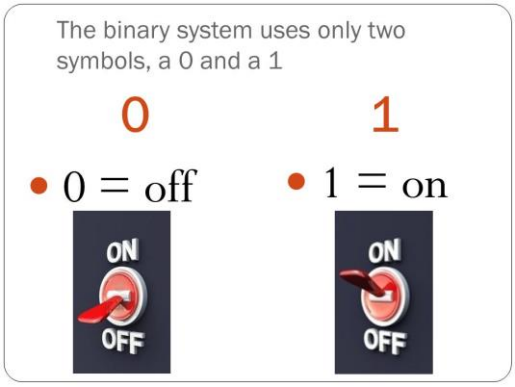
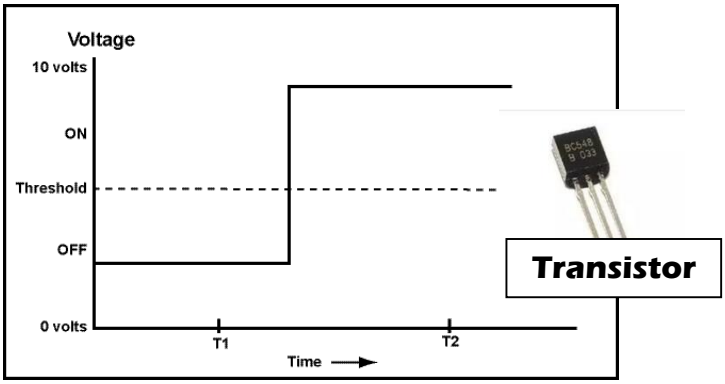
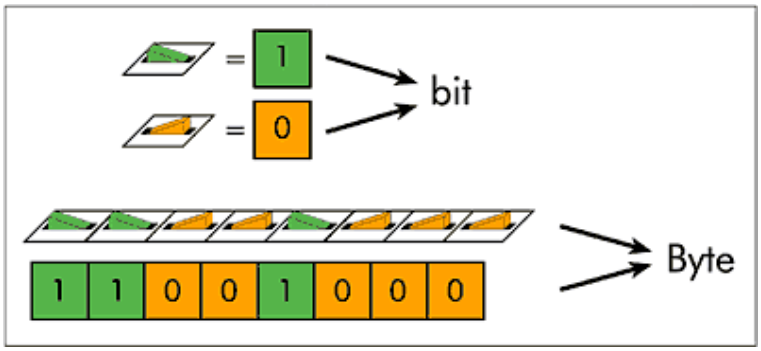
Computer Memory, Memory Allocation, and Data Types:

At the physical level, computer memory consists of a large number of **on** and **off** signals that can be stored. Each **on/off** signal is created by transistors. Transistors are, essentially, tiny switches that create areas of **low voltage** (OFF) or **high voltage** (ON) in a particular spot in the circuit. A single **on/off** signal and is called **one bit.**

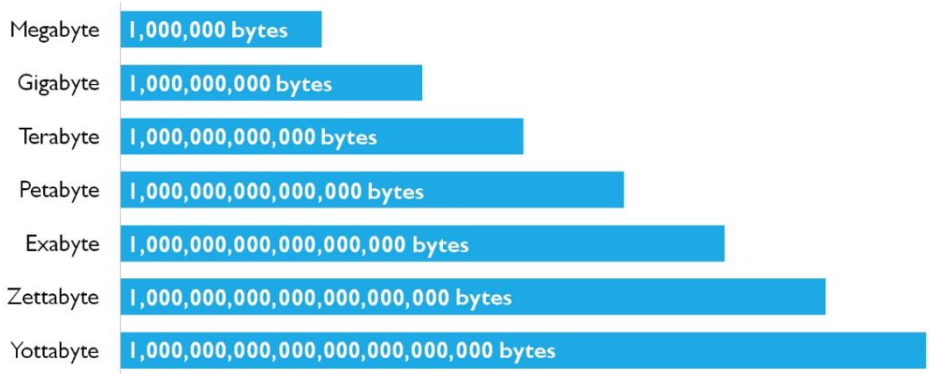




A bit is the name of **1 on/off** signal.



A **Byte** is 8 Bits. (very important!) (bits are usually grouped into **BYTES**)



A **Gigabyte** is **1 billion bytes** (that's **8 billion on/off switches**)

The Binary Number System

Since a computer is set up as a bunch of **on/off** switches, the easiest way to represent and store information is using the **Binary** number system….it only has 2 digits. "1" for **ON** and "0" for **OFF**! Perfect. You can check out a youtube video or find an online exercise, but the charts below basically shows you how it works. It's like our familiar *base 10* system but the columns **don't** go 0's, 10's, 100's, 1000's. They go 1's 2's 4's 8's 16's…..

place value in the binary system is based on 2

2^5	2^4	2^3	2^2	2^1	2^0	
32's	16's	8's	4's	2's	1's	
thirty-twos	sixteens	eights	fours	twos	ones	
		1	1	0	0	= 12

a Base-2 system

Binary Value	Decimal Representation	Decimal Value
	8 4 2 1	
0 0 0 0	0 + 0 + 0 + 0	0
0 0 0 1	0 + 0 + 0 + 1	1
0 0 1 0	0 + 0 + 2 + 0	2
0 0 1 1	0 + 0 + 2 + 1	3
0 1 0 0	0 + 4 + 0 + 0	4
0 1 0 1	0 + 4 + 0 + 1	5
0 1 1 0	0 + 4 + 2 + 0	6
0 1 1 1	0 + 4 + 2 + 1	7
1 0 0 0	8 + 0 + 0 + 0	8
1 0 0 1	8 + 0 + 0 + 1	9
1 0 1 0	8 + 0 + 2 + 0	10

You Try:

Find the **decimal number** represented by the **binary number** stored in each of these **Bytes**:

11001001
01000111
10000110
00010001
10001000
00111110
01010101
10101010

Answers on the next page.

Answers:

201
71
134
17
136
62
85
170

BYTES

To store information, computers always group bits into **BYTES.**

In a computer, **bits** are usually grouped in packs of 8 bits or "switches" (called **one byte**). **Using Binary** One **Byte** can represent 256 different decimal numbers (0-255).

This means a single **Byte** we can have 256 different unique patterns/combinations of **on/off** signals to represent 256 different numbers…. **or anything** else we want.

Examples of different **Bytes** used to represent letters

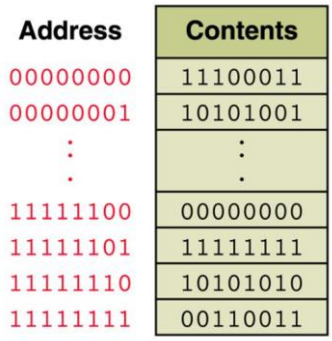
Converting the text "hope" into binary

Characters:	h	o	p	e
ASCII Values:	104	111	112	101
Binary Values:	01101000	01101111	01110000	01100101 ← 1 **Byte** (8 bits)
Bits:	8	8	8	8

ComputerHope.com

Memory Addresses

Each **Byte** has an actual **physical location** and can be given a **unique address**. We can think of all of our computer's **memory as just one giant group** of **bytes** that we can read and write.



Address	Contents
00000000	11100011
00000001	10101001
⋮	⋮
11111100	00000000
11111101	11111111
11111110	10101010
11111111	00110011

Each **Byte** can be thought to have ***an actual physical location!***

Data types

Previously, you learned that we can **declare** different types of **variables** depending on what we are storing (integer, character, decimal value). Each of the data types takes up memory. Here is a list of **Data Types** you should become familiar with:

Type	Storage size Required	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Please notice that different **amount of memory** needed for different **data types**.

You can use "C" to get the exact number of bytes needed to store a particular data type on your computer. You can use the `sizeof()` operator.

The expressions `sizeof(type)` will output the storage size of the object or type in bytes. Given below is an example to get the size of int type on any machine

Give it a try: (cut and paste)

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

int main() {
   printf("Storage size for int : %d \n", sizeof(int));

   return 0;
}
```

Float and Double variables

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file `float.h` will give you details about the binary representation of **Floats** and **doubles** numbers in your programs. The following example prints the storage space taken by a float type **and its range values and precision.**
Give it a try:

```
#include <stdio.h>
#include <float.h>
int main() {
   printf("Storage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value: %d\n", FLT_DIG );
   return 0;  }
```

Why do we care about storage space of Data Types?

The reason is very important in **some instances**. When programming the second most important objective (after making the program do what it is supposed to) is **saving memory to make the program run as fast as possible.** Complex programs that store millions of bytes of information rely on **data type** declaration to conserve and organize memory.

Variable Type	Keyword	Bytes Required	Range	Format
Character (signed)	Char	1	-128 to +127	%c
Integer (signed)	Int	2	-32768 to +32767	%d
Float (signed)	Float	4	-3.4e38 to +3.4e38	%f
Double	Double	8	-1.7e308 to + 1.7e308	%lf
Long integer (signed)	Long	4	2,147,483,648 to 2,147,438,647	%ld
Character (unsigned)	Unsigned char	1	0 to 255	%c
Integer (unsigned)	Unsigned int	2	0 to 65535	%u
Unsigned long integer	unsigned long	4	0 to 4,294,967,295	%lu
Long double	Long double	10	-1.7e932 to +1.7e932	%Lf

Pointers

Every variable has a **memory location** and every memory location has **an address**. We can Get the **address** of a variable in the following way:

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

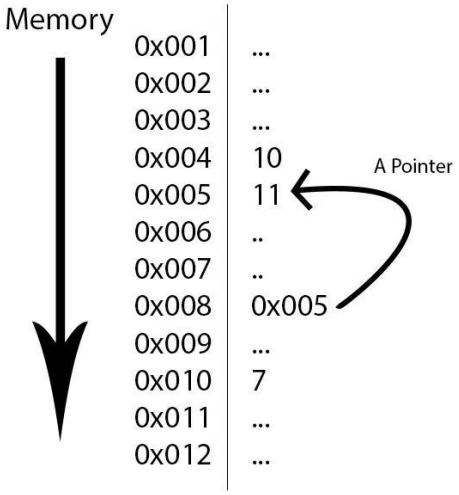
A pointer is a **variable** whose **value** is the **memory address** of another. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch;    /* pointer to a character */
```

Some C programming tasks are performed more easily with **pointers**, while some task are impossible without them.

C was developed when computers were much less powerful than they are today. Being efficient with memory usage was often vital. The raw ability to work with particular memory locations was a useful option to have.

For now, using pointers isn't necessary. Just try to remember what they are and what they do. If they pop up in the future, at least you will have some knowledge of their existence.



ASCII CODE

You might remember ASCII code from previous years of coding. Here is a refresher:

American Standard Code of Information Interchange (ASCII)

To create a link between **binary** and the **keyboard** a standard code was created that **labelled each key stroke** with a **decimal number** that could be then be translated into **binary**. To do this, *ASCII code was developed (in 1963).* ASCII has become **worldwide standard**. It contains all the letters in the alphabet plus some additional characters. ASCII characters are always represented by the same number. For example, the ASCII code for the capital letter "A" is always represented by the order number 65, which can easily be represented using 0s and 1s in binary: *65 expressed as a binary number is 1000001.*

Here are some Examples:

Decimal			Binary	Character
057	071	039	00111001	9
058	072	03A	00111010	:
059	073	03B	00111011	;
060	074	03C	00111100	<
061	075	03D	00111101	=
062	076	03E	00111110	>
063	077	03F	00111111	?
064	100	040	01000000	@
065	101	041	01000001	A
066	102	042	01000010	B
067	103	043	01000011	C
068	104	044	01000100	D

Capital A has an ASCII value of 65. Important if we want to represent this character in binary