CSC105, Introduction to Computer Science
Data Representation on Computers

Numeric Representations.

As we have seen, all forms of data and instructions are encoded on computer systems as binary strings. A **binary string** is a sequence of binary digits (bits) of some standardized length. Most binary strings have lengths that are powers of two.

- **byte** = 8 contiguous bits
- 2 bytes = 16 bits
- 4 bytes = 32 bits, etc.

Large quantities of bits have standard names too.

• **kilobyte** (**KB**)	= 2^{10}	= 1,024 bytes
• **megabyte** (**MB**)	= 2^{20}	= 1,048,576 bytes
• **gigabyte** (**GB**)	= 2^{30}	= 1,073,741,824 bytes
• **terabyte** (**TB**)	= 2^{40}	= 1,099,511,627,776 bytes

These quantities are used to express the size of files, storage capacities, memories, etc.

A **data type** is a binary representation of some form of information. Number, for example, is usually represented on computer systems by way of several different data types. Keep in mind that we are talking about coding schemes. Obviously, there are not really different *kinds* of numbers. But, there are different *representations* of numbers.

- **unsigned integers** [*8, 16 bits, possibly larger*]
- **signed integers** (twos complement) [*32 or 64 bits*]
- **floating point numbers** (real numbers) [*32 or 64 bits*]
- other specialized forms, e.g., BCD [*variable length*]

Signed integers are usually encoded using **twos complement**. This scheme has these advantages.

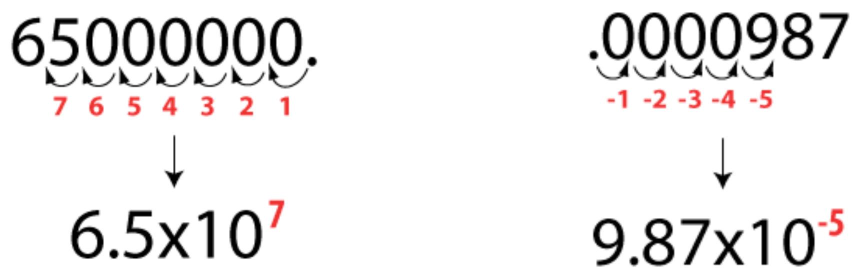
- the most significant bit can be interpreted as the **sign bit**

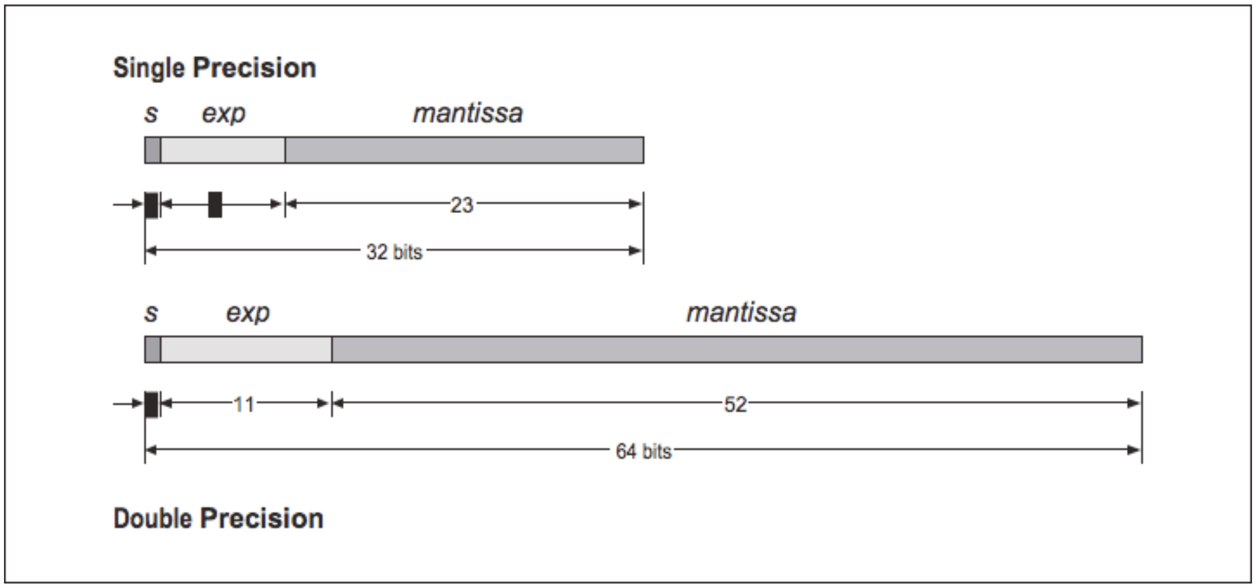
  0 = positive

  1 = negative
- arithmetic is easier: e.g., subtraction can be calculated by adding the first value to the complement of the second value.

Floating Point

In the lab, you worked with examples of both unsigned and signed integers. Floating point is very different. It is used to represent real numbers. The encoding scheme models **scientific notation**. We use scientific notation to capture the scale and significance of numbers in a more concise form. For example, using standard notation, large and small values are often represented with leading or trailing zeros. These zeros add no significance to the number—but indicate whether the number is large or small (scale). Scientific notation makes for a more compact notation.





Floating point numbers encode and store three separate pieces of information.

- a bit representing the sign of the value (0 = positive; 1 = negative).
- several bits denoting the exponent value (i.e., power of 2).
- the significant digits in the form of binary fraction in which the binary point is understood as left of the first binary digit (0.1*ddddddd*, where *d* = binary digit).

Because number codes are finite and numbers are not, computers can commit errors when they try to express values that are

- too big for the precision. (*overflow*)
- too small for the precision. (*underflow*)

The number of bits used for a data code tells us the possible range of values that the data type can express. For examples,

- one byte = 2^8 = 256 possible values (0…255 in binary)
- 32 bits = 2^{32} = 4,294,967,296 possible values (0…4294967295 in binary)

Thus, when designing a code, one must determine how many meaningful values will need to be represented. Naturally, as precisions getter larger, it costs more to store and transmit that data. (Consider also that it is usually collections of these values and not just single values.)

- a 12 megapixel camera captures images at a resolution of 4256 × 2832 pixels. When viewing these images, each color pixel may be expanded to 36 or 42 bits. Thus, the size of the image

  433,907,712 bits (54,238,464 bytes or ≈ 51 MB)

  506,225,664 bits (63,278,208 bytes or ≈ 60 MB)

(Images are seldom stored at these sizes. They are usually compressed for storage and decompressed for viewing, printing, etc.)

Since binary codes can be lengthy, it is convenient to represent them in more concise formats. Two popular notations are

- **octal** (base-8)
- **hexadecimal** (base-16)

The advantage of using these bases is that they permit easy conversions to and from binary.

Each octal digit represents **3** consecutive binary digits or bits.

000	**=**	**0**
001	**=**	**1**
010	**=**	**2**
011	**=**	**3**
100	**=**	**4**
101	**=**	**5**
110	**=**	**6**
111	**=**	**7**

9. Each hex digit represents **4** consecutive binary digits or bits.

0000	**=**	**0**
0001	**=**	**1**
0010	**=**	**2**
0011	**=**	**3**
0100	**=**	**4**
0101	**=**	**5**
0110	**=**	**6**
0111	**=**	**7**
1000	**=**	**8**
1001	**=**	**9**
1010	**=**	**A**
1011	**=**	**B**
1100	**=**	**C**
1101	**=**	**D**
1110	**=**	**E**
1111	**=**	**F**

1 As we will see, character code schemes are often displayed in hexadecimal notation for greater readability.

Translating from hex to decimal is not difficult. Here is an example.

U+03B1 (Greek letter alpha) = **0000 0011 1011 0001** in binary =

$2^9 + 2^8 + 2^7 + 2^5 + 2^4 + 2^0 = 512 + 256 + 128 + 32 + 16 + 1 =$ **945**.

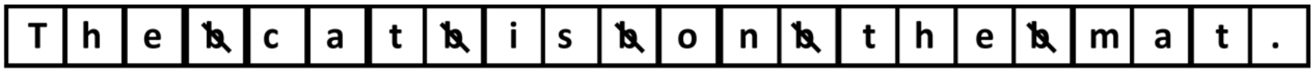
Representing Text

Numbers are interesting, but our focus in this course will be using computational thinking to solve problems that mostly involve text or the written word.

Most conventional systems for representing text have used separate symbols called **character sets**. (Think of older printing technologies where the typesetter created the printed page by building lines of text using individual letters, etc.) On computer systems, text has been represented as **character codes**. In other words, each symbol of text has a binary value assigned to it. Character codes typically have the following characteristics.

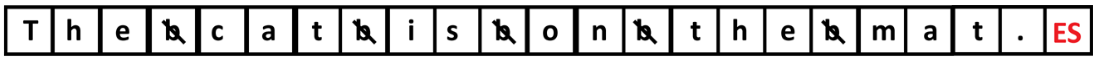
- there is a one-to-one encoding of the text symbol set, i.e., each symbol has a **unique code.**
- there is a uniform number of bits assigned to represent each character, i.e., **uniform precision.**
- the order of values assigned models the lexical order of the symbols, i.e., **enforces the collating sequence.**

We can think of text as strings of character codes. For example,



Each character is represented by a uniform number of binary digits or bits. The string, however, is a variable number of units. So, we must somehow signify where the string ends. Thus, character codes typically have **special** or **invisible characters** that perform this and related functions.

What we need here is an **end-of-string** indicator. Most character codes usually have such an invisible character code—although operating systems may specify their own requirements.



Invisible character codes also play a role in formatting text. For example, the blanks shown above are invisible characters. Likewise, **end-of-line** and **carriage returns** must also be encoded. Other invisible character codes include **form-feeds** (new page), **tabs**, etc.

Technically, a **text string** is stored in primary memory. An entire document is no more than a single variable-length text string. Here the string is **23** units in length. Documents can have thousands and even millions of characters. The point is that—from the computer's perspective—the document is one long string or serial structure. It does not have the three-dimensional organization that we are accustomed to.

We often want to preserve text objects elsewhere for more permanent storage. To accomplish this, the operating system stores the strings as **files**, that is **text files**.

T	h	e	␢	c	a	t	␢	i	s	␢	o	n	␢	t	h	e	␢	m	a	t	.	EF

The only practical difference will be the conventional **end-of-file** character and some header information stored in front.

One of the earliest computer character codes employed by most systems is **ASCII** code. This stands for **American Standard Code for Information Interchange**.

- its origin is for **data transmission** over teletype machines.
- ASCII is a compact code of **8-bits** or **one byte** per symbol (although the original was 7-bits).
- it is limited to the **Roman alphabet** (upper- and lower-cases), numerals, punctuation, special symbols ($, #, @, etc.), and control characters (i.e., invisible characters).
- ASCII preserves the **collating sequence**.

Lots of legacy text materials are still stored in ASCII format.

ASCII is clearly limited when treating documents that use other alphabets. Consequently, in 1991, a consortium of U.S. IT companies and a working group from the ISO (International Standards Organization) joined together to develop a more global coding system for representing text.

The result is **Unicode**, which is the dominant code on computer systems today. The original design requirements were these.

- **backward compatible with ASCII.**
- capable of representing related alphabets from **European languages**.
- capable of representing **Middle Eastern scripts** (right-to-left)
- and **Asian scripts**

To accommodate so many different codes, the size of a single symbol code had to be made bigger. The original plan was to expand each symbol to a 16-bit code (two bytes).

- 2^{16} = 65,536 different values or codes.

Later, this was expanded to 32-bits (four bytes).

- 2^{32} = > 4 billion+ different code possibilities.

Here is an example of how it works for backward compatibility with ASCII.

• ASCII ‘c’	=							0110	0011
• Unicode ‘c’	=	0000	0000	0000	0000	0000	0000	0110	0011
in hexadecimal,		0	0	0	0	0	0	6	3

i.e., U+00000063.

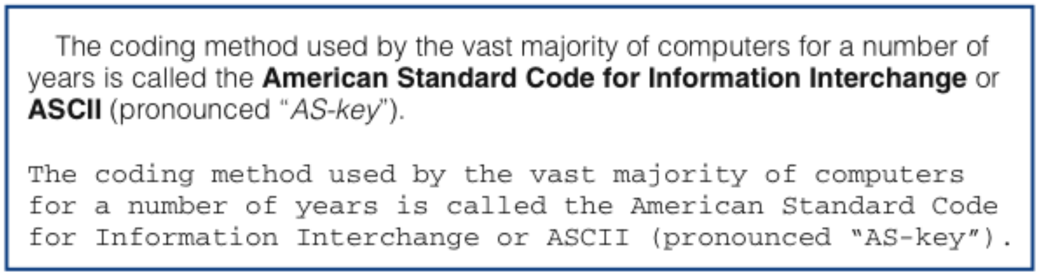
For applications, there are several Unicode variations that permit shorter precision codes for convenience. For example, **UTF-8** is a variable-length coding scheme supported by the Internet Engineering Task Force (IETF) for Internet applications. It is the *de facto* standard for the World Wide Web and is also common for e-mail applications.

UTF is not a separate code but rather a *mapping* of Unicode. In particular, UTF-8 encodes each character symbol using one to four bytes (or octets). Thus, the first 128 symbols of Unicode (and ASCII) are preserved as single octets. UTF-8 has also become a popular coding standard for both operating systems and programming languages.

Formatting Text.

Most organizations and enterprises generate large amounts of text documents that must be stored for archival and general use. There is a problem, however, with the compatibility of these documents.

Traditional character codes like ASCII and Unicode lack any special formatting features for representing text. In fact, files composed of Unicode (or ASCII) symbols are called **plaintext** files. Compare the two passages



The coding method used by the vast majority of computers for a number of years is called the **American Standard Code for Information Interchange** or **ASCII** (pronounced "*AS-key*").

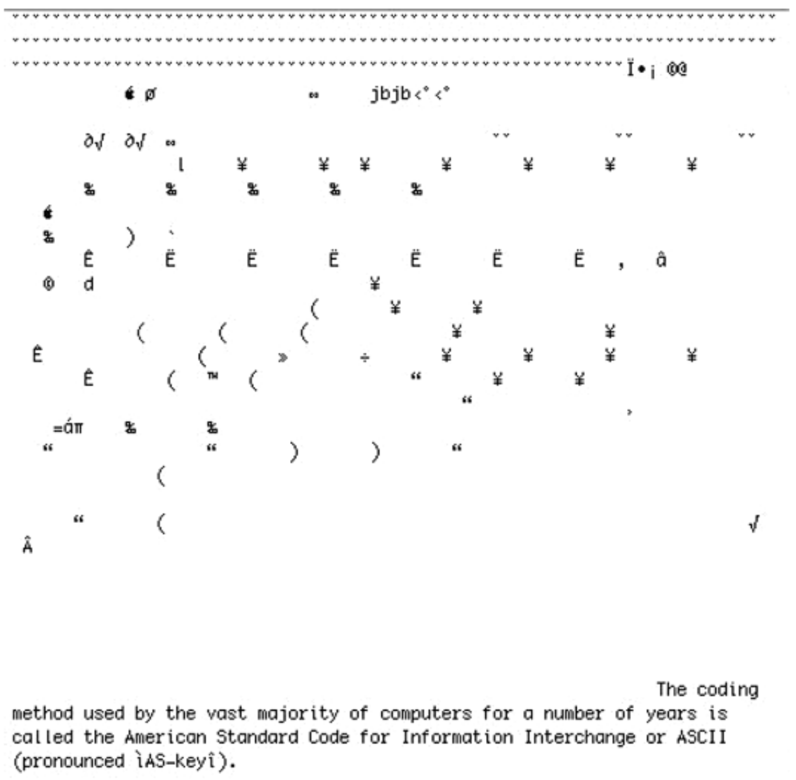
```
The coding method used by the vast majority of computers
for a number of years is called the American Standard Code
for Information Interchange or ASCII (pronounced "AS-key").
```

The former exhibits

- **font face** (choice of symbol style).
- **font size**.
- special styling such as **italics** and **boldface**.
- other format control, such as **tab position** and **margin control**.

The bottom version is rightly dubbed **plain text**.

If we were to examine the former sample as a plain text file, we would see something very different.



```
                                                                          Ï•¡ @@
        ¢ ø            ••    jbjb<°<°
      ∂√ ∂√ ••
           l    ¥    ¥ ¥    ¥    ¥    ¥    ¥
      %    %    %    %    %
  ¢
  %      )  `
      Ê    Ë    Ë    Ë    Ë    Ë    Ë  ,  â
  ®   d                       ¥
                        (    ¥    ¥
         (    (    (        ¥         ¥
 Ê            (    »    ÷    ¥    ¥    ¥    ¥
      Ê    (  ™  (       “    ¥    ¥
                              “
   =áπ   %    %
  “             “    )    )    “
           (
    “      (                                        √
 Â

                                                    The coding
method used by the vast majority of computers for a number of years is
called the American Standard Code for Information Interchange or ASCII
(pronounced ìAS-keyî).
```

Most of it is unreadable. This is because extra coding is introduced. The problem, however, is that these codes are proprietary—and seldom compatible.

How do we solve this compatibility problem? For example, imagine that we manage documents for a large international corporation with offices or locations around the world.

At first glance, we seem to have only two (extreme) alternatives.

i. **store everything as plaintext.**

strengths:	compatible
weaknesses:	loss of too much layout and format information.

ii. **store everything using a single proprietary software application.**

strengths:	compatible
weaknesses:	every machine must have a license.

In fact, there are a few other choices available.

Microsoft has developed its own solution by offering a common denominator format called **Rich Text Format** or **RTF**. It offers a lot of basic format control such as fonts, sizes, margin control, etc. But, not all features are represented. Because it is royalty-free, other word processing applications can use it to store files.

This suggests a third possible solution.

iii. **store everything as RTF.**

strengths:	more compatibility, less expensive than ii
weaknesses:	loss of some layout and format information.

Another development is that of Adobe's **Portable Document Format** or **PDF**. A PDF document is actually includes a graphic representation of the "look" of the document rather than a simple text code. Thus, PDF documents preserve the original layout and design very accurately.

As a bonus, Adobe distributes the **reader software** for free. So, documents in PDF form can be viewed easily on virtually any machine. But, because they are delivered like graphic images, they cannot by edited without Adobe's proprietary **editing (writing) software**, which is expensive.

iv. **store everything as PDF.**

strengths: compatible; readers are free

weaknesses: editing documents can be expensive because special software must be purchased and installed on any machine that edits.

A number of organizations have adopted an entirely different solution: create Web pages for the documents! And, as you may have noticed, a lot of companies archive their text information in this form. How good a solution is it?

Web documents are actually text that includes **markup** symbols. The markup codes instruct the client computer (that received the document from some Web server) how to display it. Web pages are typically encoded using **Hypertext Markup Language** (**HTML**) or **Extended HTML** (**XHTML**).

v. **store everything as HTML or XHTML.**

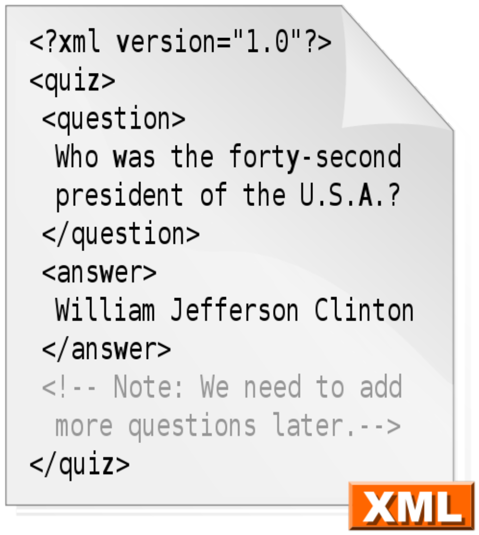
strengths: compatible, inexpensive to distribute.

weaknesses: conversion of legacy documents to HTML is not trivial (and can be expensive).

Here is a short example,

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!-- this is a comment -->
<!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
  <head>
    <title>The title</title>
  </head>
  <body>
    <p>Some paragraph.</p>
  </body>
</html>
```

XML is another markup language that can be used to organize text information from both documents and databases. XML is actually an ancestor of HTML and XHTML. Specifically, **Extensible Markup Language** (**XML**) is a markup language that defines its own set of rules for encoding text documents. It is intended to be a format that makes the document readable by both humans and machines.



```
<?xml version="1.0"?>
<quiz>
 <question>
  Who was the forty-second
  president of the U.S.A.?
 </question>
 <answer>
  William Jefferson Clinton
 </answer>
 <!-- Note: We need to add
  more questions later.-->
</quiz>
```

The example shows how an arbitrary document type can include semantic information along with the text content.

A database often contains information that requires some interpretation and reorganization by human users before the data can be employed by applications automatically. Take, for example, this simple text database. The first line indicates field names. The remaining lines of text are individual employee records,

```
name,dateOfBirth,dept,jobTitle
John,1962-11-24,accounting,senior accountant
Tina,1962-09-26,administration,manager
Karen,1972-01-10, marketing,graphic designer
Michael,1978-02-11,research,programmer
Sandra,1976-10-26,marketing,account manager
```

When converted to XML format, it might look like this.

```
<employees>
     <employee>
          <name>John</name>
          <dateOfBirth>1962-11-24</dateOfBirth>
          <dept>accounting</dept>
          <jobTitle>senior accountant</jobTitle>
     </employee>
     <employee>
          <name>Tina</name>
          <dateOfBirth>1962-09-26</dateOfBirth>
          <dept>administration</dept>
          <jobTitle>manager</jobTitle>
     </employee>
     <!-- and so forth -->
</employees>
```

The principal advantage is that this data could be used by other applications without the need for humans reorganizing the data.

vi. **store everything as XML.** *(same as before)*

strengths: compatible, inexpensive to distribute.

weaknesses: conversion of legacy documents to XML is not trivial (and can be expensive).