

Pointers and Memory

David Marchant
Based on slides by Troels Henriksen

2025-09-15

Data representations

Examining data in C

A machine view of text

Binary IO

Locality of reference

Examining data representations

- **Code to print byte representation of data**

- ▶ Casting pointer to unsigned char* allows treatment as byte array.

```
void show_bytes(unsigned char* start, size_t len) {  
    size_t i;  
    for (i = 0; i < len; i++) {  
        printf("%p\t0x%.2x\n", start+i, start[i]);  
    }  
    printf("\n");  
}
```

printf directives:

- %p: Print pointer.
- %x: Print hexadecimal.

show_bytes execution example

```
int a = 15213;  
printf("int a = 15213;\n");  
show_bytes((unsigned char*) &a, sizeof(int));
```

Result (Linux x86-64):

```
0x7fffb7f71dbc 6d  
0x7fffb7f71dbd 3b  
0x7fffb7f71dbe 00  
0x7fffb7f71dbf 00
```

Data representations

Examining data in C

A machine view of text

Binary IO

Locality of reference

Text IO

```
printf("Hello, world!\n");
```

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;
```

```
printf("an integer: %d\n", x);
```


Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;  
printf("an integer: %d\n", x);
```

an integer: 123

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;
```

```
printf("an integer: %d\n", x);
```

an integer: 123

```
printf("an integer: %5d\n", x);
```

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;
```

```
printf("an integer: %d\n", x);
```

an integer: 123

```
printf("an integer: %5d\n", x);
```

an integer: 123

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;
```

```
printf("an integer: %d\n", x);
```

an integer: 123

```
printf("an integer: %5d\n", x);
```

an integer: 123

```
double y = 1.23;
```

```
printf("a float: %f\n", y);
```

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;  
printf("an integer: %d\n", x);
```

an integer: 123

```
printf("an integer: %5d\n", x);
```

an integer: 123

```
double y = 1.23;  
printf("a float: %f\n", y);
```

a float: 1.230000

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;
```

```
printf("an integer: %d\n", x);
```

an integer: 123

```
printf("an integer: %5d\n", x);
```

an integer: 123

```
double y = 1.23;
```

```
printf("a float: %f\n", y);
```

a float: 1.230000

```
printf("a mess: %d\n", y);
```

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;  
printf("an integer: %d\n", x);
```

an integer: 123

```
printf("an integer: %5d\n", x);
```

an integer: 123

```
double y = 1.23;  
printf("a float: %f\n", y);
```

a float: 1.230000

```
printf("a mess: %d\n", y);
```

a mess: -171948608

Text IO

```
printf("Hello, world!\n");
```

Hello, world!

```
int x = 123;
```

```
printf("an integer: %d\n", x);
```

an integer: 123

```
printf("an integer: %5d\n", x);
```

an integer: 123

```
double y = 1.23;
```

```
printf("a float: %f\n", y);
```

a float: 1.230000

```
printf("a mess: %d\n", y);
```

a mess: -171948608

Make sure format specifiers and argument types match!

Text representation

- Machines only understand numbers, and text is an abstraction!
- E.g. when the terminal receives a byte with the value 65, it draws an A.
- `printf()` determines which *bytes* must be written to the terminal to produce the text corresponding to e.g. the number 123: [49, 50, 51].

Character sets

A character set maps a *number* to a *character*.

- ASCII defines characters in the range 0—127 (asciitable.com).
- Some are invisible/unprintable *control characters*
- *Unicode* is a superset of ASCII that defines tens of thousands of characters for all the world's scripts.

We'll assume **ASCII**, which has the simple property that 1 byte = 1 character.

The ASCII table

Control characters				Normal characters											
000	nul	016	dle	032	␣	048	0	064	@	080	P	096	'	112	p
001	soh	017	dc1	033	!	049	1	065	A	081	Q	097	a	113	q
002	stx	018	dc2	034	“	050	2	066	B	082	R	098	b	114	r
003	etx	019	dc3	035	#	051	3	067	C	083	S	099	c	115	s
004	eot	020	dc4	036	\$	052	4	068	D	084	T	100	d	116	t
005	enq	021	nak	037	%	053	5	069	E	085	U	101	e	117	u
006	ack	022	syn	038	&	054	6	070	F	086	V	102	f	118	v
007	bel	023	etb	039	'	055	7	071	G	087	W	103	g	119	w
008	bs	024	can	040	(056	8	072	H	088	X	104	h	120	x
009	tab	025	em	041)	057	9	073	I	089	Y	105	i	121	y
010	lf	026	eof	042	*	058	:	074	J	090	Z	106	j	122	z
011	vt	027	esc	043	+	059	;	075	K	091	[107	k	123	{
012	np	028	fs	044	,	060	<	076	L	092	␣	108	l	124	
013	cr	029	gs	045	-	061	=	077	M	093]	109	m	125	}
014	so	030	rs	046	.	062	>	078	N	094	^	110	n	126	~
015	si	031	us	047	/	063	?	079	O	095	_	111	o	127	del

Turning numbers into text

```
int x = 1234;  
printf("x: %d\n", x);
```

Turning numbers into text

```
int x = 1234;  
printf("x: %d\n", x);
```

The text *string* that is passed to `printf()` looks like this in memory:

Characters	x	:		%	d	\n	\0
Bytes	120	58	32	37	100	10	0

Turning numbers into text

```
int x = 1234;  
printf("x: %d\n", x);
```

The text *string* that is passed to `printf()` looks like this in memory:

Characters	x	:		%	d	\n	\0
Bytes	120	58	32	37	100	10	0

`printf()` rewrites format specifiers (`%d`) to the textual representation of their corresponding value argument:

Characters	x	:		1	2	3	4	\n	\0
Bytes	120	58	32	49	50	51	52	10	0

These bytes (except the 0) are then written to *standard output* (typically the terminal) which interprets them as characters and eventually draws pixels on the screen.

Machine representation versus text representation

```
int x = 305419896;
```

- Written as hexadecimal (base-16), this number is 0x12345678.
- One hexadecimal digit is 4 bit, so each group of two digits is one byte, and the number takes four bytes (32 bits).
- The *machine representation* in memory on an x86 CPU is
0x78 0x56 0x34 0x12
- A *decimal text representation* in memory on *any* CPU is
0x33 0x30 0x35 0x34 0x35 0x36 0x37 0x38
- Endianness has *no effect on text* (at least not with single-byte characters).
- In C, we have the additional convention that any string must be NUL-terminated.
- We identify a string with the address of its first character.

Data representations

Examining data in C

A machine view of text

Binary IO

Locality of reference

Writing bytes

The `fwrite` procedure writes raw data to an open file:

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

`ptr`: the address in memory of the data.

`size`: the size of each data element in bytes.

`nmemb`: the number of data elements.

`stream`: the target file (opened with `fopen()`).

- Returns the number of data elements written (equal to `nmemb` unless an error occurs).
- Usually no difference between writing one `size x*y` element or `x size-y` elements—do whatever is convenient.

Example of `fwrite()`

```
#include <stdio.h>

int main(void) {
    // Open for writing ("w")
    FILE *f = fopen("output", "w");

    char c = 42;

    fwrite(&c, sizeof(char), 1, f);
    c = 43;
    fwrite(&c, sizeof(char), 1, f);

    fclose(f);
}
```

- Produces a file output.
- File contains the byte 42, corresponding to the ASCII character `*`.
- **char** is just an 8-bit integer type!
 - ▶ No special “character” meaning.
 - ▶ Most Unicode characters will not fit in a single `char` (e.g. ‘æ’ needs 16 bits in UTF-8).
 - ▶ Name is unfortunate/historical.
 - ▶ Signedness is *implementation-defined* for historical reasons.

Another example

```
#include <stdio.h>

int main(void) {
    FILE *f = fopen("output", "w");

    int x = 0x53505048;
    // Stored as 0x48 0x50 0x50 0x53

    fwrite(&x, sizeof(int), 1, f);

    fclose(f);
}
```

- Writes bytes 0x48 0x50 0x50 0x53.
- Corresponds to ASCII characters HPPS.
- A big-endian machine would produce SPPH.
- **Don't write code that depends on this!**

Converting a non-negative integer to its ASCII representation

```
FILE *f = fopen("output", "w");
int x = 1337;           // Number to write;
char s[10];            // Output buffer.
int i = 10;             // Index of last character written.
while (1) {
    int d = x % 10;      // Pick out last decimal digit.
    x = x / 10;          // Remove last digit.
    i = i - 1;           // Index of next character.
    s[i] = '0' + d;      // Save ASCII character for digit.
    if (x == 0) { break; } // Stop if all digits written.
}
fwrite(&s[i], sizeof(char), 10-i, f); // Write ASCII bytes.
fclose(f);                        // Close output file.
```

Reading bytes

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

`ptr`: where to put the data we read.

`size`: the size of each data element in bytes.

`nmemb`: the number of data elements.

`stream`: the target file (opened with `fopen()`).

Very similar to `fwrite()`!

Reading all the bytes in a file

```
#include <stdio.h>
#include <assert.h>

int main(int argc, char* argv[]) {
    FILE *f = fopen(argv[1], "r");
    unsigned char c;
    while (fread(&c, sizeof(char), 1, f) == 1) {
        printf("%3d_", (int)c);
        if (c > 31 && c < 127) {
            fwrite(&c, sizeof(char), 1, stdout);
        }
        printf("\n");
    }
}
```

Running fread-bytes

```
$ gcc -o fread-bytes -Wall -Wextra -pedantic fread-bytes.c
```

Running fread-bytes

```
$ gcc -o fread-bytes -Wall -Wextra -pedantic fread-bytes.c
```

```
$ ./fread-bytes fread-bytes.c
```

```
35 #
```

```
105 i
```

```
110 n
```

```
99 c
```

```
108 l
```

```
117 u
```

```
100 d
```

```
101 e
```

```
32
```

```
60 <
```

```
...
```

Running fread-bytes

```
$ gcc -o fread-bytes -Wall -Wextra -pedantic fread-bytes.c
```

```
$ ./fread-bytes fread-bytes.c $ ./fread-bytes fread-bytes
```

35	#	127
105	i	69 E
110	n	76 L
99	c	70 F
108	l	2
117	u	1
100	d	1
101	e	0
32		0
60	<	0
...		...

Text files versus binary files

- To the system there is no difference between “text files” and “binary files”!
- All files are just byte sequences.
- *Colloquially*: a text file is a file that is understandable when the bytes are interpreted as characters (in ASCII or some other character set).

Text files versus binary files

- **To the system there is no difference between “text files” and “binary files”!**
- All files are just byte sequences.
- *Colloquially*: a text file is a file that is understandable when the bytes are interpreted as characters (in ASCII or some other character set).

Compactness of storage

- A 32-bit integer takes up to 12 bytes to store as base-10 ASCII digits
- 4 bytes as raw data
- **Raw data takes up less space and is much faster to read.**
- But we need special programs to decode the data to human-readable form.

IO takeaways

- Use `printf()` for text output.
- (And `scanf()` for text *input*.)
- Use `fwrite()` to write raw data.
- Use `fread()` to read raw data.
- Raw data files are more compact and faster to read/write.

Data representations

- Examining data in C

- A machine view of text

- Binary IO

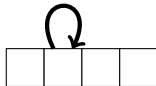
Locality of reference

Locality

Principle of locality

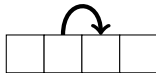
Programs tend to access data located near that which was accessed recently.

Temporal locality



Accessing data that was accessed recently.

Spatial locality



Accessing data that is close to data that was accessed recently.

- **General principles** — definition of “close” depends on the exact form of storage.
 - ▶ E.g. addresses for memory.

Locality example

```
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

Data references

- References array elements in succession (*stride* of 1).

Locality example

```
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

Data references

- References array elements in succession (*stride* of 1). **Spatial locality.**
- References variable `sum` each iteration.

Locality example

```
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

Data references

- References array elements in succession (*stride* of 1). **Spatial locality.**
- References variable `sum` each iteration. **Temporal locality.**

Instruction references

- Executes instructions in sequence.

Locality example

```
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

Data references

- References array elements in succession (*stride* of 1). **Spatial locality.**
- References variable `sum` each iteration. **Temporal locality.**

Instruction references

- Executes instructions in sequence. **Spatial locality.**
- Cycles through loop repeatedly.

Locality example

```
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

Data references

- References array elements in succession (*stride* of 1). **Spatial locality.**
- References variable `sum` each iteration. **Temporal locality.**

Instruction references

- Executes instructions in sequence. **Spatial locality.**
- Cycles through loop repeatedly. **Temporal locality.**

Code as it's normally written has good locality by default, so we tend to focus only on *data locality*.

C array layout

```
int A[M][N];
```

To represent multi-dimensional arrays, C uses *row major order*.

Main consequence

- Rows are contiguous in memory.

Example

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$

is represented as

11	12	13	14	21	22	23	24	31	32	33	34
----	----	----	----	----	----	----	----	----	----	----	----

Implications

- $A[i][j]$ and $A[i][j+1]$ are adjacent.
- $A[i][j]$ and $A[i+1][j]$ are distant.

Eyeballing locality

Being able to glance at code and get a qualitative sense of its locality properties is a key skill for a programmer.

Does this function have good locality with respect to array A?

```
int sumrows(int A[M][N]) {  
    int sum = 0;  
  
    for (int i = 0; i < M; i++)  
        for (int j = 0; j < N; j++)  
            sum += A[i][j];  
    return sum;  
}
```

Does this function have good locality with respect to array A?

```
int sumcols(int A[M][N]) {  
    int sum = 0;  
  
    for (int j = 0; j < N; j++)  
        for (int i = 0; i < M; i++)  
            sum += A[i][j];  
    return sum;  
}
```

Transforming code for better locality

Can we permute the loops of this function such that we are accessing the memory of array A with a stride of 1?

```
int sum3d(int A[L][M][N]) {  
    int sum = 0;  
  
    for (int i = 0; i < M; i++)  
        for (int j = 0; j < N; j++)  
            for (int k = 0; k < L; k++)  
                sum += A[k][i][j];  
    return sum;  
}
```

Transforming code for better locality

Can we permute the loops of this function such that we are accessing the memory of array A with a stride of 1?

```
int sum3d(int A[L][M][N]) {  
    int sum = 0;  
  
    for (int i = 0; i < M; i++)  
        for (int j = 0; j < N; j++)  
            for (int k = 0; k < L; k++)  
                sum += A[k][i][j];  
    return sum;  
}
```

Yes: place them in order k, i, j.