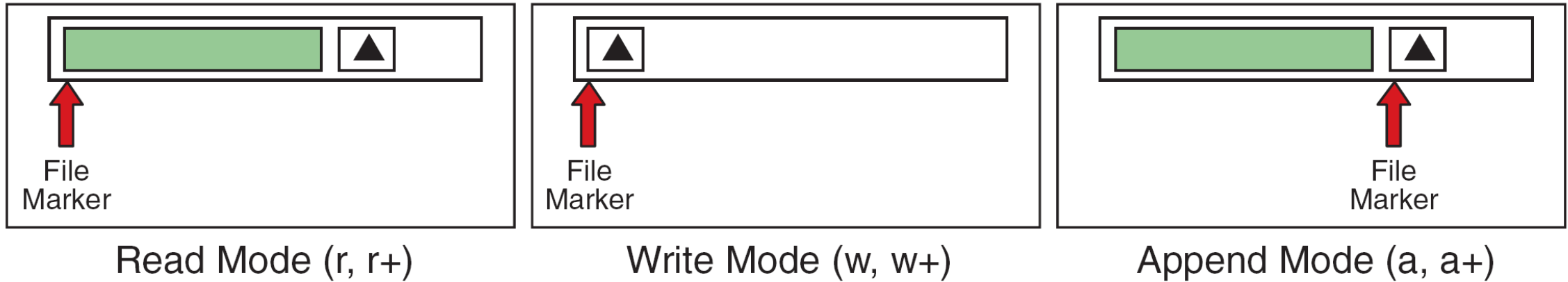
Append writes always go to the end. Add b for binary mode where text translation differs.

Switching between reading and writing



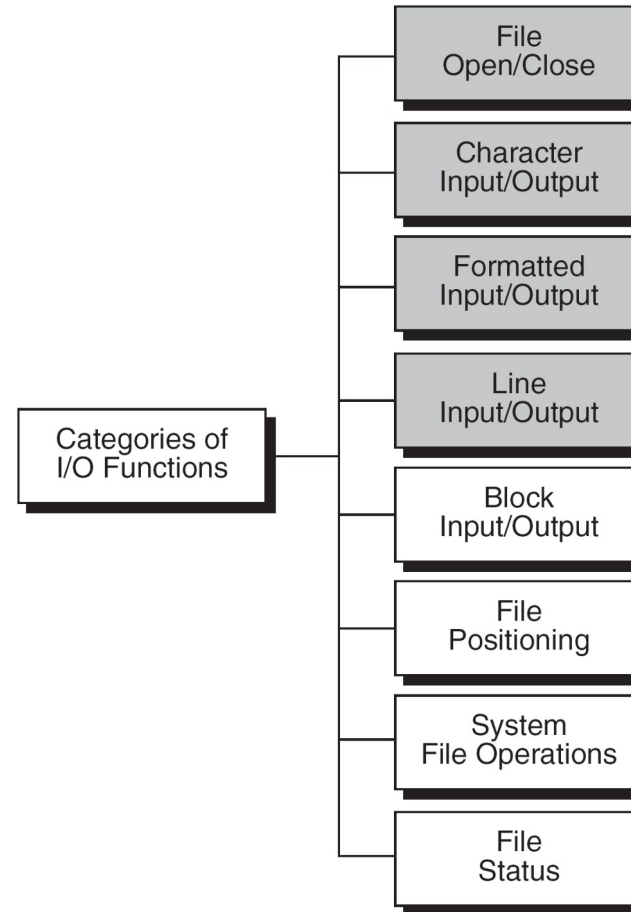
In update mode, output-to-input needs fflush or positioning. Input-to-output needs positioning unless input reached EOF.

Opening positions: original reference



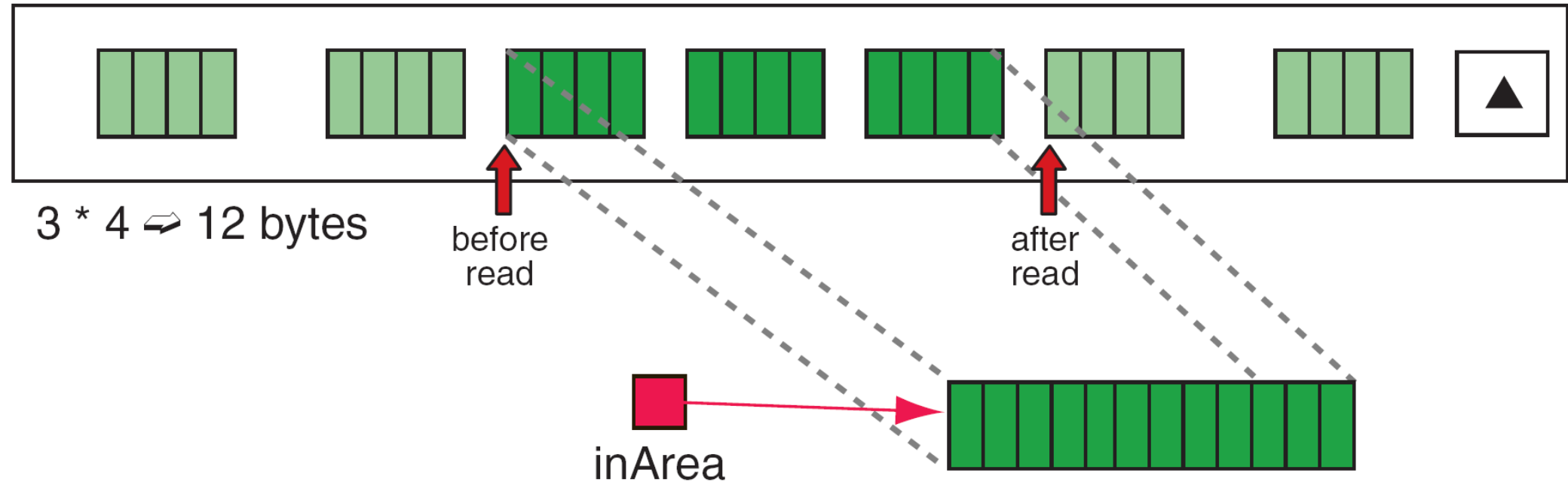
Append guarantees writes at the end. Do not infer a portable initial read position for a+ from this simplified picture.

Families of file operations



Recognize the categories. This does not add all of these APIs to the lab requirements.

Reading a block of integers



```
fread (inArea, sizeof (int), 3, spData);
```

fread returns the number of complete elements read. The 12-byte calculation assumes `sizeof(int)` is 4.

A block-read loop

```
1 // Read a file of integers, three integers at a time.
2 {
3     ...
4 // Local Declarations
5 FILE* spIntFile;
6 int itemsRead;
7 int intAry[3];
8
9 // Statements
10 spIntFile = fopen("int_file.dat", "rb");
11 ...
12 while ((itemsRead = fread(intAry,
13     sizeof(int), 3, spIntFile)) != 0)
14 {
15     // process array
16     ...
17 } // while
18 ...
19 } // block
```

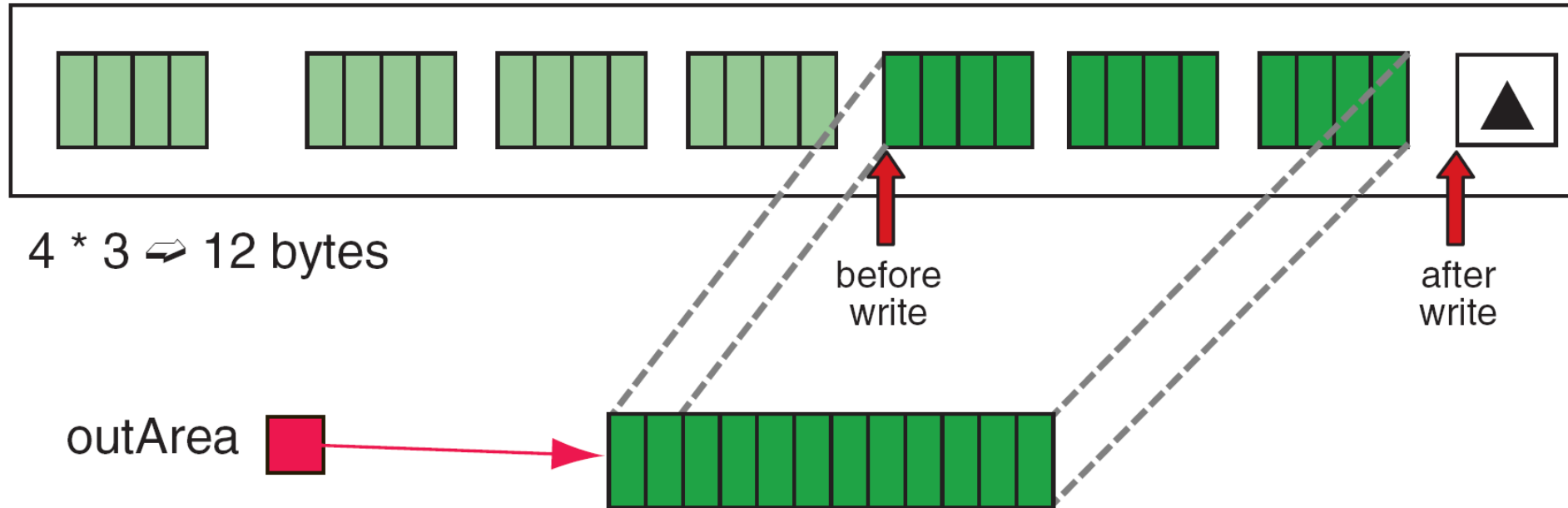
Check fopen before use, process only the returned element count, inspect ferror and close the stream.

A helper reads one structure

```
1  /* Reads one student's data from a file
2      Pre   spStuFile is opened for reading
3      Post  stu data structure filled
4      ioResults returned
5  */
6  int readStudent (STU* oneStudent, FILE* spStuFile)
7  {
8      // Local Declarations
9      int ioResults;
10
11     // Statements
12     ioResults = fread(oneStudent,
13                       sizeof(STU), 1, spStuFile);
14     return ioResults;
15 } // readStudent
```

The return count is size_t. Zero can mean EOF or error; inspect stream status and reject truncated data.

Writing a block of integers



```
fwrite (outArea, sizeof (int), 3, spOut);
```

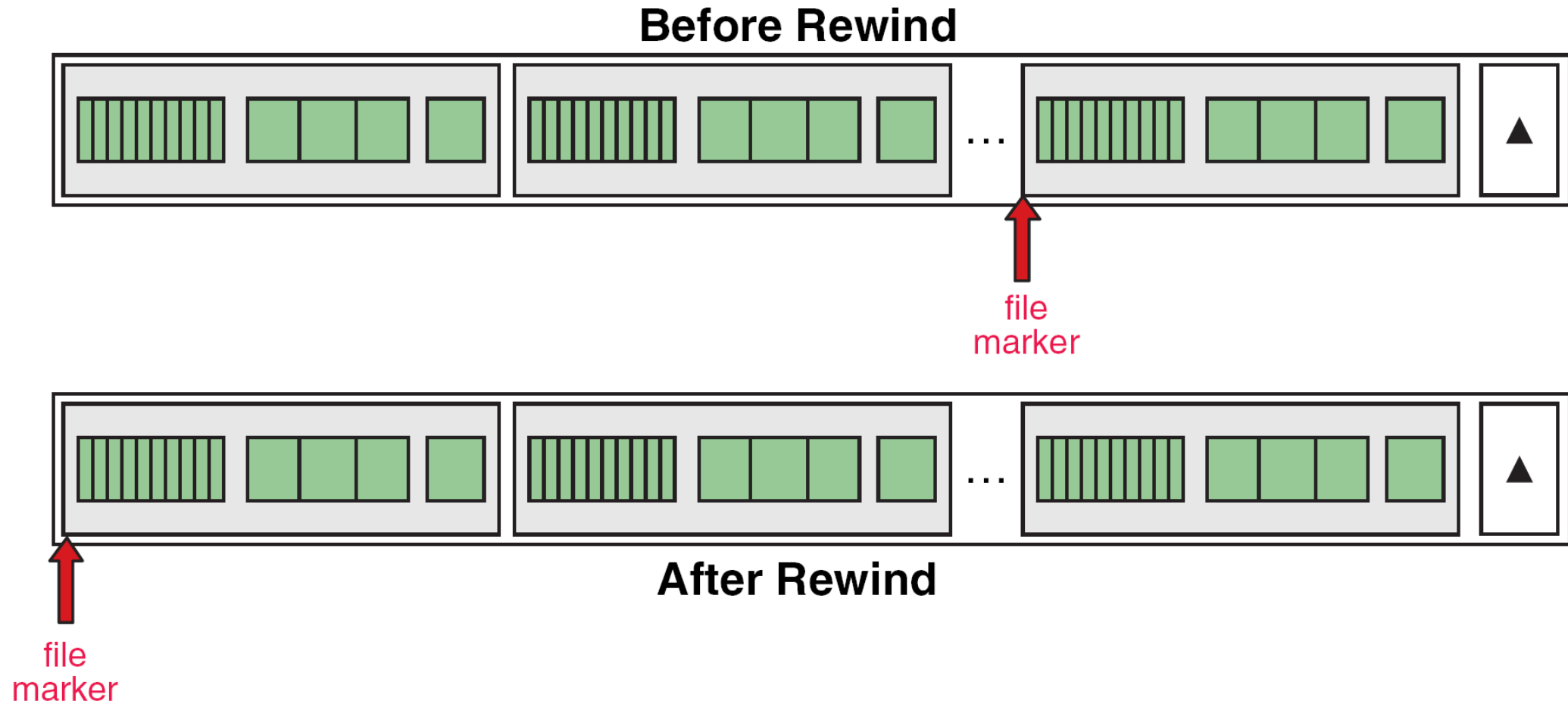
fwrite returns the number of complete elements written. Check the count and the later fclose result.

A helper writes one structure

```
1  /* Writes one student's record to a binary file.
2      Pre  aStudent has been filled
3      Post spOut is open for writing
4      Post aStudent written to spOut
5  */
6  void writeStudent (STU* aStudent, FILE* spOut)
7
8  {
9      // Local Declarations
10     int ioResult;
11
12     // Statements
13     ioResult = fwrite(aStudent,
14                       sizeof(STU), 1, spOut);
15     if (ioResult != 1)
16     {
17         printf("\a Error writing student file \a\n");
18         exit (100);
19     } // if
20     return;
21 }
```

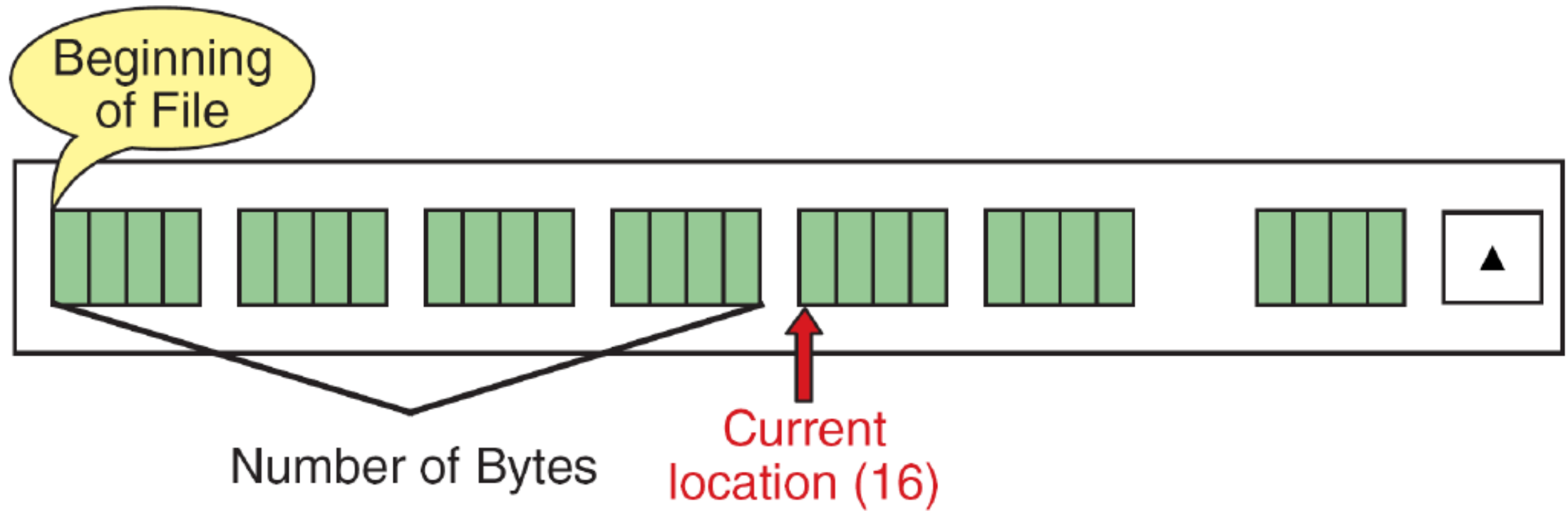
The original exits on a short write. Library-style code can instead report failure so the caller performs cleanup.

Returning to the beginning



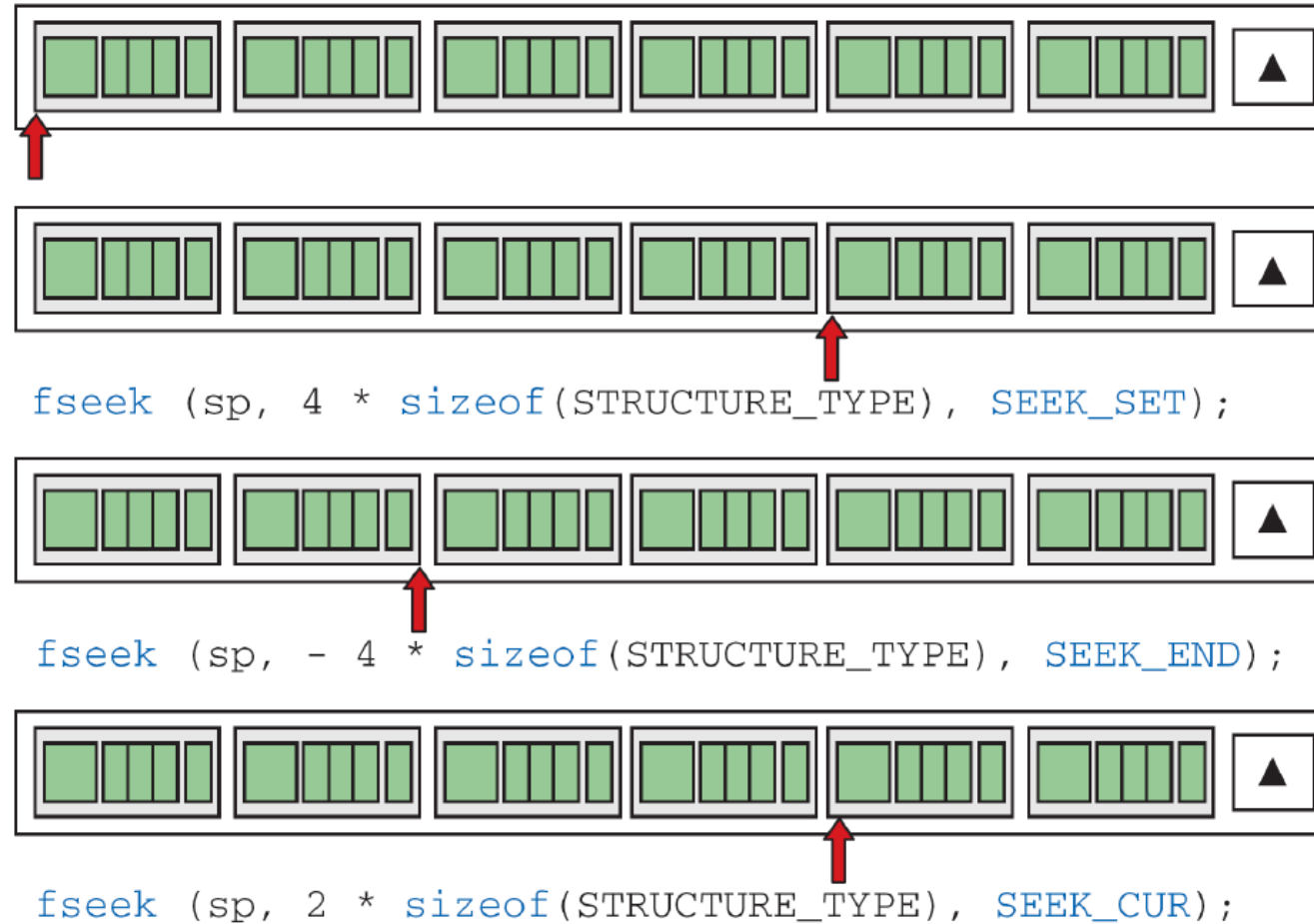
rewind clears EOF and error indicators. If positioning errors must be reported, use and check fseek.

Recording a file position



`ftell` returns long and `-1L` on failure. A text-stream position is not generally a byte count for arithmetic.

Moving within a binary file



Offsets require a defined record size and representable long value. Avoid unsigned negation in the middle example.

Appending binary files: declarations

```
1  /* This program appends two binary files 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* sp1;  
12 FILE* sp2;  
13 int    data;  
14 long   dataCount;  
15 char   fileID[13];  
16
```

This example assumes compatible integer representations. An explicit portable format needs an encoding specification.

Appending binary files: the loop

```
17 // Statements
18 printf("This program appends two files.\n");
19 printf("Please enter file ID of the primary file: ");
20 scanf("%12s", fileID);
21 if (!(sp1 = fopen (fileID, "ab")))
22     printf("\aCan't open %s\n", fileID), exit (100);
23
24 if (!(dataCount = (ftell (sp1))))
25     printf("\a%s has no data\n", fileID), exit (101);
26 dataCount /= sizeof(int);
27
28 printf("Please enter file ID of the second file: ");
29 scanf("%12s", fileID);
30 if (!(sp2 = fopen (fileID, "rb")))
31     printf("\aCan't open %s\n", fileID), exit (110);
32
33 while (fread (&data, sizeof(int), 1, sp2) == 1)
34 {
35     fwrite (&data, sizeof(int), 1, sp1);
36     dataCount++;
37 } // while
```

Check scanf, ftell, every write and read status. Opening in append mode alone does not give a portable file-size query.

Appending binary files: completion

```
38
39     if (! feof(sp2))
40         printf("\aRead Error. No output.\n"), exit (120);
41
42     fclose (sp1);
43     fclose (sp2);
44
45     printf("Append complete: %ld records in file\n",
46           dataCount);
47     return 0;
48 } // main
```

Earlier writes may already have occurred when reading fails. “No output” is misleading; this is not an atomic transaction.

Brookshear & Brylow, Computer Science: An Overview, 12th Global Edition, Chapter 9: database models, integrity and file organization.

PostgreSQL tutorial: relational concepts, queries, joins and transactions. Read for ideas; installation is optional.

C reference: structures/unions, member access and standard I/O. The companion examples use PC C17.