A byte is 8 bits and can contain a numeric value in the range 0-255. Computers don't understand Latin, Cyrillic, Hindi, Arabic character sets!

Alphanumeric and special characters of the Latin alphabet are stored in memory as integer values encoded using the American standard code for the interchange of information (ASCII) code.

Other codes requiring 16 bits per character have been developed to support languages having large number of written symbols.
A program for displaying the ASCII encoding scheme

The correspondence between decimal, hexadecimal, and character representations can be readily generated via the following simple program.

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int c;
    c = ' ';
    /* Same as c = 32 or c = 0x20 */
    while (c <= 'z') {
        printf("%3d %02x %c \n", c, c, c);
        c = c + 1;
    }
}
```
Output of the ASCII table generator

:  
class/1070/examples ==> gcc -o p2 p2.c  
class/1070/examples ==> p2 | more  
32 20  
33 21 !  
34 22 "  
35 23 #  
36 24 $  
37 25 %  
38 26 &  
39 27 '  
40 28 (  
41 29 )  
42 2a *  
43 2b +  
44 2c ,  
45 2d -  
46 2e .  
47 2f /  
48 30 0  
49 31 1  
:
Output of the ASCII table generator

<table>
<thead>
<tr>
<th>ASCII Value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>`</td>
</tr>
<tr>
<td>59</td>
<td>,</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>:</td>
</tr>
<tr>
<td>69</td>
<td>Y</td>
</tr>
<tr>
<td>70</td>
<td>Z</td>
</tr>
<tr>
<td>92</td>
<td>[</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
</tbody>
</table>
Output of the ASCII table generator

120 78 x
121 79 y
122 7a z

There are also a few special characters that follow z.
Control characters:
The ASCII encodings between decimal 0 and 31 are used to encode what are commonly called *control characters*.

Control characters having decimal values in the range 1 through 26 can be entered from the keyboard by holding down the *ctrl* key while typing a letter in the set a through z.

Some control characters have “escaped code” representations in C, but all may be written in octal.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Keystroke</th>
<th>Name</th>
<th>Escaped code</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>ctrl-D</td>
<td>end of file</td>
<td>‘004’</td>
</tr>
<tr>
<td>8</td>
<td>ctrl-H</td>
<td>backspace</td>
<td>‘\b’</td>
</tr>
<tr>
<td>9</td>
<td>ctrl-I</td>
<td>tab</td>
<td>‘\t’</td>
</tr>
<tr>
<td>10</td>
<td>ctrl-J</td>
<td>newline</td>
<td>‘\n’</td>
</tr>
<tr>
<td>12</td>
<td>ctrl-L</td>
<td>page eject</td>
<td>‘\f’</td>
</tr>
<tr>
<td>13</td>
<td>ctrl-M</td>
<td>carriage return</td>
<td>‘\r’</td>
</tr>
</tbody>
</table>
Streams

- Source of or destination for data
- In C, data are input to and read from a stream
- C uses two forms of streams: text and binary

Text stream

- consists of a sequence of characters divided into lines, with each line terminated by a newline (\n)

Binary stream

- consists of a sequence of data values, such as integer, real, or complex, using their memory representation.
Input/Output

A terminal can be associated only with a text stream
   – a keyboard can only send a stream of characters into a program
   – a monitor can only display a sequence of characters

• A file can be associated with a text or binary stream
The C run-time system automatically opens three special `FILE *`s for you at the time your program starts:

- **stdin** - Normally keyboard input (may be redirected with `< `)
- **stdout** - Normally terminal output (may be directed with `>` or `1> `)
- **stderr** - Normally terminal output (may be directed with `2> `)

Since these files are predeclared and preopenend **they must not be declared nor opened** in your program!
In order to use the I/O functions defined in the C library, you must have this compiler directive near the start of the file:

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

- Includes input functions `scanf`, `fscanf`, ...
- Includes output functions `printf`, `fprintf`, ...
Console Output

What can be output?
• Any data can be output to the display screen
  – Literal values
  – Variables
  – Constants
  – Expressions (which can include all of above)
• Integer values are stored in computer memory using a binary representation.

• To communicate the value of an integer to a human, it is necessary to produce the string of ASCII characters that correspond to the rendition of the integer in the Latin alphabet.
For example, consider the byte

\[1001\ 0100\]

- This is the binary encoding of the hex number 0x94 which is the decimal number \[9 \times 16 + 4 = 148\]. Therefore, if this number is to be rendered as a decimal number on a printer or display, three bytes corresponding to the ASCII encodings of 1, 4, and 8 must be sent to the printer or display. These bytes are expressed in hexadecimal as:

\[31\ 34\ 38\]
In the C language, *run time libraries* (RTL) provide functions that interface with the Operating System to provide I/O. Some of these functions may actually be implemented as *macros* that call the actual RTL functions.

The `printf()` function (actually a macro that converts to `fprintf(stdout, )`) is used to produce the ASCII encoding of an integer and send it to an output device or file.
Formatted I/O: Printing values:

Header File

- `stdio.h` contains the function prototype and any other definitions that are needed for the `printf` function

- **Syntax of `printf()`**
  ```c
  int printf(format-string, data-values);
  ```

- **Syntax of `fprintf()`**
  ```c
  int fprintf(file-ptr, format-string, data-values);
  ```
Formatted I/O: Printing values:

`printf`

- formats and prints its arguments as specified by the format-string. Format strings consist of three types of data
  1. Whitespace
  2. Text characters
  3. Field specification or conversion specification

These may be repeated
Formatted I/O: Printing values:

`printf`

- Plain characters in the format-string are simply copied.
- Format specifications indicate how you want to see the value represented. Format specifications are made up of the percent sign (%) followed by a conversion operator, which determines what `printf` does with its arguments:
  - `%d` or `%i` – print the next argument as a signed decimal number, like 7250. The argument should be an `int`
printf

• Conversion operators (cont’d):
  o \%f – print the next argument as a signed floating point number, like 3.14159. The argument should be a float.

  o \%lf – print the next argument as a signed double precision number, like -17.3692. The argument should be a double.
Formatted I/O: Printing values:

printf

• Conversion operators (cont’d):
  
  o \%c – print the next argument as a character, like ’B’. The argument should be a char.
  
  o \%s – print the next argument as a string, like "Go Tigers". The argument should be a char * or a char [ ].
Formatted I/O: Printing values:

`printf`

- Conversion operators (cont’d):
  - `u` – print an *unsigned* as an unsigned decimal *num’ber*
  - `%oo` – print an *unsigned* as an unsigned octal number
  - `%ox` or `%X` – print an *unsigned* as an unsigned hexadecimal number (where the letters *abcdef* are used with `x` and *ABCDEF* are used with `X`)
printf

- Conversion operators (cont’d):
  - %e or %E – print a *double* using exponential format like: \([-]\d.ddd[+\-]dd\)
    (where the *e* is replaced by *E* if the uppercase *E* is specified)
  - %\% – print a single % character
There's actually a lot more that you can specify after the percent sign. The most common ones are given next. For one thing, you can put a field width in there—this is a number that tells `printf()` how many spaces to put on one side or the other of the value you're printing. That helps you line things up in nice columns. If the number is negative, the result becomes left-justified instead of right-justified. Example:

```c
printf("%10d", x);  /* prints X on the right side of the 10-space field */
printf("%-10d", x);  /* prints X on the left side of the 10-space field */
```
Formatted I/O: Printing values:

```
printf
• You can also put a "0" in front of the number if you want it to be padded with zeros:
    int x = 17;
    printf("%05d", x); /* "00017" */
```
When it comes to floating point, you can also specify how many decimal places to print by making a field width of the form "x.y" where x is the field width (you can leave this off if you want it to be just wide enough) and y is the number of digits past the decimal point to print:

```c
float f = 3.1415926535;

printf("%.2f", f);  /* 3.14 */
printf("%7.3f", f);  /* 3.141  <-- 7 spaces across */
```
Formatted I/O: Printing integer values

\textit{data Values}

- a set of zero or more data values to be formatted
- can be a constant, variable, expression
Formatted I/O: Printing integer values

• Examples:
  
  ```
  int x = 5;
  int y = 7;
  int n = printf("Welcome to Clemson University. Go Tigers\n");
  fprintf(stdout, "Clemson University. Home of the Tigers!\n");
  n = printf("x = %d, y = %d\n", x, y);
  printf("%d + %d = %d\n", x, y, x + y);
  ```

Output:

  Welcome to Clemson University. Go Tigers
  Clemson University. Home of the Tigers!
  x = 5, y = 7
  5 + 7 = 12
Specifying field width: Recall that the *format codes* may be preceded by an optional *field width* specifier. The code `%02x` shown below forces the field to be padded with leading 0's if necessary to generate the specified field width.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int x;
    int y = 78;
    x = 'A' + 65 + 0101 + 0x41 + '\n';
    printf("X = %d \n", x);
    printf("Y = %c %3d %02x %4o \n", y, y, y, y);
}
gcc -o p1 p1.c
./p1
X = 270
Y = N 78 4e 116
```

The number of values printed by *printf()* is determined by the number of distinct format codes.
Output redirection

The `printf()` function sends its output to a logical file commonly known as the *standard output* or simply `stdout`.

When a program is run in the Unix environment, the logical file `stdout` is by default associated with the screen being viewed by the person who started the program.
The > operator may be used on the command line to cause the standard output to be *redirected* to a file:

```
./p1 > p1.output
```

A file created in this way may be subsequently viewed using the `cat` command (or edited using a text editor).

```
cat p1.output
X = 270
Y = N 78 4e 116
```
When a human enters numeric data using the keyboard, the values passed to a program are the ASCII encodings of the keystrokes that were made.

For example, when I type:

```
123.45
```

6 bytes are produced by the keyboard. The hexadecimal encoding of these bytes is:

```
31 32 33 2E 34 35 - hex
```

```
1 2 3 . 4 5 - ascii
```

To perform arithmetic using my number, it must be converted to internal floating point representation.
Input of integer data

The `scanf()` / `fscanf()` function is used to:

1. consume the ASCII encoding of a numeric value
2. convert the ASCII string to the proper internal representation
3. store the result in a memory location provided by the caller.
Input of integer data

Syntax of scanf()

```c
int scanf(format-string, address-list)
```

Syntax of fscanf()

```c
int fscanf(file-ptr, format-string, address-list)
```

Note: A successful scanf() / fscanf() returns the number of values read.
Input of integer data

As with `printf()`, a format code controls the process. The format code specifies both the input encoding and the desired type of value to be produced:

- `%d`  string of decimal characters  int
- `%x`  string of hex characters  unsigned int
- `%o`  string of octal characters  unsigned int
- `%c`  ascii encoded character  char
- `%f`  floating point number in decimal  float
- `%lf`  floating point number in decimal  double
- `%e`  floating pt in scientific notation  float
Input of integer data

Also as with `printf()` the number of values that `scanf()` will attempt to read is determined by the number of format codes provided. For each format code provided, it is mandatory that an address be provided to hold the data that is read.
Input / Output

It is *extremely important* to note that:

- the *value* to be printed is passed to *printf()*
- but, the *address* of the variable to receive the value must be passed to *scanf()*

The & operator in C is the “*address of* “ operator.
/* p3.c */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;
    int r;
    int b;
    int b;
    r = scanf("%d %d", &a, &b);
    printf("Got %d items with values %d %d \n",
            r, a, b);
}
Input of integer data

**Care and feeding of input format specifiers**

Embedding of *extra spaces* and including '\n' in `scanf()` format strings can also lead to weird behavior and should be avoided. Specification of field widths is *dangerous* unless you really know what you are doing.
Input of integer data – Effect of invalid input

/* p3.c */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;
    int r;
    int b;
    r = scanf(" %d %d ", &a, &b);
    printf("Got %d items with values %d %d \n", r, a, b);
}

If you accidentally enter a non-numeric value, `scanf()` will abort and return only 1 value. The value 4927 represents the uninitialized value of `b`.

./p3
1 t 6
Got 1 items with values 1 4927
Input of integer data

**Pitfalls of field widths:**
It is legal to specify field widths to scanf() but usually dangerous to do so!

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;
    int r;
    int b;
    r = scanf("%2d %2d ", &a, &b);
    printf("Got %d items with values %d %d \n", r, a, b);
}
```

```
./p4
123 456
```

Got 2 items with values 12 3
Detecting end-of-file with *scanf()*

*scanf()* returns the number of values it actually obtained (which may be less than the number of values requested); therefore, the proper way to test for end-of-file is to *ensure that the number of values obtained was the number of values requested.*
Input of integer data

For example,

```c
/* example.c */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;
    int r;
    int b;
    while ((r = scanf("%d %d", &a, &b)) == 2)
    {
        printf("Got %d items with values %d %d \n", r, a, b);
    }
}
```
Input of integer data

Alternate code for the previous example:

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int a;
    int r;
    int b;
    r = scanf("%d %d", &a, &b); /* read before entering */
    while (r == 2) {
        printf("Got %d items with values %d %d \n", r, a, b);
        r = scanf("%d %d", &a, &b); /* read at bottom */
    }
}
```
Input of integer data

**Input redirection**

Like the *stdout* the *stdin* may also be redirected.

```
a.out < input.txt
```

To redirect both *stdin* and *stdout* use:

```
a.out < input.txt > output.txt
```

when invoked in this manner whenever the program *a.out* reads from *stdin* via *scanf()* or *fscanf(stdin,)*, the input will actually be read from a file named *input.txt* and any data written to *stdout* will end up in the file *output.txt*. 
The necessity of format conversion

As stated previously, the %d format causes scanf() to convert the ASCII text representation of a number to an internal binary integer value.

One might think it would be possible to avoid this and be able to deal with mixed alphabetic and numeric data by simply switching to the %c format.

However, if we need to do arithmetic on what we read in, this approach does not work.
The necessity of format conversion

#include <stdio.h>
int main(void)
{
    char i, j, k;
    scanf("%c %c %c", &i, &j, &k);
    printf("The first input was %c \n", i);
    printf("The third input was %c \n", k);
    printf("Their product is %c \n", i * k);
}

./a.out
2 C 4
The first input was 2
The third input was 4
Their product is (
The necessity of format conversion

Why do we get '(' and not 8 for the answer? Since no format conversion is done the value stored in \( i \) is the ASCII code for the numeral 2 which is 0x32 = 50.

Similarly the value stored in \( j \) is 0x34 = 52.

When they are multiplied, the result is 50 \( \times \) 52 = 2600 which is too large to be held in 8 bits.

The 8 bit product is (2600 mod 256) (the remainder when 2600 is divided by 256).

That value is 2600 - 2560 = 40 = 0x28 which according to the ASCII table is the encoding of '('.
Floating point input and output

The `%e` and `%f` format codes are used to print floating point numbers in scientific and decimal format.

The **l** modifier (the letter **l**) should be used for *doubles*. Field size specification is of the form:

`field-width,number-of-digits-to-right-of-decimal`
#include <stdio.h>
int main(int argc, char *argv[]) {
    float a;
    double b;
    float c;
    double d;
    a = 1024.123;
    b = 1.024123e+03;
    scanf("%f %le", &c, &d);
    printf("a = %10.2f b = %8.4lf c = %f d = %8le \n",
           a, b, c, d);
}
gcc -o p5 p5.c
./p5
12.4356  1.4e22
a = 1024.12 b = 1024.1230 c = 12.435600 d = 1.400000e+22