Data Representation in C

Troels Henriksen

Data representations

- Examining data in C
- A machine view of text
- Binary IO

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

Text IO

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

Text IO

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

```
Hello, world!
```

Text IO

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

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

```
Hello, world!
```

Text IO

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

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

```
Hello, world!

an integer: 123
```

Text IO

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

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

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

```
Hello, world!

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
```

Text IO

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

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

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

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

```
Hello, world!

an integer: 123

an integer:   123
```

Text IO

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

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

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

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

```
Hello, world!

an integer: 123

an integer:   123

a float: 1.230000
```

Text IO

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

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

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

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

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

```
Hello, world!

an integer: 123

an integer:   123

a float: 1.230000
```

Text IO

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

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

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

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

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

```
Hello, world!

an integer: 123

an integer:   123

a float: 1.230000

a mess: 4202562
```

Text IO

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

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

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

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

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

```
Hello, world!

an integer: 123

an integer:   123

a float: 1.230000

a mess: 4202562
```

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

<table>
<tr><th colspan="4">Control characters</th><th colspan="12">Normal characters</th></tr>
<tr><td>000</td><td>nul</td><td>016</td><td>dle</td><td>032</td><td>␣</td><td>048</td><td>0</td><td>064</td><td>@</td><td>080</td><td>P</td><td>096</td><td>‘</td><td>112</td><td>p</td></tr>
<tr><td>001</td><td>soh</td><td>017</td><td>dc1</td><td>033</td><td>!</td><td>049</td><td>1</td><td>065</td><td>A</td><td>081</td><td>Q</td><td>097</td><td>a</td><td>113</td><td>q</td></tr>
<tr><td>002</td><td>stx</td><td>018</td><td>dc2</td><td>034</td><td>“</td><td>050</td><td>2</td><td>066</td><td>B</td><td>082</td><td>R</td><td>098</td><td>b</td><td>114</td><td>r</td></tr>
<tr><td>003</td><td>etx</td><td>019</td><td>dc3</td><td>035</td><td>#</td><td>051</td><td>3</td><td>067</td><td>C</td><td>083</td><td>S</td><td>099</td><td>c</td><td>115</td><td>s</td></tr>
<tr><td>004</td><td>eot</td><td>020</td><td>dc4</td><td>036</td><td>$</td><td>052</td><td>4</td><td>068</td><td>D</td><td>084</td><td>T</td><td>100</td><td>d</td><td>116</td><td>t</td></tr>
<tr><td>005</td><td>enq</td><td>021</td><td>nak</td><td>037</td><td>%</td><td>053</td><td>5</td><td>069</td><td>E</td><td>085</td><td>U</td><td>101</td><td>e</td><td>117</td><td>u</td></tr>
<tr><td>006</td><td>ack</td><td>022</td><td>syn</td><td>038</td><td>&</td><td>054</td><td>6</td><td>070</td><td>F</td><td>086</td><td>V</td><td>102</td><td>f</td><td>118</td><td>v</td></tr>
<tr><td>007</td><td>bel</td><td>023</td><td>etb</td><td>039</td><td>'</td><td>055</td><td>7</td><td>071</td><td>G</td><td>087</td><td>W</td><td>103</td><td>g</td><td>119</td><td>w</td></tr>
<tr><td>008</td><td>bs</td><td>024</td><td>can</td><td>040</td><td>(</td><td>056</td><td>8</td><td>072</td><td>H</td><td>088</td><td>X</td><td>104</td><td>h</td><td>120</td><td>x</td></tr>
<tr><td>009</td><td>tab</td><td>025</td><td>em</td><td>041</td><td>)</td><td>057</td><td>9</td><td>073</td><td>I</td><td>089</td><td>Y</td><td>105</td><td>i</td><td>121</td><td>y</td></tr>
<tr><td>010</td><td>lf</td><td>026</td><td>eof</td><td>042</td><td>*</td><td>058</td><td>:</td><td>074</td><td>J</td><td>090</td><td>Z</td><td>106</td><td>j</td><td>122</td><td>z</td></tr>
<tr><td>011</td><td>vt</td><td>027</td><td>esc</td><td>043</td><td>+</td><td>059</td><td>;</td><td>075</td><td>K</td><td>091</td><td>[</td><td>107</td><td>k</td><td>123</td><td>{</td></tr>
<tr><td>012</td><td>np</td><td>028</td><td>fs</td><td>044</td><td>’</td><td>060</td><td><</td><td>076</td><td>L</td><td>092</td><td>␣</td><td>108</td><td>l</td><td>124</td><td>|</td></tr>
<tr><td>013</td><td>cr</td><td>029</td><td>gs</td><td>045</td><td>-</td><td>061</td><td>=</td><td>077</td><td>M</td><td>093</td><td>]</td><td>109</td><td>m</td><td>125</td><td>}</td></tr>
<tr><td>014</td><td>so</td><td>030</td><td>rs</td><td>046</td><td>.</td><td>062</td><td>></td><td>078</td><td>N</td><td>094</td><td>ˆ</td><td>110</td><td>n</td><td>126</td><td>˜</td></tr>
<tr><td>015</td><td>si</td><td>031</td><td>us</td><td>047</td><td>/</td><td>063</td><td>?</td><td>079</td><td>O</td><td>095</td><td>_</td><td>111</td><td>o</td><td>127</td><td>del</td></tr>
</table>

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

```c
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

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()`

```c
#include <stdio.h>

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

  char c = 42;

  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

```c
#include <stdio.h>

int main() {
  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
 35 #
105 i
110 n
 99 c
108 l
117 u
100 d
101 e
 32
 60 <
...
```

```
$ ./fread-bytes fread-bytes
127
 69 E
 76 L
 70 F
  2
  1
  1
  0
  0
  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.