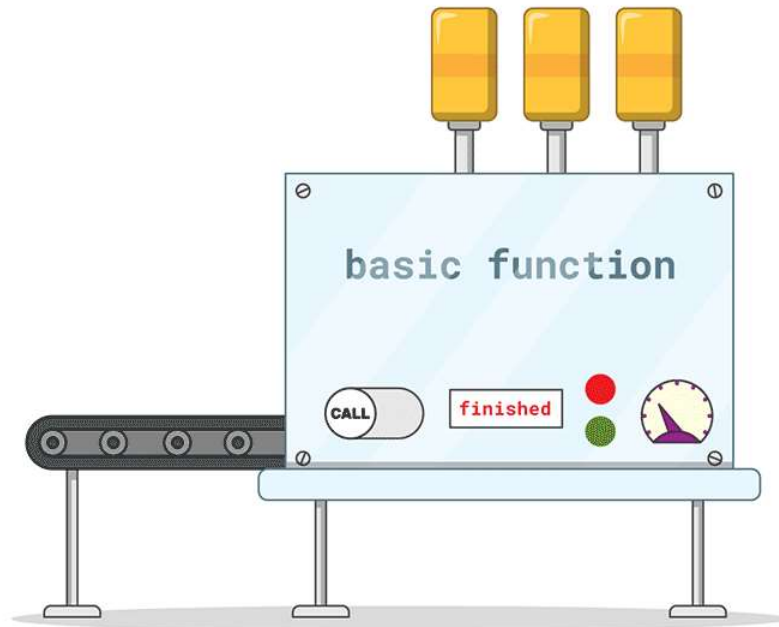


Functions and Arrays

Functions

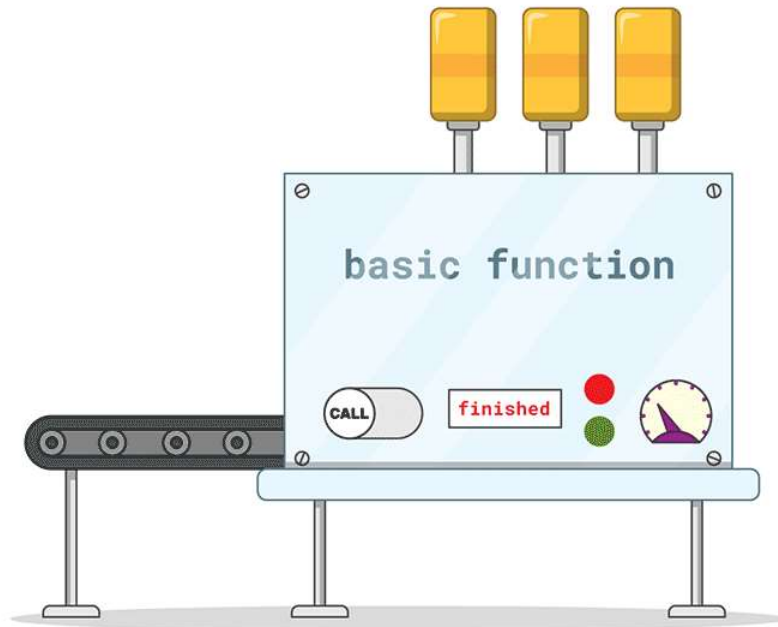
Remember `scanf` and `printf` discussed in the input and output section of this blog? We gave a hint that these were functions and briefly discussed what functions are. Let's elaborate on that in this section. So, we know that when we write `printf("zine")`, the computer gets a command to print "zine" on the screen. Behind the scenes, a separate code was written while designing the C language according to which the special task of printing is assigned whenever we write or call a print function in our code i.e. to display something on the screen. "Calling" a function is the technical term to use a function.



Similar to `printf()` we can also define our own functions that do some specific tasks and can be used simply by calling them as we call `print`. This is an integral part of programming. Let us start with a definition.

A function is a block of statements that performs a specific task. Let's say you are writing a program and you need to perform the same task in that program more than once. In such a case you have two options:

a) Use the same set of statements every time you want to perform the task. A lot of `Ctrl + C - Ctrl + V` !



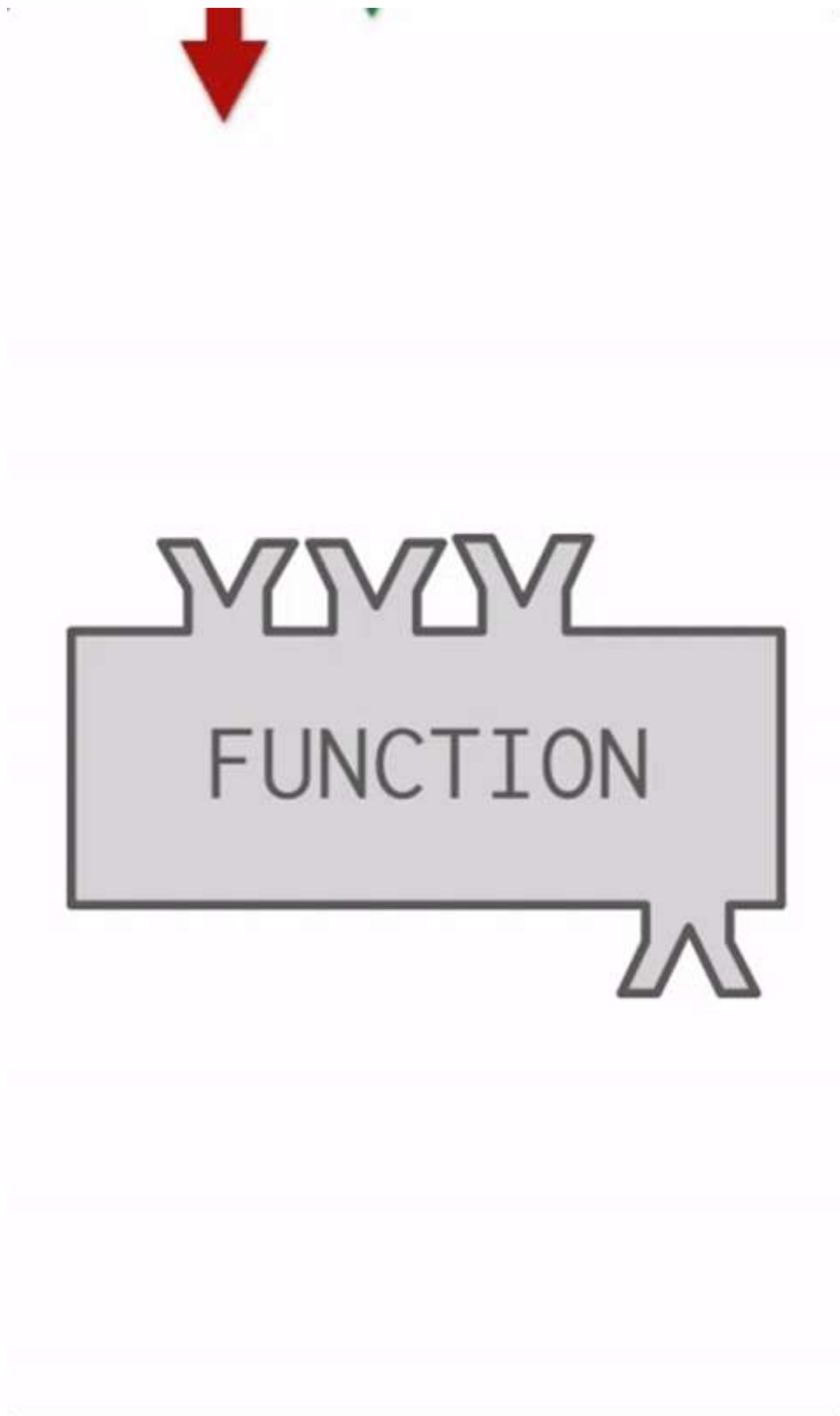
b) Or, create a function to perform that task and just call it every time you need to perform that task.

Using option (b) is easier and concise, and a good programmer always uses functions while writing code. In other words,

Functions are used because of the following reasons –

- a) To improve the readability of code.
- b) Improves the reusability of the code, the same function can be used in any program rather than writing the same code from scratch.

Now let us try to understand how they actually work. In essence, functions accept some input (called arguments), do some processing on this input, through the block of statements present, and give an output.



Let's see an example of a simple function that takes two numbers and prints their sum.

```
void sum(int a,int b)
{
    printf("%d",a+b);
}
```

We know a,b are the arguments to the function. But what is this "void"? This denotes the return type of the function. We can return values in a function. Essentially, it means that if we try to assign the function value to a variable as

```
variable = sum(a,b);
```

This is the type of variable required. "Void" in this case denotes no value is returned and hence the function cannot be assigned to a variable. The return type can be int, char, float, or any other data type.

So when we call or sum function as

A screenshot of a console window. On the left, there is a dark grey box containing the code `sum(10,15);` in white text. To the right, the console output is shown on a black background. It displays the number '25' in white, followed by a green message: `...Program finished with exit code 0` and `Press ENTER to exit console.` with a small white cursor icon at the end.

We totally understand that the concepts of functions can be confusing sometimes, so we suggest you visit this video.(complete the link)

Try to find out the output of the following codes.

```
1  #include<stdio.h>
2
3  int  calc()
4  {int a=5, b=6, c=7;
5   int d = a%5 + b-- + b/2 + ++c;
6   return d;
7
8  }
9
10 int main()
11 {
12     int number = calc();
13     printf("%d", number);
14     return 0;
15 }
```

A function `calc()` is made outside the main function. The value returned from the function `calc()` is printed again in the main.

Try doing a similar problem using the concept of arguments.

```
#include<stdio.h>

int calc(int a, int b , int c)
{
    int d = a%5 + b-- + b/2 + ++c;
    return d;
}

int main()
{
    int number1 = calc(5,8,6);
    printf("%d", number1);
    int number2 = calc(10,20,30);
    printf("%d", number2);
    int number3 = calc(4,3,8);
    printf("%d", number3);

    return 0;
}
```

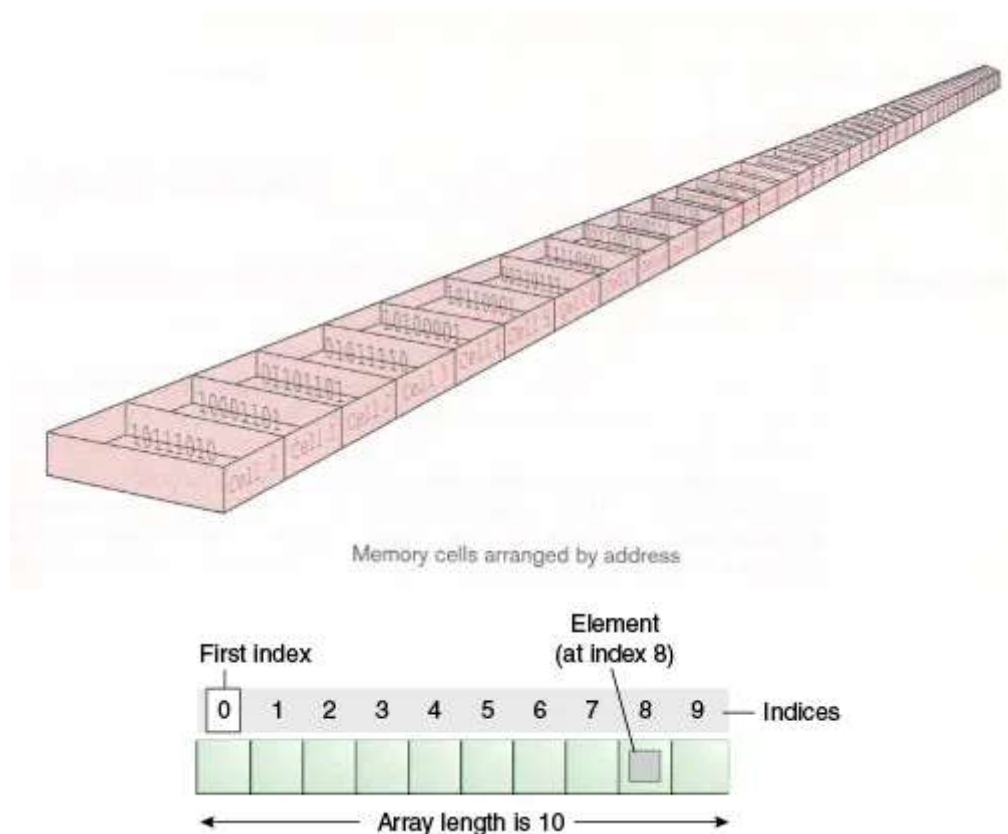
Try sketching the flow(not flowchart) of this program showing the function calls.

Concept of Arrays

We have already learned about variables and how we use them to store various data during the course of programming. Now think of a problem where we need to store many variables(of same data type) for ex:- adding 10 natural numbers. Here we need to store them in a new data type called an array.

Taking the reference from the variables part of the blog, recall that different variables are randomly distributed in the RAM. Arrays have special features where all the variables of the same array are stored

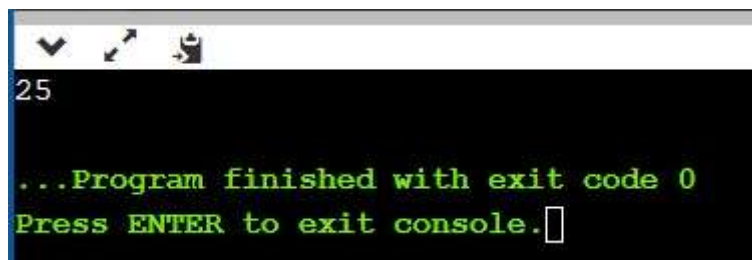
in a sequence as shown in the figure below. We will discuss further the use and importance of this property.



An array is a contiguous memory location that contains a group of elements, such as an integer or string. Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The first box corresponds to the first element and the last block to the last element.

A screenshot of a console window. The window has a title bar with standard OS icons. The main area is black with green text. It displays the number '25' on the first line. The second line contains the text '...Program finished with exit code 0' followed by 'Press ENTER to exit console.' and a cursor icon.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [array size];
```

This is called a single-dimensional array. The array size must be an integer constant greater than zero and the type can be any valid C data type. For example, to declare a 10-element array called the balance of type double, use this statement –

```
int balance [10];
```

To explore more about arrays click on the topics below:-

Add links here.

Can we find out the address for a certain specified block in a 2-D array if we are given the address of its first data(first-row first column) and its data type? Think of a generalized formula.

Can we find out the address for a certain specified block in a 2-D array if we are given the address of its first data(first-row first column) and its data type? Think of a generalized formula.


```
9  #include <stdio.h>
10
11 int main()
12 {
13     int a=1,b=2,c=3,d=4,e=5,f=6,g=7,h=8,i=9,j=10;
14     printf("%d ",a);
15     printf("%d ",b);
16     printf("%d ",c);
17     printf("%d ",d);
18     printf("%d ",e);
19     printf("%d ",f);
20     printf("%d ",g);
21     printf("%d ",h);
22     printf("%d ",i);
23     printf("%d ",j);
}
```

input

1 2 3 4 5 6 7 8 9 10

...Program finished with exit code 0
Press ENTER to exit console.

Above is a simple example of printing the first 10 natural numbers. Now it's your task to print the first 1000 natural numbers. Can you? Of course, you can! But will you do it with the same method? Oh common guys TIME IS MONEY!!

Arrays save us from the tedious task of declaring too many variables and using multiple statements. Wanna know how?

```
9  #include <stdio.h>
10
11 int main()
12 {
13     int a[10]={1,2,3,4,5,6,7,8,9,10};
14     for(int i=0;i<10;i++)
15     {
16         printf("%d ",a[i]);
17     }
}
```

1 2 3 4 5 6 7 8 9 10

...Program finished with exit code 0
Press ENTER to exit console.

See how we can solve the same problem in 3-4 lines which took 10-12 lines without arrays. In arrays, we just need to define the array variable name. Its value is directly accessed by its index. Using loops with arrays can do wonders! We can store thousands of elements in arrays and access them with just one simple loop directly by its index.

Limitation of the array over variables

All the elements in an array should have the same data type whereas when we define a number of variables, we can have a variety of elements having different data types. Let us see this by an example:

`int a[10];` Here we can store 10 different variables in `a` but the constraint being they all must be of the same data type i.e. `int`.

`int a; float b; char c;`

Here we have reserved three blocks of memory but are free of the constraint we had in our array example. We can store 3 variables of different data types.

We have learned the basic and necessary tools used to develop and think of an algorithm to get a solution. Now let's implement that knowledge.

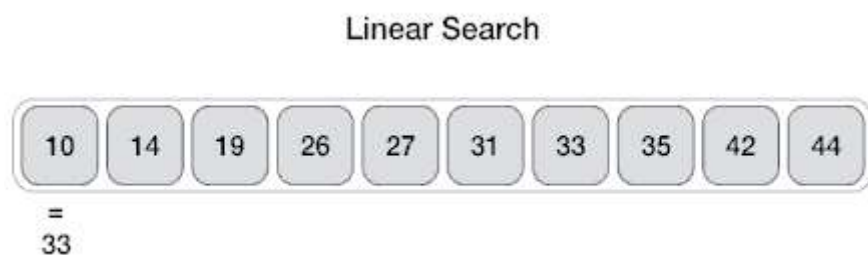
How are contacts in our mobile phones sorted according to names? How come we just type the names in the search bar and it pops up from the long list? All these tasks are accomplished using the most widely used algorithms - Searching and Sorting algorithms. These algorithms are quite interesting and can handle simple tasks of sorting and searching files in your pc to complex computing problems. In this era of the digital economy, data is the new mineral oil. And anything that helps in managing data is highly valuable, so are searching and sorting algorithms. Let's learn some of the basic searching and sorting algorithms:

Searching

Suppose we are given a data set of integers in an array and we need to find out whether a certain integer is present in it or not. One simple way would be to compare each and every data with our integer that you all must have thought till now. Let us now look at various algorithms and your task would be to find out which one is the most effective and time-efficient.

We will start with the simplest one which is a linear search.

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm:

1. Compare the first element with the data.
2. If the first element is equal to the data i.e. $A[0]$, type data found and exit.
3. If the first element is not equal to the data, go to the next element i.e. $A[1]$.
4. Repeat the process until data does not match with any of the elements.
5. If we reach the end element during this process and $A[\text{last element}]$ is not equal to data, then print "data not found".
6. Exit. There is a more time-efficient algorithm for searching known as binary search.

Sorting

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

For example, The below list of characters is sorted in increasing order of their ASCII values. That is, the character with lesser ASCII value will be placed first than the character with higher ASCII value.

The diagram shows the string "geeksforgeeks" on the left, labeled "Input" in red. An arrow points to the string "eeeefggkorrss" on the right, labeled "Output" in red. The characters are color-coded: 'e' is light green, 'f' is medium green, 'g' is dark green, 'k' is black, 'o' is light grey, 'r' is medium grey, and 's' is dark grey.

There are many sorting algorithms. Some commonly used have been mentioned here:

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. Let us look at it with the help of an example. Let the array to be sorted is



Bubble sort starts with the very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this-



Next, we compare 33 and 35. We find that both are already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller than 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

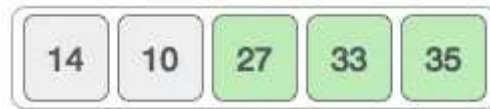


To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like

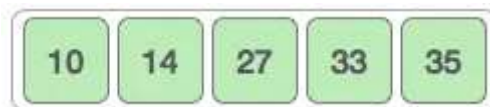
this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.



Another example for better understanding is given below, try to visualize it without explanation.

Eg: We have a simple array [5, 3, 8, 2, 1, 4]

We have some more algorithms for sorting such as selection sort and insertion sort.(complete the links)