Introduction to Data Structures

- In simple words, Data Structure is just a way to store and organize data so that it can be used efficiently.
- DS deals with storage and Access of data in different way depending on purpose and application
- Some of the common Data Structures are Array, Structure, Linked List, Stack, Queue, priority queues, Tree, Graph, etc.

Data Type and Data Structure

Data Type

- A *data type* is the most basic and the most common classification of data.
- It helps compiler to know the form or the type of information that will be used throughout the code.
- It is a type of information transmitted between the programmer and the compiler where the programmer informs the compiler about what type of data is to be stored and also tells how much space it requires in the memory.
- Some basic examples are int, char, float etc.
- It is the type of any variable used in the code.

```
#include <iostream.h>
using namespace std;

void main()
{
   int a;
   a = 5;

   float b;
   b = 5.0;

   char c;
   c = 'A';

   char d[10];
```

```
    d = "example";
}
```

In Above example various Data types and identifiers:

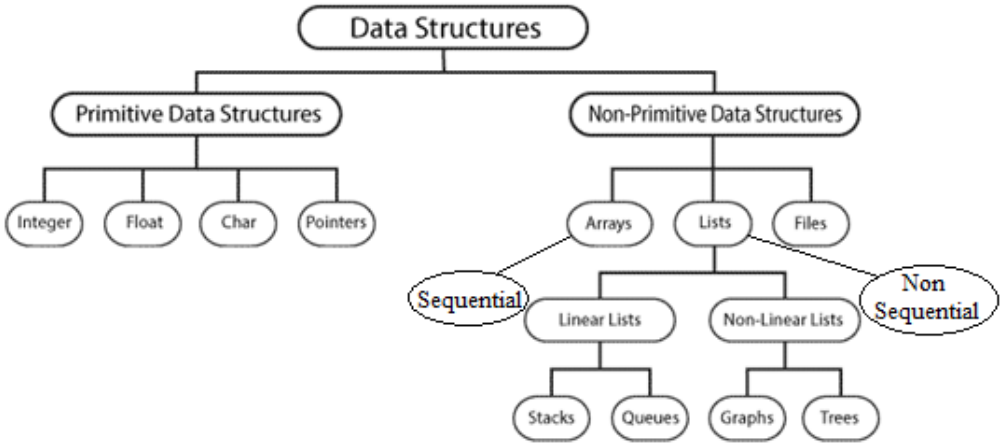
- Variable ‘a’ is of data type integer which is denoted by int a and it will be used as an integer type variable throughout the process of the code
- Variables ‘b’, ‘c’ and ‘d’ are of type float, character and string respectively

Data Structure

- A data structure is a collection of different forms and different types of data that has a set of specific operations that can be performed. It is a collection of data types. It is a way of organizing the items in terms of memory, and also the way of accessing each item through some defined logic.
- Some examples of data structures are stacks, queues, linked lists, binary tree and many more.
- Data structures perform some special operations only like insertion, deletion and traversal.
    - For example, you have to store data for many employees where each employee has his name, employee id and a mobile number
    - So this kind of data requires complex data management, which means it requires data structure comprised of multiple primitive data types. So data structures are one of the most important aspects when implementing coding concepts in real-world applications

DATA TYPES	DATA STRUCTURES
Data Type is the kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of the given data type only	Data Structure is the collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program.
Implementation through Data Types is a form of abstract implementation	Implementation through Data Structures is called concrete implementation
Can hold values and not data, so it is data less	Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables	The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity	Time complexity comes into play when working with data structures
Examples: int, float, double	Examples: stacks, queues, tree

Types of Data Structures



Primitive Data Structures

- Basic data structures that directly operate upon the machine instructions
- They have different representations on different computers.
- E.g. Integers, Floating point numbers, Character constants, String constants and Pointers

Non-primitive Data Structures

- more complicated data structures
- derived from primitive data structures.
- Group of same or different data items with relationship between each data item.
- E.g. Arrays, Lists and File

Linear Data Structures

- Data are accessed in a sequential manner
    - however, the elements can be stored in these data structures in any order.
- Eg. Array, Linked List, Stack, Queue, Hashing, etc.

Non-Linear Data Structure

- Elements are not stored in linear manner
- The data elements have hierarchical relationship which involves the relationship between the child, parent, and grandparent

- E.g. Trees, Binary Search Trees, Graphs, Heaps, Trie, Segment Tree etc.

Sequential Data Structure

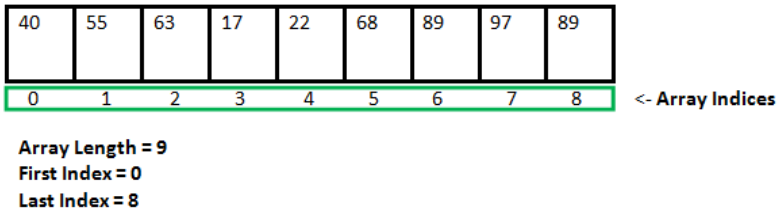
- Storage of data is contiguous
- E.g. Array

Non Sequential Data Structure

- Storage of data is Non contiguous
- E.g. Linked List

Static Data Structure

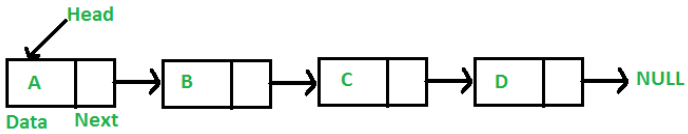
- In Static data structure the size of the structure is fixed.
- The content of the data structure can be modified but without changing the memory space allocated to it.



- E.g. Array

Dynamic Data structure

- the size of the structure in not fixed
- memory allocation and size can be modified during the operations performed on it
- designed to facilitate change of data structures in the run time.



- Example of Dynamic Data Structures: Linked List
    - **Static Data Structure vs Dynamic Data Structure**

      Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time which may be considered

efficient with respect to memory complexity of the code. Static Data Structure provides more easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

- Use of Dynamic Data Structure in Competitive Programming
  In competitive programming the constraints on memory limit is not much high and we cannot exceed the memory limit. Given higher value of the constraints we cannot allocate a static data structure of that size so Dynamic Data Structures can be useful.

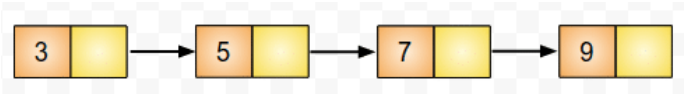
Example of DS

Array:

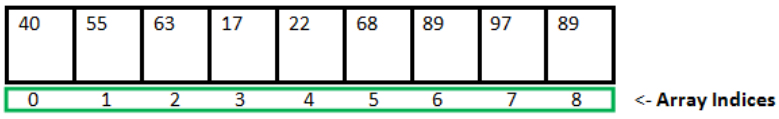
- collection of similar type of data.
- all the data present in the array will be of a single data type.
- stored in a contiguous memory location

Link list:

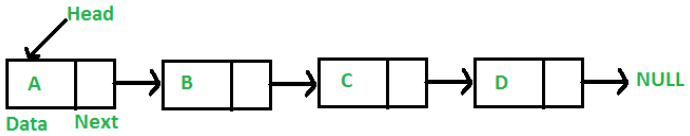
- linear data structure with elements linked using pointers
  - unlike arrays, linked list elements are not stored at a contiguous memory location,
- gives benefit over the array, in array searching operation requires O(1) time complexity but insertion and deletion take O(n) time complexity which is a bit costly but these operations in Linked lists are fast.



- used in a real-world data structure like Graphs and Binary Search tree.



Array Length = 9
First Index = 0
Last Index = 8



Link list vs Array

1. An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.
2. In the array the elements belong to indexes, i.e., if you want to get into the fourth element you have to write the variable name with its index or location within the square bracket.
3. In a linked list though, you have to start from the head and work your way through until you get to the fourth element.
4. Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.
5. Operations like insertion and deletion in arrays consume a lot of time. On the other hand, the performance of these operations in Linked lists is fast.
6. Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.
7. In an array, memory is assigned during compile time while in a Linked list it is allocated during execution or runtime.
8. Elements are stored consecutively in arrays whereas it is stored randomly in Linked lists.
9. The requirement of memory is less due to actual data being stored within the index in the array.
10. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements.
11. In addition memory utilization is inefficient in the array. Conversely, memory utilization is efficient in the linked list.

Some important points in favor of Linked Lists.

1. The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, the upper limit is rarely reached.
2. Inserting a new element in an array of elements is expensive because a room has to be created for the new elements and to create room existing elements have to be shifted.
3. For example, suppose we maintain a sorted list of IDs in an array id[].

   id[] = [1000, 1010, 1050, 2000, 2040, …..].

   And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

4. Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Linked list provides the following two advantages over arrays

- Dynamic size
- Ease of insertion/deletion

Linked lists have following drawbacks:

- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.
- Extra memory space for a pointer is required with each element of the list.
- Arrays have better cache locality that can make a pretty big difference in performance.

The data structures can also be classified on the basis of the following characteristics:

Characterstic	Description
Primitive Data Structure	includes basic data types like integer, character, float, pointers. It is also called **built-in data structure** as these are the basic data types in which data are stored.
Non- Primitive Data structure	also called **user-defined data structure** includes Arrays, Linklist, stack, Trees etc. Basically we define non-primitive data structures according to our usage to store and manipulate data.
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**

Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

Primitive Data Structure

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 4,294,967,295
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

```
#include <iostream>
using namespace std;

int main() {
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

   return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

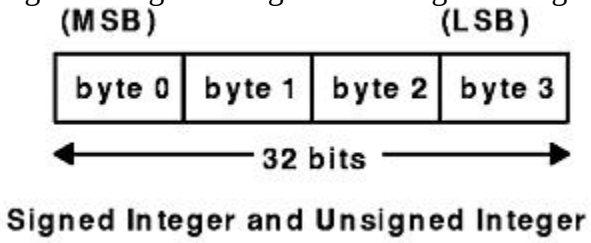
When the above code is compiled and executed, it produces the following result which can vary from machine to machine −

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

Integer

Size 4 Byte

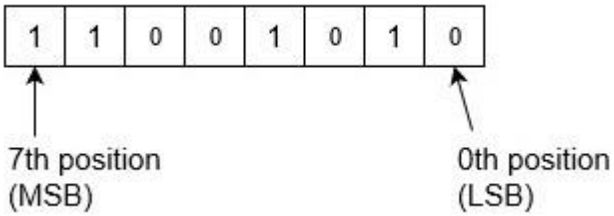
Figure 1. Signed Integer and Unsigned Integer



Signed Integer and Unsigned Integer

For better understanding let us assume 1 byte value

Example-1 − Convert binary number 11001010 into decimal number. Since there is no binary point here and no fractional part. So,



Binary to decimal is,

$= (11001010)_2$
$= 1x2^7+1x2^6+0x2^5+0x2^4+1x2^3+0x2^2+1x2^1+0x2^0$
$= 128+64+0+0+8+0+2+0$
$= (202)_{10}$

Signed and Unsigned Int

- In Signed Int left most bit (MSB) is used to denote the sign (negative or Positive)
- The rest of the bits are then used to denote the value normally.
- MSB is used to denote whether it's positive (with a 0) or negative (with a 1).
- If you want to get technical, a sign bit of 0 denotes that the number is a *non-negative*, which means it can equal to the decimal zero or a positive number.
- The negative number is stored in 2's Complement notation

(-/+)	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	0	1	1	0	1

$$01001101_2 = +77_{10}$$

- if we have an 8-bit signed integer, the first bit tells us whether it's a negative or not, and the other seven bits will tell us what the actual number is. Because of this, we're technically working with a more limited range of numbers that can be represented; 7 bits can't store numbers as big as 8 bits could.

More insight with 4 bit storage

Unsigned Binary Numbers

Example 1a:

Unsigned $0101_2 = 5_{10}$

2^3	2^2	2^1	2^0
0	**1**	0	**1**

- **Example 2a:**
  Unsigned $1001_2 = 9_{10}$

2^3	2^2	2^1	2^0
1	0	0	**1**

Signed binary numbers

Example 1b:

- ***Signed*** $0101_2 = +5_{10}$

(-/+)	2^2	2^1	2^0
0	**1**	0	**1**

- When a signed binary number is positive or negative it's 'marked' with a 0 or 1 respectively at the first far-left bit, the sign bit. The number above doesn't change at all. It's just more explicitly a positive number.

Example 2b:

Signed $1001_2 = -7_{10}$

(-/+)	2^2	2^1	2^0
1	**0**	**0**	1

- But the above binary number completely changes. And we're now representing a negative!
- Negative binary integers
- Read it as 2's complement notation (2's complement of 001 = 111)
- Signed $1001_2 = -7_{10}$

Note : you calculate a negative binary integer's value starting at 1, not 0. Because the decimal zero is not included in a negatively signed bit integer, we don't start counting at zero as we would when it's a positively signed bit integer.

To explain that quirk let's compare positively and negatively signed integers. Working with a 4-bit integer, if we had four bits with a value of zero, the number would equal to 0. That's the lowest value we can have. Because a *non-negative* signed bit means we can have a positive integer, or a 0.

$0000 = 0_{10}$

A 4-bit negative integer of four bits of one values (the ones now being the "off switch"), the number would not equal 0, but -1. Which makes sense, since that's the highest decimal number we can represent while still having a negative.

1 1 1 1 = -1_{10}

But that means, when we're adding up our values to get our final decimal number, we start our counting from 1, not from 0.

Here's a visual comparison of the decimal and binary equivalents that show how a 0 signed bit integer is the decimal 0_{10} or larger, while a 1 signed bit integer is decimal -1_{10} or smaller.

Two's complement binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

For 8 bit signed number the range will be

$-2^{7} – 0 – (2^{7}-1)$

For 8 bit unsigned number the range will be

$0 - (2^{8}-1)$

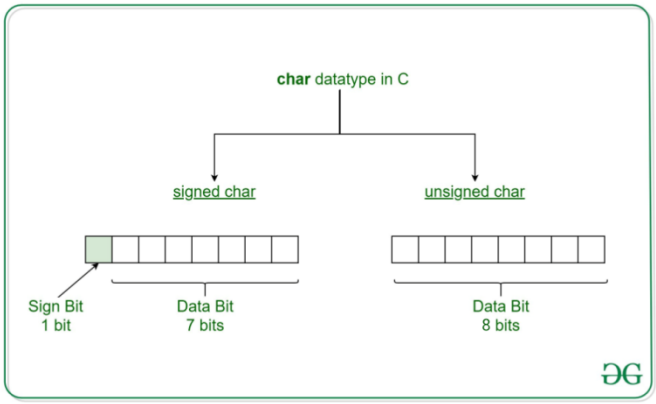
Same way for 32bit (4 byte integer)

Signed : $-2^{31} – 0 – (2^{31}-1)$

Unsigned: $0 – (2^{32}-1)$

Character

- **char** is the most basic data type in C.
- It stores a single character and requires a **single byte of memory** in almost all compilers.
- Now character datatype can be divided into 2 types:
    - signed char
    - unsigned char



- **unsigned char** is a character datatype where the variable consumes all the 8 bits of the memory and there is no sign bit (which is there in signed char).
- So it means that the range of unsigned char data type ranges from 0 to 255.

Syntax:

```
unsigned char [variable_name] = [value]
```

Example:

```
unsigned char ch = 'a';
```

- **Initializing an unsigned char:** Here we try to insert a char in the unsigned char variable with the help of ASCII value. So the ASCII value 97 will be converted to a character value, i.e. ‘a’ and it will be inserted in unsigned char.

filter_none
edit
play_arrow
brightness_4

```
// C program to show unsigned char

#include <stdio.h>

int main()
{

    int chr = 97;
    unsigned char i = chr;
    printf("unsigned char: %c\n", i);
```

```
    return 0;
}
```

Output:

```
unsigned char: a
```

Initializing an unsigned char with signed value: Here we try to insert a char in the unsigned char variable with the help of ASCII value. So the ASCII value -1 will be first converted to a range 0-255 by rounding. So it will be 255. Now, this value will be converted to a character value, i.e. ‘ÿ’ and it will be inserted in unsigned char.

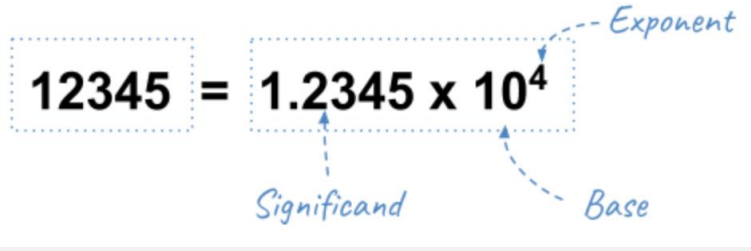
filter_none
edit
play_arrow
brightness_4

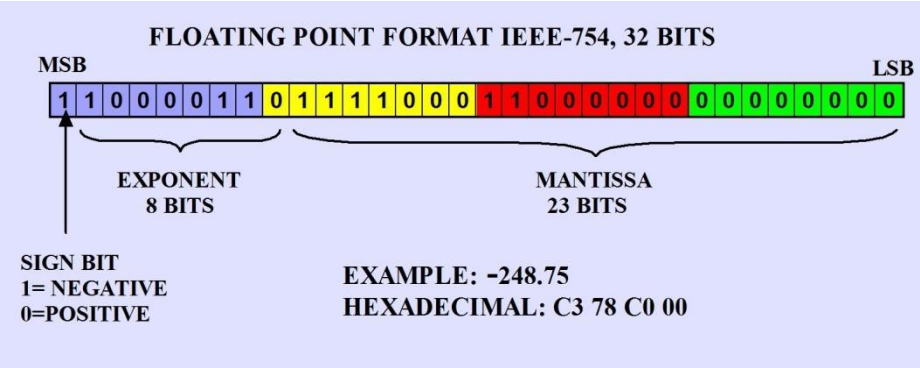
```
// C program to show unsigned char

#include <stdio.h>

int main()
{

    int chr = -1;
    unsigned char i = chr;
    printf("unsigned char: %c\n", i);

    return 0;
}
```

Output:

```
unsigned char: ÿ
```

How floating-point no is stored in memory?





A typical single-precision 32-bit floating-point memory layout has the following fields :

1. sign
2. exponent
3. significand(mantissa)

Sign

- The high-order bit indicates a sign.
- `0` indicates a positive value, `1` indicates negative.

Exponent

- The next 8 bits are used for the exponent which can be positive or negative
- In floating number, no concept called 2's complement to store negative numbers. To overcame that, they came up with bias concept where we add some positive value to negative exponent and make it positive.
- In general, whether it negative or positive they add bias value to exponent value to reduce implementation complexity.
  - `Formula to calculate bias value is`

    $$bias_n = 2^{n-1} - 1;$$

  - Here, we have allocated 8 bits for exponent.
  - So n will be 8

- $2^{7-1} = 127$
- Normalized exponent = Actual exponent + bias value
  - $= 3 + 127 = 130 = (10000010)_2$

- Thus by adding (Add Bias 127) exponent will be encoded such that
  - `1000 0000` represents `0`
  - `0000 0000` represents `-128`
  - `1111 1111` represents `127`
  -

Significant (Mantissa)

- The remaining 23-bits used for the significant (mantissa).
- Each bit represents a negative power of 2 counting from the left, so:

```
01101 = 0 * 2^-1 + 1 * 2^-2 + 1 * 2^-3 +
0 * 2^-4 + 1 * 2^-5
      = 0.25 + 0.125 + 0.03125
      = 0.40625
```

Practical Example

Let us consider very famous float value `3.14`(PI)

- **Sign**: Zero here, as PI is positive!
- **Exponent calculation**
  - `3` : `0011` in binary
  - `0.14`

```
0.14 x 2 = 0.28, 0
0.28 x 2 = 0.56, 00
0.56 x 2 = 1.12, 001
0.12 x 2 = 0.24, 0010
0.24 x 2 = 0.48, 00100
0.48 x 2 = 0.96, 001000
0.96 x 2 = 1.92, 0010001
0.92 x 2 = 1.84, 00100011
0.84 x 2 = 1.68, 001000111
And so on . . .
```

  - `0.14 = 001000111...`
  - 3.14 = `(3.14 * 2^0) = 11.001000111...`

- Shift it (normalize it) and adjust the exponent accordingly
    - 3.14 = `(3.14 * 2^0) = 11.001000111... = 1.1001000111... * 2^1 (1.57 * 2^1)`
- Add the bias of `127` to the exponent `1` and store it(i.e. `128` = `1000 0000`) `0 1000 0000 1100 1000 111...`
- Forget the top `1` of the mantissa (which is always supposed to be `1`, except for some special values, so it is not stored), and you get: `0 1000 0000 1001 0001 111...`
- So our value of `3.14` would be represented as something like:

```
0 10000000 10010001111010111000011
    ^      ^                ^
    |      |                |
    |      |                +--- significand = 0.7853975
    |      |
    |      +------------------- exponent = 1
    |
    +------------------------- sign = 0 (positive)
```

- The number of bits in the exponent determines the range (the minimum and maximum values you can represent)

Summing up significand

- If you add up all the bits in the significand, they don't total `0.7853975`(which should be, according to 7 digit precision). They come out to `0.78539747`.
- There aren't quite enough bits to store the value exactly. we can only store an approximation.
- The number of bits in the significand determines the precision.
- 23-bits gives us roughly 6 decimal digits of precision. 64-bit floating-point types give roughly 12 to 15 digits of precision.

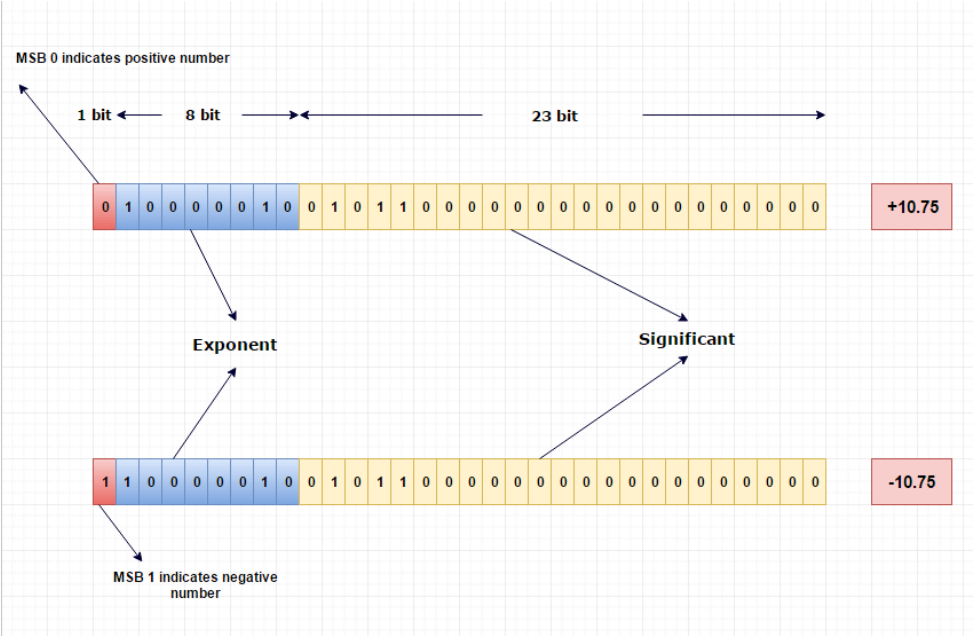
Strange! But fact

- Some values cannot represent exactly no matter how many bits you use. Just as values like 1/3 cannot represent in a finite number of decimal digits, values like 1/10 cannot represent in a finite number of bits.

- Since values are approximate, calculations with them are also approximate, and rounding errors accumulate.

More Example:

Pictorial Explanation



Let's see things working

```
#include <stdio.h>
#include <string.h>/* Print binary stored in plain 32 bit block */
void intToBinary(unsigned int n)
{
        int c, k;
        for (c = 31; c >= 0; c--)
        {
                k = n >> c;
                if (k & 1)  printf("1");
                else        printf("0");
        }
        printf("\n");
}int main(void)
{
        unsigned int m;
        float f = 3.14;         /* See hex representation */
        printf("f = %a\n", f);
        /* Copy memory representation of float to plain 32 bit block */
        memcpy(&m, &f, sizeof (m));
        intToBinary(m);         return 0;
}
```

- This C code will print binary representation of float on the console.

```
f = 0x3.23d70cp+0
01000000010010001111010111000011
```

Where the decimal point is stored?

- The decimal point not explicitly stored anywhere.

[Click here to read more . . . !]