C-string format with scanf/printf

```c
char myCString[4];
int intVar;
scanf("%3s", &intVar ); /*reads up to 3 chars
   and stores them PLUS \0 in the 4-byte var. intVar*/
scanf("%3s", myCString); /*DITTO into the 4-byte byte array*/
/* DIFFERENT from other output formats! */
printf("%s", &intVar); /* Print a C-String! */
printf("%s", myCString); /* Another C-string */
```

C strings are null-terminated char arrays

- They go by the address of their first char.
- In C/C++, with array char myChArray[56];
  myChArray (no brackets!) denotes the (const) ADDRESS OF the first character.
- myChArray is equivalent to &myChArray[0]
Bit - Binary Digit

Basic Unit of Information stored and manipulated in our computers. Computer\(^1\) hardware stores & manipulates & transmits all data \textbf{digitally}: which means with on/off, 0/1, \textbf{true}/\textbf{false} (usually) electrical\(^2\) signals.

A \underline{bit} (binary digit) is a single unit of memory or transmission that can have only 2 possible values.

\(^1\)and much other popular electronic product
\(^2\)also optical and magnetic
A representation is a coding scheme to give meaning to bit strings (sequences). Different kinds of data are each represented by different meanings we give to sequences of bits. Some hardware (printers, keyboards e.g.) embodies particular representations: 01000010 makes standard printers print B. The base or radix 2 (binary) number system is used by present computers to represent non-negative integers.
Example: 0001 1100 1000 0110

\[ 0 \cdot 2^{15} + 0 \cdot 2^{14} + 0 \cdot 2^{13} + 1 \cdot 2^{12} \\
+ 1 \cdot 2^{11} + 1 \cdot 2^{10} + 0 \cdot 2^{9} + 0 \cdot 2^{8} \\
+ 1 \cdot 2^{7} + 0 \cdot 2^{6} + 0 \cdot 2^{5} + 0 \cdot 2^{4} \\
+ 0 \cdot 2^{3} + 1 \cdot 2^{2} + 1 \cdot 2^{1} + 0 \cdot 2^{0} \]

\[= (0+0+0+4096)+(2048+1024+0+0)+(256+0+0+0)+(0+4+2+0)\]

\[= 7302\]
## Computer Experts all know their powers of 2:

| $2^1$ | 2 | $2^{11}$ | 2048 = 2$Ki$
|-------|---|----------|------------------|
| $2^2$ | 4 | $2^{12}$ | 4096 = 4$Ki$
| $2^3$ | 8 | $2^{13}$ | 8192 = 8$Ki$
| $2^4$ | $16_{\text{ten}} = 0\times10$ | $2^{14}$ | 16,384 = 16$Ki$
| $2^5$ | $32_{\text{ten}} = 0\times20$ | $2^{15}$ | 32,768 = 32$Ki$
| $2^6$ | $64_{\text{ten}} = 0\times40$ | $2^{16}$ | 65,536 = 64$Ki$
| $2^7$ | $128_{\text{ten}} = 0\times80$ | $2^8$ | 256
| $2^9$ | 512 | $2^{10}$ | 1024 = 1$\text{kibi} = 1\text{Ki}$
| $2^{20}$ | 1,048,576 = 1$\text{Mi}$ | $2^{30}$ | 1,073,741,824 = 1$\text{Gi}$

"Thus 1024 bytes of storage is officially a kibibyte, not a kilobyte. However, computer professionals generally dislike this unit (they say it sounds like a cat food) so the ambiguity in the size of a kilobyte persists. The prefix is a contraction of "kilobinary." The symbol $Ki$-, rather than $ki$-, was chosen for uniformity with the other binary prefixes ($Mi$-, $Gi$-, etc.)." (Russ Rowlett, UNC)
How to convert a number binary; SAME IDEA for converting a number to English!

Let \( X \) be a variable initialized with the number. Repeat until done:

1. If \( X == 0 \) write the remaining bits and go home. Otherwise, guess or figure out (by dividing) the largest power of 2 that “fits”
   Specifically, what is the largest power of two which is less than or equal to \( X \)?

2. Write zeros for any bits that should be zero. (for English, write nothing.) Write the bit for that power of 2.

3. Compute \( X = X - \) (that power of 2).

Example? 7302.
How to access individual bits in C/C++/Java (1) Bitwise Logical Operators (2) Shift operators

Bitwise Logical Operators

apply to integer type objects (char, short, int, long int and their unsigned counterparts) and give integer results defined in terms of bits.

Advice: Use C/C++ unsigned long int for bitstrings when 32 bits are needed. (long guarantees 32 bits from ANSI C.)

“All unsigned types use straight binary notation, regardless of whether the signed types use 2-s complement, 1-s compl. or sign magnitude ... the sign bit is treated as an ordinary bit.” (Harbison & Steele)

---

3 also bool after conversion to int

4 C/C++ on general purpose computers uses 2-s complement integers, but other sign representations are possible.
Since the LC-3 Instruction Set Architecture (1) uses (like all others) bit fields in the machine language code, (2) and has bitwise operations, C++ bitwise operations are useful to simulate LC-3 in C++/C/Java.

First, single bit “Boolean” or truth value operations:
Unary “NOT” (complement): (also denoted by ¬, ⌜, ~)

<table>
<thead>
<tr>
<th>x</th>
<th>NOT(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Binary “AND,” “(inclusive) OR,” “EXCLUSIVE OR”:
(Binary means “TWO OPERANDS” here)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x AND y</th>
<th>x OR y</th>
<th>x XOR y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Other Symbols: ∧, &, · | ∨, |, + | eor, ⊕
Other names for logic values:

- **1**: true, on, asserted, enabled
- **0**: false, off, deasserted, disabled

“on, asserted, enabled” etc. are popular in electrical and computer engineering circles.
C++ bitwise operators apply Boolean operations to the individual bits of the binary integer representations.

```c++
unsigned char X = 0x0C; // 0000 1100 (binary)
unsigned char Y = 0x0A; // 0000 1010

(X & Y) == 0x04 // 0000 1000 AND
(X | Y) == 0x0E // 0000 1110 OR
(X ^ Y) == 0x06 // 0000 0110 EOR
(~ X) == 0xF3 // 1111 0011 COMPL
```
Caution: It’s a common bug to confuse Bitwise operations with Logical AND, OR, NOT:

\[(X \&\& Y) == 1 \text{ if } X!=0 \text{ and } Y!=0, \text{ 0 otherwise}\]
\[(X || Y) == 1 \text{ if } X!=0 \text{ or } Y!=0, \text{ 0 otherwise}\]
\[(! X) == 0 \text{ if } X!=0, \text{ 1 if } X==0\]

- In C/C++, consider any non-zero int or pointer as “true,” and 0 as “false.”
- Many C programmers write `#define true 1` and `#define false 0`
- In C++/Java, `true` and `false` are literals (constants) of type `bool` (`boolean` in Java).
- (In C++, non-zero ints or pointers are converted to `bool` `true`, zero converts to `false`, `true` converts to 1 and `false` to 0).
Shifts For integral $X$, $AMT$, $AMT \geq 0$

- $X \ll AMT$ is $X$ shifted Left $AMT$ bit positions. Zero bits are shifted in from the right.
- Example: $(0x000F8001 \ll 3) == 0x007C0008$
- For unsigned $X$ or $X \geq 0$  
  $X \gg AMT$ is $X$ shifted Right $AMT$ bit positions. Zero bits are shifted in from the left.
- Example: $(0x007C0008 \gg 3) == 0x000F8001$
- For signed $X$, $X < 0$, in $X \gg AMT$, whether 0s or 1s are shifted in is IMPLEMENTATION DEPENDENT!!
Bitwise Op. Applications

1. Extract or compose bit fields when format is externally defined. (Hardware simulation, device driver software, network packet analysis/synthesis).

2. Work on small, fixed universe subsets efficiently. (HS 7.6, ios flags, Strou. (6.2.4) and p.616-7.)

3. Efficient special case arithmetic operations:

   ```c
   if( X & 3 ) { /* X is not a multiple of 4 */ }
   Y = X & (~0x3FF);
   /* round down to nearest 1K multiple */
   if( (X & 1) == 0 ){ /* X is even */ }
   ```
How would you sort, using about 1 Megabyte of memory, a few million 7 digit telephone numbers from a disk file once every 1/2 hour, so that we can rapidly tell if a number is assigned?

A discussion is given in Jon Bentley’s “Programming Pearls” column in the December 1999 issue of *Dr. Dobb’s Journal*. 
How to access individual bits in C/C++/Java

Bitwise operations

- & Bitwise AND
  ```
  int I; 0x80000000 & I equals 0 if the top-order bit (bit 31) is 0; equals 0x80000000 ≠ 0 if that bit is 1.
  ```
- | Bitwise OR
  ```
  I = I | 0x8000000; makes bit 31 of I become (or stay) 1. It “sets” bit 31.
  ```
How to access individual bits in C/C++/Java

Bitwise operations

- & Bitwise AND int I; 0x80000000 & I equals 0 if the top-order bit (bit 31) is 0; equals 0x80000000 ̸= 0 if that bit is 1.
- | Bitwise OR I = I | 0x80000000; makes bit 31 of I become (or stay) 1. It “sets” bit 31.
  What does I = I & 0x7FFFFFFFF; do?
How to access individual bits in C/C++/Java

Bitwise operations

- & Bitwise AND int I; 0x80000000 & I equals 0 if the top-order bit (bit 31) is 0; equals 0x80000000 ≠ 0 if that bit is 1.

- | Bitwise OR I = I | 0x80000000; makes bit 31 of I become (or stay) 1. It “sets” bit 31. What does I = I & 0x7FFFFFFF; do? “clears” bit 31.

- ~ Bitwise NOT ~0x7FFFFFFF = ?

Bit-shift operations

- I << k SHIFTS the bits LEFT k positions. k can be constant or variable.

- I >> k Guess what?
How to access individual bits in C/C++/Java

Bitwise operations

- & Bitwise AND int I; 0x80000000 & I equals 0 if the top-order bit (bit 31) is 0; equals 0x80000000 ≠ 0 if that bit is 1.
- | Bitwise OR I = I | 0x80000000; makes bit 31 of I become (or stay) 1. It “sets” bit 31.
  What does I = I & 0x7FFFFFFF; do? “clears” bit 31.
- ~ Bitwise NOT ~0x7FFFFFFF = ? 0x80000000

Bit-shift operations

- I << k SHIFTS the bits LEFT k positions. k can be constant or variable.
- I >> k Guess what?
Addition in Decimal

\[
\begin{array}{c}
3 & 6 \\
8 & 7 \\
\hline
1 & 3 & 8 & 7 \\
\end{array}
\]
Addition in Decimal

\[
\begin{array}{c}
1 \text{ carries} \\
3 6 \\
8 7 \\
\hline
3
\end{array}
\]

\[
\begin{array}{c}
6 \\
7 \\
\hline
1 3
\end{array}
\]
Addition in Decimal

\[
\begin{array}{c}
1 1 \\
3 6 \\
8 7 \\
\hline
2 3
\end{array}
\]

\[
\begin{array}{c}
6 \\
7 \\
\hline
1 3 \\
1 2
\end{array}
\]
Addition in Decimal

\[
\begin{align*}
\text{carries} & \\
3 & 6 \\
8 & 7 \\
\hline
1 & 2 & 3 \\
\end{align*}
\]

\[
\begin{align*}
6 & 1 & 1 \\
7 & 3 & 0 \\
\hline
1 & 8 & 0 \\
\hline
1 & 2 & 0 \\
\end{align*}
\]

\[
\begin{align*}
0 & 1 \\
\end{align*}
\]
Addition in Binary: Single digits

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
\hline
0 & 1 & 1 & 1 0 = 2
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 1 0
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\hline
1 1 \; = \; 3
\end{array}
\]

Mathematics requires a binary computer to work this way!
Express these rules with LOGIC

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A

B

Carry Sum

Explain logically how $A$ and $B$ determine $Sum$ and $Carry$?
Express these rules with LOGIC

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Carry</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Sum</td>
</tr>
</tbody>
</table>

Explain logically how $A$ and $B$ determine $Sum$ and $Carry$?

$Sum = 1$ exactly when $(A = 1$ and $B = 0)$ or $(A = 0$ and $B = 1)$. 

The electronic devices called logic gates determine output signals like $Carry$ from inputs like $A$ and $B$. Typical computation time: $0.1$ nanosecond = $10^{-10}$ second.
Express these rules with LOGIC

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explain logically how $A$ and $B$ determine $Sum$ and $Carry$?

$Sum = 1$ exactly when $(A = 1$ and $B = 0)$ or $(A = 0$ and $B = 1)$.

$Carry = 1$ exactly when $(A = 1$ and $B = 1)$. 

The electronic devices called logic gates determine output signals like $Carry$ from inputs like $A$ and $B$. Typical computation time: $0.1$ nanosecond $= 10^{-10}$ second.
Express these rules with LOGIC

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
\hline
\text{Carry} & \text{Sum} \\
0 & 0 \\
0 & 1 \\
0 & 1 \\
1 & 0 \\
\end{array}
\]

A  B

Explain logically how \( A \) and \( B \) determine \textit{Sum} and \textit{Carry}?

\( \text{Sum} = 1 \) exactly when \(( A = 1 \text{ and } B = 0)\) or \(( A = 0 \text{ and } B = 1)\).

\( \text{Carry} = 1 \) exactly when \(( A = 1 \text{ and } B = 1)\).

The electronic devices called \textit{logic gates} determine output signals like \textit{Carry} from inputs like \( A \) and \( B \). Typical computation time: \( 0.1 \text{ nanosecond} = 10^{-10} \text{ second} \).
### 3 Basic Gates: AND, OR, and NOT

**Truth Table for the AND gate (and AND Boolean Operation):**

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \text{ AND } B = A &amp; B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Carry when adding $A + B$ in binary

**Other useful gates and Boolean Operations:**

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X \text{ OR } Y = X | Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(our OR is *Inclusive*)

$$
\begin{array}{c|c|c}
W & \text{NOT } W = \sim W \\
0 & 1 \\
1 & 0 \\
\end{array}
$$
### Some Boolean Expressions

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\neg B$</th>
<th>$(A &amp; \neg B)$</th>
<th>$\neg A$</th>
<th>$(\neg A &amp; B)$</th>
<th>$(A &amp; B) \mid (\neg A &amp; B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two independent Boolean Variables</th>
<th>Intermediate Values</th>
<th>Desired Result</th>
</tr>
</thead>
</table>

This is *Sum!*
This 2-input 2-output circuit is called a Half-Adder. Abstract version:

\[
\begin{array}{c}
A \\
\hline
B \\
\end{array}
\quad
\begin{array}{c}
\text{SUM}(A,B) \\
\hline
\text{CARRY}(A,B) \\
\end{array}
\]
Digital Electronic View

A
Half Adder 1
SUM(A,B)
CARRY(A,B)

B
C

Half Adder 2
SUM(A,B,C)
CARRY(A,B,C)
Two abstracted half-adder sub-circuits:
#include <stdio.h>
unsigned char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N ); //Print N as decimal integer.
    N = N + 1;
}

overflows or wraps around when N = 255.

0 1 ... 253 254 255 0 1 2 3 ... 255 0 ... 252 253 254 255 0 1 2 3 ... 229 230 231

Something like this **absolutely must happen** since 8 bits can only distinguish $2^8 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$ different data values.
#include <stdio.h>
signed char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N );//Print N as decimal integer.
    N = N + 1;
}

N declared **signed** this time

0 1 2 3 ... 126 127 -128 -127 -126 -125 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
... -29 -28 -27 -26 -25
The reason for the difference is subtle:

- Standard C `printf("%d", integer value );` 
  %d format ALWAYS uses SIGNED interpretation.

- C uses “usual argument conversions” on (8 bit) char types to (16 or 32) bit int since the `printf` function doesn’t have declared argument types. `printf` is not type-safe!

- Bits of unsigned char variables are interpreted in unsigned binary for conversion to int.

- Bits of signed char variables are interpreted in signed binary for conversion to int.
This is correct! The first 16-bit number 31 is 00110001 The first two 0’s are in the $2^7 = 128$ and $2^6 = 64$ places.

The left-hand end, where we begin reading it, has the BIGGER valued binary digits. That is normal: 2007 means “Two THOUSAND (plus only) SEVEN”,

\[
\begin{array}{cccccccccccccccc}
31e2 & 0011 & 0001 & 1110 & 0010 & \ldots & 0011000111100010 \\
b5ea & 1011 & 0101 & 1110 & 1010 & \ldots & 1011010111101010 \\
f7ca & 1111 & 0111 & 1100 & 1010 & \ldots & 111101111001010 \\
39da & \cdots & 0011 & 1001 & 1101 & 1010 & \cdots & 0011100111011010 \\
9ddb & \cdots & 1001 & 1101 & 1101 & 1011 & \cdots & 1001110111011011 \\
ec0 & 1010 & 1110 & 1010 & 0000 & \ldots & 1010111010100000 \\
ed5e & 1100 & 1011 & 0101 & 1110 & \ldots & 1100101101011110 \\
210e & 0010 & 0001 & 0000 & 1110 & \ldots & 0010000100001110 \\
ffff & 1111 & 1111 & 1111 & 1111 & \ldots & 1111111111111111 \\
\end{array}
\]
Wrong Bit-order Endianness!

But, if we write each 8-bit binary value so the LITTLER valued digits are at the left-hand end, we get: $31_{\text{hex}} = 1000\ 1100 = 1 \cdot 2^0 + 0 \cdot 2^1 + \cdots + 1 \cdot 2^8 + 1 \cdot 2^{16} + 0 + 0$

Now, if we write the bits for each 8-bit number in this fashion:

$$\begin{align*}
31 & \quad e2 & \quad 10001100 & \quad 01000111 & \quad \text{---/} \quad 1000110001000111 \\
b5 & \quad ea & \quad 10101101 & \quad 01010111 & \quad \text{---/} \quad 1010110101010111 \\
f7 & \quad ca & \quad 11101111 & \quad 01010011 & \quad \text{---/} \quad 1110111101010011 \\
39 & \quad da & \quad \text{---\ } & \quad 10011100 & \quad 01011011 & \quad \text{---\ } & \quad 1001110001011011 \\
9d & \quad db & \quad \text{---\ } & \quad 10111001 & \quad 11011011 & \quad \text{---\ } & \quad 1011100111011011 \\
ce & \quad c0 & \quad 01110011 & \quad 00000011 & \quad \text{---\ } & \quad 01110011100000011 \\
ed & \quad 5e & \quad 10110111 & \quad 01111010 & \quad \text{---\ } & \quad 10110111011101010 \\
21 & \quad 0e & \quad 10000100 & \quad 01110000 & \quad \text{---\ } & \quad 1000010001100000 \\
ff & \quad ff & \quad 11111111 & \quad 11111111 & \quad \text{---\ } & \quad 1111111111111111
\end{align*}$$
s/0/ / and s/1/M/ result

M   MM   M   MMM
M   M   MM   M   M   MMM
MMM  MMMM  M   M   MM
M   MMM   M   MM   MM
M   MMM   MMM   MM   MM
   MMM   MM   MM
M   MM   MMM   MMMM   M
M   M   MMM
MMMMMMMMMMMMMMMMM
s/0/ / and s/1/M/ result

And how I made it with bitmap:

```
M MM M MMM
M M MM M M MMM
MMM MMMM M M MM
M MMM M MM MM
M MMM MMM MM MM
MMM MM MM MM
M MM MMM MMMM M
M M MMM
MMMMMMMMMMMMMM
```
```c
#define Mystery_width 16
#define Mystery_height 16
static unsigned char Mystery_bits[] = {
    0x31, 0xe2, 0xb5, 0xea, 0xf7, 0xca, 0x39, 0xda, 0x9d, 0xdb, 0xce, 0xc0,
    0xed, 0x5e, 0x21, 0x0e, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
```

which I edited using emacs to get

```c
0x31, 0xe2, 0xb5, 0xea, 0xf7, 0xca, 0x39, 0xda, 0x9d, 0xdb, 0xce, 0xc0,
0xed, 0x5e, 0x21, 0x0e, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
```
edited into

```
31e2          ?????????????????
b5ea          ?????????????????
etc
```
... we looked at the data in .xbm file

1. Surprise: .xbm format: C array initializer for an array of bytes, where the bits in each byte (like 0x31 and 0xe2), taken in LITTLE-ENDIAN BIT ORDER, represent a left-to-right sub-row of 8 pixels.

2. HAPPY FACT: ALL CPUs do their binary arithmetic on various length words in BIG-ENDIAN BIT ORDER. (Familiar big-endian digit order example: 2007 is in our millennium).

3. (but not all graphics standards!)

4. SAD FACT: MULTI-BYTE numeric data (from memory, files, network apps) is interpreted with different BYTE ORDER ("ENDIAN-NESS") by DIFFERENT CPU’s.
   Big Endian: SPARC from SUN (in its unix), MIPS (Nintendos)
   Little Endian: IA32 (Pentium & clone PC’s)

5. Earth’s current Internet is Big Endian.