C-string format with scanf/printf

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

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

"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 $X = X - \text{(that power of 2)}$.

Example? 7302.
Bitwise Logical Operators

apply to integer type objects (char, short, int, long int and their unsigned counterparts\(^3\)) and give integer results defined in terms of \textbf{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)\(^4\)

---

\(^3\)also bool \textbf{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 \( \bar{x} \), \( \sim x \), \( \sim \))

<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: \( \wedge, \&, \cdot \), \( \lor, |, + \), \( \text{eor, } \oplus \)
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.

```cpp
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 if X!=0 and Y!=0, 0 otherwise
(X || Y) == 1 if X!=0 or Y!=0, 0 otherwise
( ! X) == 0 if X!=0, 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 ≥ 0

- X << AMT is X shifted Left AMT bit positions. Zero bits are shifted in from the right.
- Example: (0x000F8001 << 3) == 0x007C0008
- For unsigned X or X ≥ 0
  X >> AMT is X shifted Right AMT bit positions. Zero bits are shifted in from the left.
- Example: (0x007C0008 >> 3) == 0x000F8001
- For signed X, X < 0, in X >> AMT, whether 0s or 1s are shifted in is IMPLEMENTATION DEPENDENT!!
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.
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?
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` 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 = ?`
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?
Bit shifts in Java

- **<<** Left-shift (as in C and C++)
- **>>** Signed Right-shift.
- **>>>** Unsigned Right-shift.

The ambiguity allowed in standard C/C++ on whether \( x \gg \ldots \) on signed integral types \( x \) is does sign-extension or not has been removed by the Java designers.

In Java, all integral types are **signed** (8-bit byte, 16-bit short, 32-bit int, and 64-bit long) except for the 16-bit char type.