Flowcharts

INTRODUCTION

Here are the steps that may be followed to solve an algorithmic problem:

- Analysing the problem statement means making the objective of the program clear in our minds like what the input is and what is the required output.
- Sometimes the problems are of complex nature, to make them easier to understand, we can break-down the problem into smaller sub-parts.
- In order to save our time in debugging our code, we should first-of-all write down the solution on a paper with basic steps that would help us get a clear intuition of what we are going to do with the problem statement.
- In order to make the solution error-free, the next step is to verify the solution by checking it with a bunch of test cases.
- Now, we clearly know what we are going to do in the code. In this step we will start coding our solution on the compiler.

Basically, in order to structure our solution, we use flowcharts. A flowchart would be a diagrammatic representation of our algorithm - a step-by-step approach to solve our problem.

Flowcharts

Uses of Flowcharts

➔ Used in documentation.

➔ Used to communicate one's solution with others, basically used for group projects.

➔ To check out at any step what we are going to do and get a clear explanation of the flow of statements.

Flowchart components

- **Terminators**

Mainly used to denote the start point of the program.

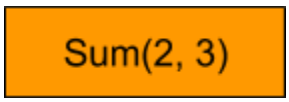
Used to denote the end point of the program.

- **Input/Output**

Used for taking input from the user and store it in variable ‘var’.

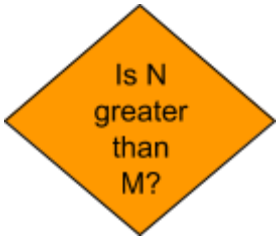
Used to output value stored in variable ‘var’.

- **Process**



Used to perform the operation(s) in the program. For example: Sum(2, 3) just performs arithmetic summation of the numbers 2 and 3.

- **Decision**



Used to make decision(s) in the program means it depends on some condition and answers in the form of TRUE(for yes) and FALSE(for no).

- **Arrows**

Generally, used to show the flow of the program from one step to another. The head of the arrow shows the next step and the tail shows the previous one.

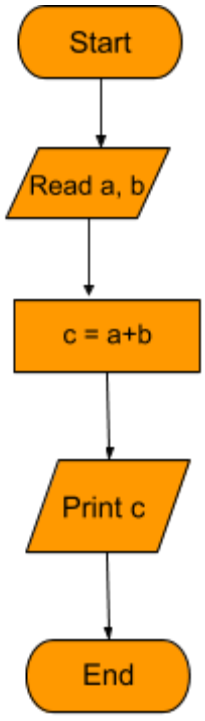
- **Connector**

Used to connect different parts of the program and are used in case of break-through. Generally, used for functions(which we will study in our further sections).

Example 1:

Suppose we have to make a flowchart for adding 2 numbers a and b.

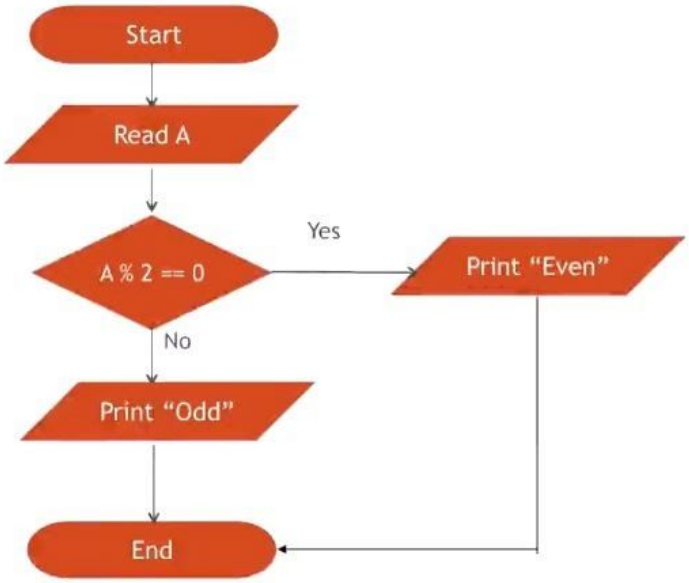
Solution:



Example 2:

Suppose we have to check if a number is even or odd.

Solution:



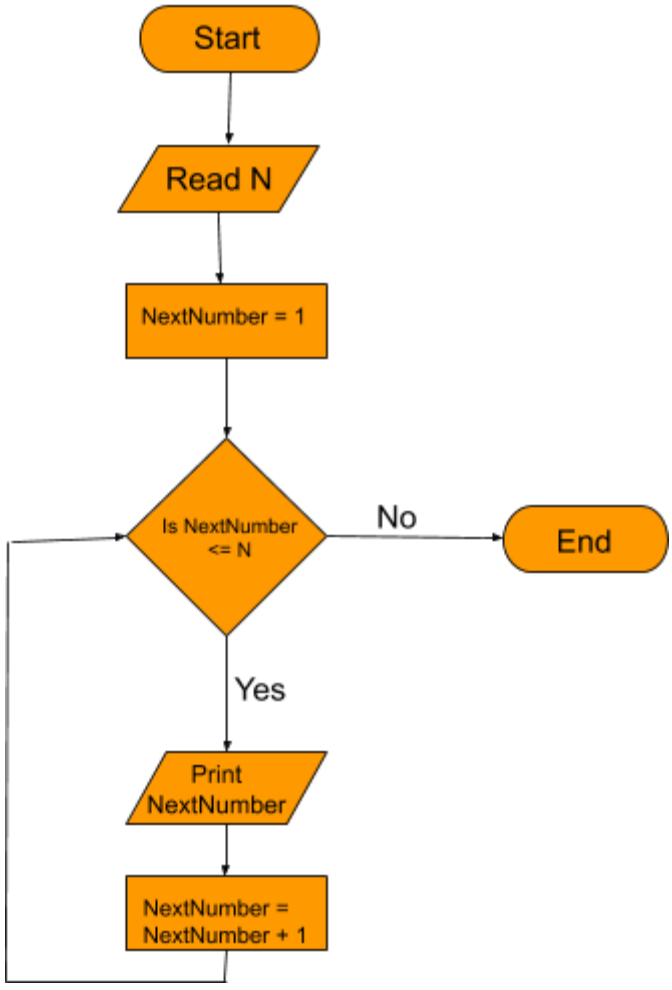
Note: Operator % is used to find the remainder of a number with respect to another number. For example:

- 4 % 3 = 1
- 10 % 5 = 0
- 4 % 2 = 0
- 4 % 4 = 0
- 0 % 1 = 0
- 1 % 0 = undefined (as it leads to division by 0)

Example 3:

To print the numbers less than or equal to n where n is given by the user.

Solution:



Summary

- Flowcharts are the building-block of any program written in any language.
- Different shapes used to have different meanings.
- Every problem can be represented in the form of a flow chart.
- Sometimes, it becomes a bulky process to represent any program using flowchart. In those cases, try to find out the optimal solution to the given problem.

Getting Started

For this course, you will be provided with the in-built compiler of Coding Ninjas. However, if you want to run programs and practice them on your local desktop, there are various compilers out there like Code blocks, VS Code, Dev C++, Atom and many more. We have provided the steps for installing Code Blocks in a separate file.

About Code blocks

Code blocks is an Integrated Development Environment (IDE) for C/C++. To set the path of compiler, follow the given steps:

1. Click on menu **Settings** -> **Compiler.**
2. Then click on the tab **Toolchain Executables.**
3. In the text box under **Compiler's installation directly,** click on the button **Auto Detect**. A pop-up appears.
4. In case, pop-up says, **Couldn't auto detect...,** that means you have downloaded the incorrect setup of code blocks. Uninstall this setup, and install the setup with **MinGW**. Then repeat the above steps.

To create a new file:

1. Follow **File -> New -> Empty File.**
2. Then save that file with extension **.cpp**
3. In order to run and compile the program press F11 or click on the button right next to a play button which says **Build and Run**.

You will get your output window after following step 3.

Looking at the code

C++ code begins with the inclusion of header files. There are many header files available in the C++ programming language which you will discuss while moving ahead with the course.

So, what are these header files?

The names of program elements such as variables, functions, classes, and so on must be declared before they can be used. For example, you can't just write x = 42 without first declaring variable 'x' as:

```
int x = 42;
```

The declaration tells the compiler whether the element is an int, a double, a float, a function or a class. Similarly, header files allow us to put declarations in one location and then import them wherever we need them. This can save a lot of typing in multi-file programs. To declare a header file, we use **#include** directive in every .cpp file. This #include is used to ensure that they are not inserted multiple times into a single .cpp file.

Note: # operator is known as **Macros.**

The following are not allowed, or are considered as bad practices while putting header files into the code:

- Built-in-type definitions at namespace or global scope
- non-inline function definitions
- Non-const variable definitions
- Aggregate definitions
- Unnamed namespaces
- Using directives

Now, moving forward to the code:

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

iostream stands for Input/Output stream, meaning this header file is necessary if you want to take input through the user or print output to the screen. This header file contains the definitions for the functions:

- **cin** : used to take input
- **cout** : used to print output

namespace defines which input/output form is to be used. You will understand these better as you progress in the course.

Note: semicolon (;) is used for terminating a C++ statement. ie. different statements in a C++ program are separated by semicolon

main() function:

Look at the following piece of code:

```
int main() {

    Statement 1;
    Statement 2;
    ...
}
```

You can see the highlighted portion above. Let's discuss each portion stepwise.

Starting with the line:

```
int main()
```

- **int** : This is the return-type of the function. You will get this thing clear once you reach the **Functions** topic.

- **main()** : This is the portion of any C++ code inside which all the commands are written and gets executed.
    - This is the line at which the program will begin executing. This statement is similar to the start block of flowcharts.
    - As you will move further in the course, you will get a clear glimpse of what this function is. Till then, just note that you will have to write all the programs inside this block.
- **{}** : all the code written inside the curly braces is said to be in one block, also known as scope of a particular function. Again, these things will be clear when you will study functions.

For now, just understand that this is the format in which we are gonna write our basic C++ code. From time to time as you will move forward with the course, you will get a clear and better understanding.

Declaring a variable:

To declare a variable, we should always know what type of value it should hold, whether it's an integer (int), decimal number (float, double), character value (char). In general, the variable is declared as follows:

Datatype variableName = VALUE;

Note: Datatype is the type of variable:

- int : Integer value
- float, double : Decimal number
- char : Character values (including special characters)
- bool : Boolean values (true or false)
- long : Contains integer values but with larger size
- short : Contains integer values but with smaller size

Table for datatype and its size in C++: (This can vary from compiler to compiler and system to system depending on the version you are using)

Datatype	Default size
bool	1 byte
char	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes

For example: To declare an integer variable ‘a’ with a value of 5, the structure looks like:

```
int a = 5;
```

Similarly, this way other types of variables can also be declared. There is one more type of variable known as **string** variables which store combinations of characters. You will study that in your further lectures.

Rules for variable names:

- Can’t begin with a number.
- Spaces and special characters except underscore(_) are not allowed.
- C++ keywords (reserved words) must not be used as a variable name.
- C++ is case-sensitive, meaning a variable with name ‘A’ is different from variable with name ‘a’. (Difference in the upper-case and lower-case holds true)

Printing/Providing output:

For printing statements in C++ programs, we use the **cout** statement**.**

For example: If you want to print “Hello World!” (without parenthesis) in your code, we will write it in following way:

```
cout << “Hello World!”;
```

A full view of the basic C++ program is given below for the above example:

Code:

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

int main() {
    cout<<”Hello World!”;
}
```

Output:

```
Hello World!
```

Line separator:

For separating different lines in C++, we use **endl** or **‘\n’**.

For example:

Code:

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

int main() {
    cout<<"Hello World1"<<endl;
    cout<<"Hello World2"<<'\n';

}
```

Output:

```
Hello World1
Hello World2
```

Taking input from the user:

To take input from the user, we use the **cin** statement.

For example: If you want to input a number from the user:

```
int n;
cin >> n;
```

A full view of the basic C++ program in which you want to take input of 2 numbers and then print the sum of them:

Code:

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

int main() {
    int a, b, sum;
    cin >> a;
    cin >> b;
    sum = a + b;
    cout << "Sum of two numbers: ";
    cout << sum << endl;
}
```

Input:

```
1
2
```

Output:

```
Sum of two numbers: 3
```

Operators in C++

There are 3 types of operators in C++

- Arithmetic operators
- Relational operators
- Logical operators

Discussing each of them in detail...

Arithmetic operators:

These are used in mathematical operations in the same way as that in algebra.

OPERATOR	DESCRIPTION
+	Add two operands
-	Subtracts second operand from the first
*	Multiplies two operands
/	Divides numerator by denominator
%	Calculates Remainder of division

Relational operators:

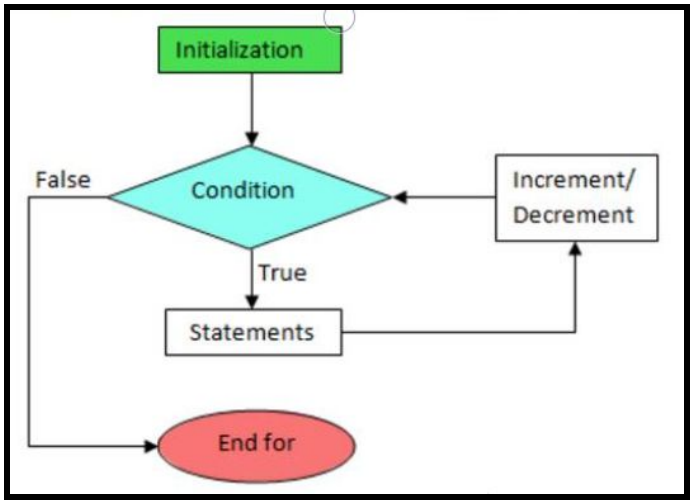
C++ relational operators specify the relation between two variables by comparing them.

Following table shows the relational operators that are supported by C++.

OPERATOR	DESCRIPTION
==	Checks if two operands are equal
!=	Checks if two operands are not equal
>	Checks if operand on the left is greater than operand on the right
<	Checks if operand on the left is lesser than operand on the right
>=	Checks if operand on the left is greater than or equal to operand on the right
<=	Checks if operand on the left is lesser than or equal to operand on the right

Logical operators:

C++ supports 3 types of logical operators. The result of these operators is a boolean value i.e., True(BITWISE ‘1’) or False(BITWISE ‘0’). Refer the visual representation below:



OPERATOR	DESCRIPTION

&&	Logical AND
||	Logical OR
!	Logical NOT

How is data Stored ?

- **For integers:**

  The most commonly used is a signed 32-bit integer type. When you store an integer, its corresponding binary value is stored. There is a separate way of storing positive and negative numbers. For positive numbers the integral value is simply converted into binary value while for negative numbers their 2's complement form is stored.

  ### For negative numbers:

  Computers use 2's complement in representing signed integers because:

  - There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.
  - Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the **addition logic.**

  For example: int i = -4;

  Steps to calculate Two's Complement of -4 are as follows:

  Step1: Take Binary Equivalent of the positive value (4 in this case)

  ```
  0000 0000 0000 0000 0000 0000 0000 0100
  ```

  Step2: Write 1's complement of binary representation by inverting the bits

```
1111 1111 1111 1111 1111 1111 1111 1011
```

Step3: Find 2's complement by adding 1 to the corresponding 1's complement

```
 1111 1111 1111 1111 1111 1111 1111 1011
+0000 0000 0000 0000 0000 0000 0000 0001
-----------------------------------------------------------------
 1111 1111 1111 1111 1111 1111 1111 1100
```

Thus, integer -4 is represented by the above binary sequence in C++.

- **For float and double values:**

  In C++, any value declared with a decimal point is by-default of type double (which is generally of 8-bytes). If we want to assign a float value (which is generally of 4-bytes), then we must use 'f' or 'F' literal to specify that the current value is "float".

  For example:

```
float var = 10.4f          // float value
Double val = 10.4          // double value
```

- **For character values:**

  Every character has a unique integer value, which is called ASCII value. As we know, systems only understand binary language and thus everything has to be stored in the form of binaries. So, for every character there is a corresponding integer code-ASCII code and the binary equivalent of this code is actually stored in memory when we try to store a character.

  For ASCII values, refer the link below:

  https://ascii.cl/

- **Adding int to char**

When we add int to char, we are basically adding two numbers i.e., one corresponding to the int and the other corresponding to the ASCII code for the character.

For example:

Code:

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

int main () {
    cout<< 'a' + 1;
}
```

Output:

```
98
```

Explanation:

The **ASCII value of 'a' is 97**, so it printed **97+1 = 98** as the output.

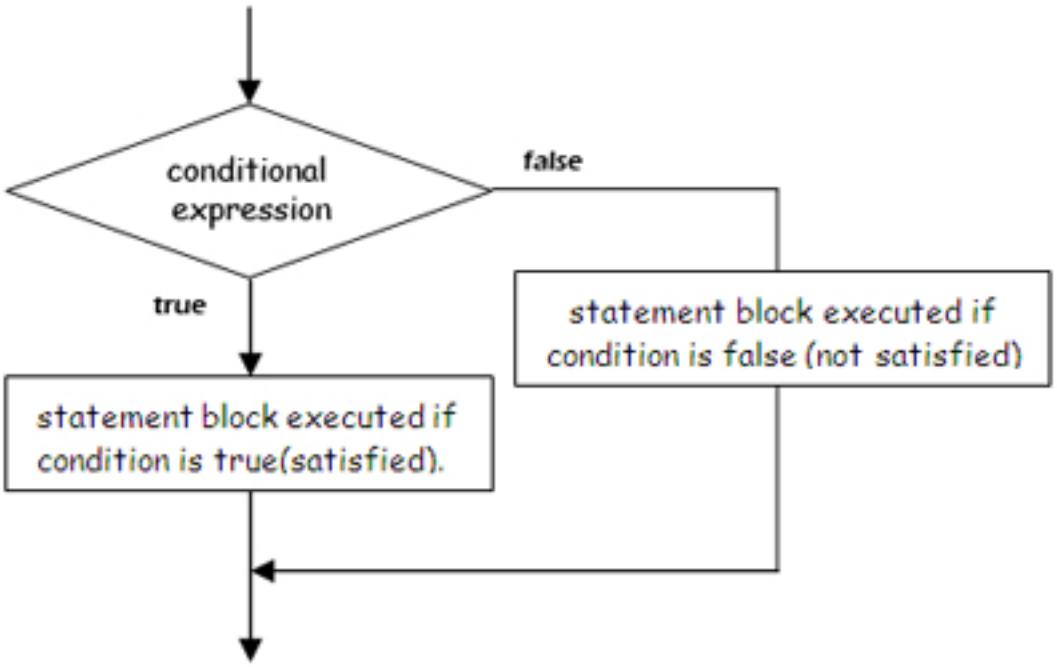
C++ Foundation with Data Structures

Lecture 3 : Conditionals and Loops

Conditional Statements (if else)

Description

Conditionals are used to execute a certain section of code only if some specific condition is fulfilled, and optionally execute other statements if the given condition is false. The result of given conditional expression must be either true or false.



Different variations of this conditional statement are –

- **if statement**
  if statement evaluates the given test expression. If it is evaluated to true, then statements inside the if block will be executed. Otherwise, statements inside if block is skipped.

  Syntax

```
if(test_expression) {
        // Statements to be executed only when test_expression is true
}
```

  Example Code:

```
int main ()
{
        int n=5;
        if ( n<10 )
        {
                cout << "Inside if statement" << endl;
        }
        cout << "Outside if statement" << endl;
}
```

Output

```
Inside if statement
Outside if statement
```

So if the condition given inside if parenthesis is true, then statements inside if block are executed first and then rest of the code. And if the condition evaluates to false, then statements inside if block will be skipped.

- **If – else statement**

if statement evaluates the given test expression. If it is evaluated to true, then statements inside the if block will be executed. Otherwise, statements inside else block will be executed. After that, rest of the statements will be executed normally.

Syntax

```
if(test_expression) {
        // Statements to be executed when test_expression is true

}
else {
        // Statements to be executed when test_expression is false
}
```

Example Code:

```
int main () {
        int a=10,b=20;
        if (a > b){
                cout<< "a is bigger" <<endl;
```

```
        }
        else {
            cout<< "b is bigger" <<endl;
        }
    }
```

Output

b is bigger

- **if – else – if**

Using this we can execute statements based on multiple conditions.

Syntax

```
if(test_expression_1) {
    // Statements to be executed only when test_expression_1 is true
}
else if(test_expression_2) {
    // Statements to be executed only when test_expression_1 is false and test_expression_2 is true
}
else if(test_expression_2) {
    // Statements to be executed only when test_expression_1 & test_expression_2 are false and test_expression_3 is true
}
....
....
else {
    // Statements to be executed only when all the above test expressions are false
}
```

Out of all block of statements, only one will be executed based on the given test expression, all others will be skipped. As soon as any expression evaluates to ttue, that block of statement will be executed and rest will be skipped. If none of the expression evaluates to true, then the statements inside else will be executed.

Example Code:

```
int main() {
	int a = 5;
	if(a < 3) {
		cout<<"one"<< endl;
	}
	else if(a < 10) {
		cout<<"two"<< endl;
	}
	else if(a < 20) {
		cout<<"three"<< endl;
	}
	else {
		cout<<"four"<< endl;
	}
}
```

Output :

two

- **Nested if statement**

We can put another if – else statement inside an if.

Syntax

```
if(test_expression_1) {
	// Statements to be executed when test_expression_1 is true
	if(test_expression_2) {
		// Statements to be executed when test_expression_2 is true
	}
	else {
		// Statements to be executed when test_expression_2 is false
	}
}
```

Example Code:

```
int main() {
        int a = 15;
        if(a > 10) {
                if(a > 20) {
                        cout<<"Hello"<<endl;
                }
                else {
                        cout<<"Hi"<<endl;
                }
        }
}
```

Output :
Hi

return keyword

return is a special keyword, when encountered ends the main. That means, no statement will be executed after return statement. We'll study about return in more detail when we'll study functions.

Example Code:

```
int main() {
        int a = 10;
        if(a > 5) {
                cout<<Hello"<<endl;
                return 0;
        }
        cout<<"Hi"<<endl;
}
```

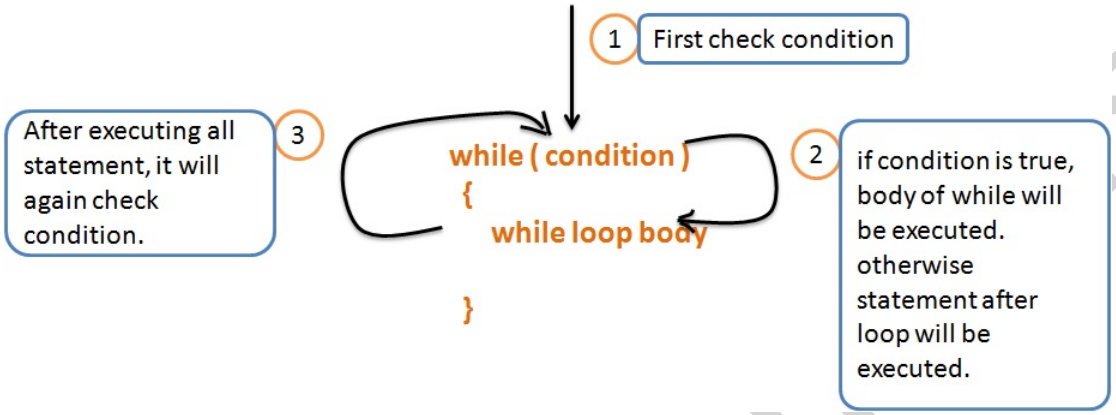
Output:
Hello

while loop

Loop statements allows us to execute a block of statements several number of times depending on certain condition. **while** is one kind of loop that we can use. When executing, if the *test expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Syntax

```
while(test_expression) {
        // Statements to be executed till test_expression is true
}
```



Example Code:

```
int main() {
                int i = 1;
                while(i <= 5) {
                        cout<<i<<endl;
                        i++;
                }
}
```

Output:

1
2
3
4
5

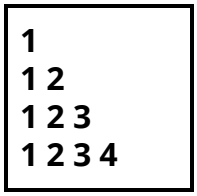
In while loop, first given test expression will be checked. If that evaluates to be true, then the statements inside while will be executed. After that, the condition will be checked again and the process continues till the given condition becomes false.

Patterns

Introduction

Patterns are a good application of loops. These will help you get yourself a better clarity over implementing loops.

Suppose, we want to print the following pattern:



Solution:

- ➔ Here, you can see that we have 4 rows.
- ➔ Now, moving to find the generic number of columns, it's clear that the i-th row (where i = 1, 2, ...) has i columns. Hence, the number of columns for i-th row is i.
- ➔ Visually, it's also clear that we want to print numbers starting from 1 to the row-th number. Like for the first row, we print numbers starting from 1 and ending at 1, for second row numbers are starting from 1 and ending at 2 and so on...

The above approach can be generalised in the following way:

- We start to figure out the number of rows that our pattern requires.
- Now, we should know how many columns do we have to print in the generic i-th row.

- Once, we have figured out the number of rows and columns, then we should focus on what to print.

There are two popular types of patterns-related questions that are usually posed:

- Square Pattern
- Triangular Pattern

Coding a pattern:

Now, starting to code this pattern. It's clear that we have to start with a loop for running over the row. So let's first begin with writing the basic structure and the loop for the row:

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

int main() {

        int n = 4;    //Number of rows taken input

        int i = 1;    // for iterating over rows starting from the first row

        while(i < n) {

                i++;      // for iterating over each row
        }
}
```

Here, we have iterated over each row.

Now for printing the above pattern, we also need to iterate over columns and that too, on each row.

For this, we will run another loop inside the while() loop. Let's see that also...

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

int main() {

        int n = 4;                      //Number of rows taken input

        int i = 1;                      // for iterating over rows starting from the first row

        while(i < n) {

                int j = 1;                // for iterating over columns
                while(j <= i) {           // since for each column row, we will be iterating
                                          //  over each column

                        cout << j;
                        j = j + 1;        // as we have discussed above that i-th row will
                                          // have i columns

                }
                cout << endl;
                i++;                      // for iterating over each row
        }
}
```

This will result in the above pattern that we discussed. These types of patterns are known as **triangular patterns** as the shape resembles a triangle.

Similarly, there are other patterns too like:

```
11111
11111
11111
11111
11111
```

This type of pattern is **square Pattern.** Try coding it out yourself...

Once, you have tried it, you can check your solution with the one given below:

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

int main() {

	int n = 5;

	int i = 0;
	while(i <= n) {

		int j = 1;
		while(j <= n) {
			cout << 1;
			j = j + 1;
		}
		cout << endl;
		i = i + 1;
	}
}
```

Like these integral valued patterns there are character valued patterns too. The only difference is that here we are printing the character values.

For example:

Given below are three patterns of the characters. They are very similar to what we discussed above.

******	abcde	A
******	abcde	AB
******	abcde	ABC
******	abcde	ABCD
******	abcde	ABCDE

Practice Examples:

Print the following patterns:

1)

```
55555
45555
34555
23455
12345
```

2)

```
ABCDE
ABCD
ABC
AB
A
```

3)

```
12344321
123**321
12****21
1******1
```

Try these yourselves and in case, you find yourself stuck then following is the approach provided...

Solution 1

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

int main() {

   int i = 5, j, k;
   while(i >= 1) {
         k = i;
         j = 1;
         while(j <= 5) {
                  if(k <= 5) {
                        cout << k;
                  }
                  else {
                        cout << 5;
                  }
                  k = k + 1;
                  j = j + 1;
         }
         cout << endl;
         i = i - 1;
    }
}
```

Solution 2

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

int main() {

         int i = 1, j;
         while(i <= 5) {
                  j = 5 - i + 1;
                  int k = 1;
                  while(k <= j) {
                           cout<<(char)(64 + k);
                           k = k + 1;
                  }
                  cout << endl;
                  i = i + 1;
         }
}
```

Solution 3

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

int main() {

        int i = 4, j;
        while(i >= 1) {

                j = 1;
                while(j <= 4) {
                        if(j <= i) {
                                cout << j;
                        }
                        else {
                                cout << "*";
                        }
                        j = j + 1;
                }

                while(j >= 1) {
                        if(j <= i) {
                                cout << j;
                        }
                        else {
                                cout << "*";
                        }
                        j = j - 1;
                }
                i = i - 1;
                cout << endl;
        }
}
```

Advanced patterns

Here, we are gonna study the patterns that include a form of mirror images at some part of the program. We'll be solving these like we did in the "Patterns 1" portion by following the same three steps of identifying the number of rows, then identifying the columns and finally what to print.

So directly moving on to programming questions:

Example 1: Print the following pattern:

```
 1  2  3  4 5
16          6
15          7
14          8
13 12 11 10 9
```

Try this out, in case you are stuck, just checkout the code below for the same.

Note: The pattern may not be totally similar, meaning it may be a bit shifted due to the difference in the number of digits of a 2-digit number and a 1-digit number.

Solution:

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

int main() {

   int i = 1, k = 6, l = 13, m = 16;
        while(i <= 5)
        {
                int j = 1;
                while(j <= 5)
                {
                        if(i == 1)
                                cout << j <<" ";
                         else if(j == 5)
                                cout << k++ << " ";
                        else if(i == 5)
```

```
                        cout << i-- << " ";
                    else if(j == 1)
                        cout << m-- << " ";
                    else
                        cout<< "  ";
                    j++;
                }
                cout << endl;
                i++;
            }
}
```

Example 2: Print the following pattern:

```
    1
   123
  12345
 1234567
123456789
 1234567
  12345
   123
    1
```

Try this out, in case you are stuck, just checkout the code below for the same.

Hint: Here you can see that there are generally two different patterns that you have to make: one spreading in the downwards directions and other one in continuation, shrinking towards the bottom.

Approach: Let's first try to make the downwards growing pattern i.e., upto row number 5. It is clearly visible that each row has 2n-1 numbers (where n = 1, 2, 3, 4, 5). Now, as we know the number of rows, columns in each row and the general formula that our pattern follows, it can be easily coded.

Moving on to the lower triangle of the pattern, if we start counting these rows in reverse order starting from number 4 down to number 1, each row contains 2n-1 numbers over here also, and can be similarly done using the upper triangle pattern but in reverse order.

Solution:

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

int main() {
   int i = 1;
   while (i <= 5)                  // Starting our first part of the pattern.
   {
         int j = i;
         while (j < 5)
         {
               cout<<" ";
               j++;
         }
         int k = 1;
         while(k < 2*i)            // upto k = 2n-1
         {
               cout << k;
               k++;
         }
         i++;
         cout << endl;
   }

   i = 4;                          // Starting our second pattern in reverse order.
   while (i > 0)
   {
         int j = 5;
         while (j > i)
         {
               cout << " ";
               j--;
         }
         int k = 1;
         while (k < 2*i)
         {
                  cout << k;
                  k++;
         }
         cout << endl;
         i--;
   }
```

```
}
```

You have already practised most of these questions earlier. So now directly moving on to some of the practice questions.

PRACTICE: Print the following programs

```
1    1
 2  2
  3 3
   4
  3 3
 2  2
1    1
```

Solutions:

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

int main()
{
   int i = 1, k=1;
   int m[7][7]={0};       // Initialises all the elements of the array equal to 0, you will
                          //   study about arrays in your further sessions.
   while (i <= 7)
   {
         int j = 1;
         while (j <= 7)
         {
               if (j == i || 8-i == j)
                     m[i-1][j-1]=k;
               j++;
         }
         if (i < 4)
               k++;
         else
               --k;
         i++;
   }

   i = 0;
   while (i < 7)
```

```
  {
        int j = 0;

        while (j < 7)
        {
                if(m[i][j]==0)
                        cout << " ";
                else
                        cout << m[i][j];
                j++;
        }
        i++;
        cout << endl;
  }
}
```

For more questions, visit the link:

http://cbasicprogram.blogspot.com/2012/04/number-patterns.html

C++ Foundation with Data Structures

Lecture 4 : Loops, Keywords, Associativity and Precedence

for loop

Loop statements allows us to execute a block of statements several number of times depending on certain condition. **for** loop is kind of loop in which we give initialization statement, test expression and update statement can be written in one line.

Inside for, three statements are written –
a. Initialization – used to initialize your loop control variables. This statement is executed first and only once.
b. Test condition – this condition is checked everytime we enter the loop. Statements inside the loop are executed till this condition evaluates to true. As soon as condition evaluates to false, loop terminates and then first statement after for loop will be executed next.
c. Updation – this statement updates the loop control variable after every execution of statements inside loop. After updation, again test conditon is checked. If that comes true, the loop executes and process repeats. And if condition is false, the loop terminates.

```
for (initializationStatement; test_expression; updateStatement) {
        // Statements to be executed till test_expression is true
}
```

Example Code :

```
int main(){
        for(int i = 0; i < 3; i++){
            cout<<"Inside for Loop : "<<i<<endl;
        }
        cout<<"Done";
}
```

Output:

```
Inside for Loop : 0
Inside for Loop : 1
Inside for Loop : 2
Done
```

In for loop its not compulsory to write all three statements i.e. initializationStatement, test_expression and updateStatement. We can skip one or more of them (even all three)

Above code can be written as:

```
int main () {
        int i=1;    // initialization is done outside the for loop
        for (; i <= 5; i++) {
                cout<<i<<end;
        }
}
```

OR

```
int main () {
        int i=1;     //// initialization is done outside the for loop
        for (; i <= 5; ) {
                cout<<i<<end;
                i++;   // updateStatement written here
        }
}
```

We can also skip the test_expression. See the example below :

Variations of for loop

- The three expressions inside for loop are optional. That means, they can be omitted as per requirement.

  Example code 1: Initialization part removed –

```
int main(){
        int i = 0;
        for(; i < 3; i++){
                cout<<i<<endl;
        }
}
```

  Output:

0
1
2

Example code 2: Updation part removed

```
int main(){
        for(int i = 0; i < 3; ){
                cout<<i<<endl;
                 i++;
    }
}
```

Output:

0
1
2

Example code 3: Condition expression removed , thus making our loop infinite –

```
int main(){
        for(int i = 0 ; ;i++){
                cout<<i<<endl;
        }
}
```

Example code 4:

We can remove all the three expression, thus forming an infinite loop-

```
#include<iostream>
using namespace std;
int main(){
        for(; ;){
                cout<<"Inside for loop : "<<endl;
        }
}
```

- **Multiple statements inside for loop**

We can initialize multiple variables, have multiple conditions and multiple update statements inside a *for loop*. We can separate multiple statements using comma, but not for conditions. They need to be combined using logical operators.

Example code:

```
int main(){
        for(int i = 0,j = 4; i < 5 && j >= 0; i++, j--){
                cout<<i<<" "<<j<<endl;
        }
}
```

Output:

```
0 4
1 3
2 2
3 1
4 0
```

break and continue

1. **break statement**: The break statement terminates the loop (for, while and do. while loop) immediately when it is encountered. As soon as break is encountered inside a loop, the loop terminates immediately. Hence the statement after loop will be executed next.
2. **continue statement:** The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else. (caution always update the counter in case of while loop else loop will never end)

```
while(test_expression) {
        // codes
        if (condition for break) {
                break;
        }
        //codes
}
```

```
for (initializationStatement; test_expression; updateStatement) {
        // codes
        if (condition for break) {
                break;
        }
        //codes
}
```

❖ **break**

- Example: (using break inside for loop)

```
int main () {
        for (int i=1; i <= 10; i++) {
                cout<<i<<end;
                if(i==5)
                {
                        break;
                }
        }
}
```

Output:
1
2
3
4
5

- Example: (using while loop)

```
int main () {
        int i = 1;
        while (i <= 10) {
                cout<<i<<endl;
                if(i==5)
                {
                        break;
                }
                i++;
        }
}
```

Output:

1
2
3
4
5

- Inner loop break:

When there are two more loops inside one another. Break from innermost loop will just exit that loop.

Example Code 1:

```
int main () {
        for (int i=1; i <=3; i++) {
                cout<<i<<end;
                for (int j=1; j<= 5; j++)
                {
                        cout<<"in…" <<endl;
                        if(j==1)
                        {
                                break;
                        }
                }
        }
}
```

Output:

1
in…
2
in…
3
in…

Example Code 2:

```
int main () {
        int i=1;
```

```
        while (i <=3) {
                cout<<i<<end;
                int j=1;
                while (j <= 5)
                {
                        cout<<"in…" <<endl;
                        if(j==1)
                        {
                                break;
                        }
                        j++;
                 }
                i++;
        }
}
```

Output:
1
in…
2
in…
3
in…

❖ Continue

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loopx.

- Example: (using for loop)

```
int main () {
        for (int i=1; i <= 5; i++) {
                if(i==3)
                {
                        continue;
                }
                cout<<i<<end;
```

```
        }
}

Output:
1
2
4
5
```

- **Example: (using while loop)**

```
int main () {
        int i=1;
        while (i <= 5) {
                if(i==3)
                {
                        i++;
                   // if increment isn't done here then loop  will run
        infinite time for i=3
                        continue;
                }
                cout<<i<<end;
                i++;
        }
}

Output:
1
2
4
5
```


Scope of variables

Scope of variables is the curly brackets {} inside which they are defined. Outside which they aren't known to the compiler. Same is for all loops and conditional statement (if).

❖ Scope of variable - for loop

```
for (initializationStatement; test_expression; updateStatement) {
        // Scope of variable defined in loop
}
```

Example:

```
int main() {
        for (int i=0; i<5; i++) {
                int j=2;        // Scope of i and j are both inside the loop they
                can't be used outside
}
```

❖ Scope of variable for while loop

```
while(test_expression) {
        // Scope of variable defined in loop
}
```

```
int main() {
        int i=0;
        while(i<5)
        {
                int j=2;   // Scope of i is main and scope of j is only the loop
                i++;
        }
}
```

❖ Scope of variable for conditional statements

```
if(test_expression) {
        // Scope of variable defined in the conditional statement
}
```

```
int main () {
        int i=0;
        if (i<5)
        {
                int j=5;          // Scope of j is only in this block
```

```
    }
        // cout<<j;  → This statement if written will give an error because
                       scope of j is inside if and is not accessible outside if.
}
```

Increment Decrement operator

Explanation

Pre-increment and *pre-decrement* operators' increments or decrements the value of the object and returns a reference to the result.

Post-increment and *post-decrement* creates a copy of the object, increments or decrements the value of the object and returns the copy from before the increment or decrement.

Post-increment(a++):

This increases value by 1, but uses old value of a in any statement.

Pre-increment(++a):

This increases value by 1, and uses increased value of a in any statement.

Post-decrement(a--):

This decreases value by 1, but uses old value of a in any statement.

Pre-decrement(++a):

This decreases value by 1, and uses decreased value of a in any statement.

```
int main () {

        int I=1, J=1, K=1, L=1;

        cout<<I++<<' '<<J-- <<' '<<++K<<' '<< --L<<endl;

        cout<<I<<' '<<J<<' '<<K<<' '<<L<<endl;

}
```

Output:
1 1 2 0

2 0 2 0

Bitwise Operators

Operator	Name	Example	Result	Description
a & *b*	and	4 & 6	4	1 if both bits are 1.
a | *b*	or	4 | 6	6	1 if either bit is 1.
a ^ *b*	xor	4 ^ 6	2	1 if both bits are different.
~*a*	not	~4	-5	Inverts the bits. (Unary bitwise compliment)
n << *p*	left shift	3 << 2	12	Shifts the bits of *n* left *p* positions. Zero bits are shifted into the low-order positions.
n >> *p*	right shift	5 >> 2	1	Shifts the bits of *n* right *p* positions.


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

int main() {
   int a = 19;  // 19 = 10011
   int b = 28;  // 28 = 11000
  int c = 0;

  c = a & b;          //  16 = 10000
  cout << "a & b = " << c << endl ;

  c = a | b;          // 31 = 11111
  cout << "a | b = " << c << endl ;

  c = a ^ b;          // 15 = 01111
  cout << "a ^ b = " << c << endl ;

  c = ~a;             // -20 = 01100
  cout << "~a = " << c << endl ;

  c = a << 2;         // 76 = 1001100
  cout << "a << 2 = " << c << endl ;
```

```
  c = a >> 2;          // 4 = 00100
  cout << "a ≫ 2 = " << c << endl ;

  return 0;
}
```

Output

```
a & b = 16
a | b = 31
a ^ b = 15
~a = -20
a << 2 = 76
a >> 2 = 4
```


Precedence and Associativity

- **Operator precedence** determines which operator is performed first in an expression with more than one operators with different precedence.
  For example, 10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30.
- **Associativity** is used when two operators of same precedence appear in an expression. Associativity can be either **L**eft **t**o **R**ight or **R**ight **t**o **L**eft. For example, '*' and '/' have same precedence and their associativity is **L**eft **t**o **R**ight, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Precedence and Associativity are two characteristics of operators that determine the evaluation order of subexpressions in absence of brackets.

Note : We should generally use add proper brackets in expressions to avoid confusion and bring clarity.

1) Associativity is only used when there are two or more operators are of same precedence.
The point to note is associativity doesn't define the order in which operands of a single operator are evaluated. For example, consider the following program,

associativity of the + operator is left to right, but it doesn't mean f1() is always called before f2(). The output of following program is in-fact compiler dependent.

```
// Associativity is not used in the below program. Output is compiler dependent.
 x = 0;
int f1() {
  x = 5;
  return x;
}
int f2() {
  x = 10;
  return x;
}
int main () {
  int p = f1() + f2();
  cout<<x;
  return 0;
}
```

2) All operators with same precedence have same associativity
This is necessary, otherwise there won't be any way for compiler to decide evaluation order of expressions which have two operators of same precedence and different associativity. For example, + and – have same associativity.

3) There is no chaining of comparison operators in C++
In C++ expression (a>b>c) will be treated as (a>b)>c .For example, consider the following program.. For example, consider the following program. The output of following program is "FALSE".

```
int main ()
{
  int a = 10, b = 20, c = 30;

  // (c > b > a) is treated as ((c > b) > a), associativity of '>'
  // is left to right. Therefore, the value becomes ((30 > 20) > 10). Since
(30>20) is true thus its answer is 1. So the expression becomes (1 > 20).

  if (c > b > a)
   cout<<"TRUE";
  else
```

```
    cout<<"FALSE";
  return 0;

}
```

Following is the Precedence table along with associativity for different operators.

OPERATOR	DESCRIPTION	ASSOCIATIVITY
( ) [ ] . -> ++ —	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ — + – ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right

^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

C++ Foundation with Data Structures

Lecture 5 : Functions, Variables and Scope

s

Functions

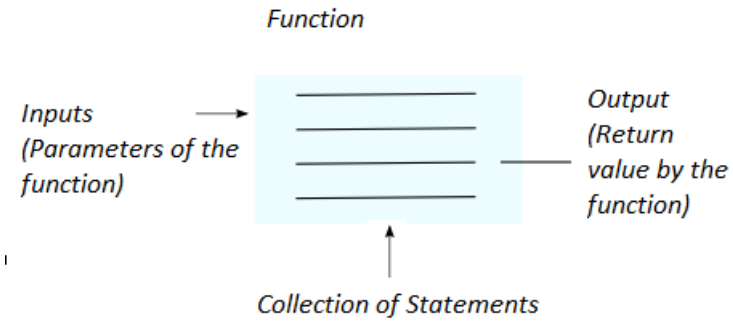
A Function is a collection of statements designed to perform a specific task. Function is like a black box that can take certain input(s) as its parameters and can output a value which is the return value. A function is created so that one can use it as many time as needed just by using the name of the function, you do not need to type the statements in the function every time required.

Defining Function

return_type function_name(parameter 1, parameter 2, ……) {

statements;

}



- ***return type:*** A function may return a value. The *return type* of the function is the data type of the value that function returns. Sometimes function is not required to return any value and still performs the desired task. The return type of such functions is ***void*.**

Example:

Following is the example of a function that sum of two numbers. Here input to the function are the numbers and output is their sum.

```
1. int findSum( int a, int b){
2.    int sum = a + b;
3.    return  sum;
4. }
```

Function Calling

Now that we have read about how to create a function lets see how to call the function. To call the function you need to know the name of the function and number of parameters required and their data types and to collect the returned value by the function you need to know the return type of the function.

Example

```
#include<iostream>
using namespace std;
int findSum( int a, int b){
   int sum = a + b;
   return  sum;
}
int main () {
  int a = 10, b = 20;
  int c = findSum (a, b); // function findSum () is called using its name and by knowing
  cout<< c;                    // the number of parameters and their data type.
}                                 // integer c is used to collect the returned value by the function
```

Output:
30

IMPORTANT POINTS:

- ***Number of parameter*** and their ***data type*** while calling must match with function signature. Consider the above example, while calling function ***findSum ()*** the number of parameters are two and both the parameter are of integer type.

- It is okay not to collect the return value of function. For example, in the above code to find the sum of two numbers it is right to print the return value directly.

  "***cout<<findSum(a,b);***"

- **void return type functions:** These are the functions that do not return any value to calling function. These functions are created and used to perform specific task just like the normal function except they do not return a value after function executes.

Following are some more examples of functions and their use to give you a better idea.

Function to find area of circle

```
1.  #include<iostream>
2.  using namespace std;
3.  double findArea( double radius){   //return type of the function is double.
4.          double area = 3.14 * radius * radius;
5.          return  area;
6.  }
7.  int main () {
8.          double radius = 5.8;
9.           double c = findArea (radius);
10.          cout<< c;
11. }
```

Function to print average

```
1.  #include<iostream>
2.  using namespace std;
3.
4.  void printAverage(int a, int b ){  //return type of the function is void
12.         int avg = (a + b) / 2;
            cout<<avg << endl;
5.  }                       // This function does not return any value
6.
7.  int main () {
8.          int a = 15, b = 25;
9.           printAverage (a, b);
10.}
```

Why do we need function?

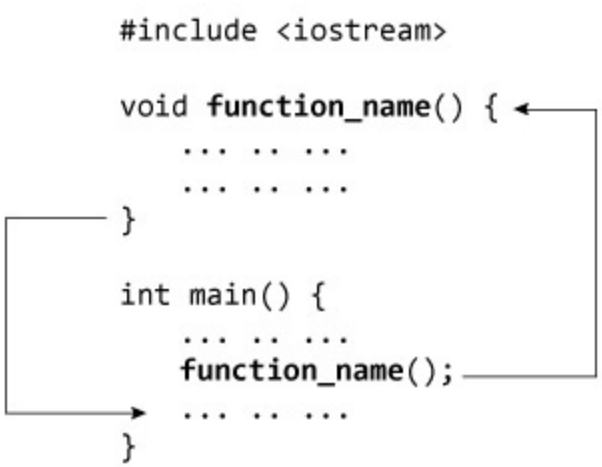
- **Reusability:** Once a function is defined, it can be used over and over again. You can call the function as many time as it is needed, which saves work. Consider that you are required to find out the area of the circle, now either you can apply the formula every time to get the area of circle or you can make a function for finding area of the circle and invoke the function whenever it is needed.
- **Neat code:** A code created with function is easy to read and dry run. You don't need to type the same statements over and over again, instead you can invoke the function whenever needed.
- **Modularisation –** Functions help in modularising code. Modularisation means to divides the code in small modules each performing specific task. Functions helps in doing so as they are the small fragments of the programme designed to perform the specified task.
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.

How does function calling works?

Consider the following code where there is a function called findsum which calculates and returns sum of two numbers.

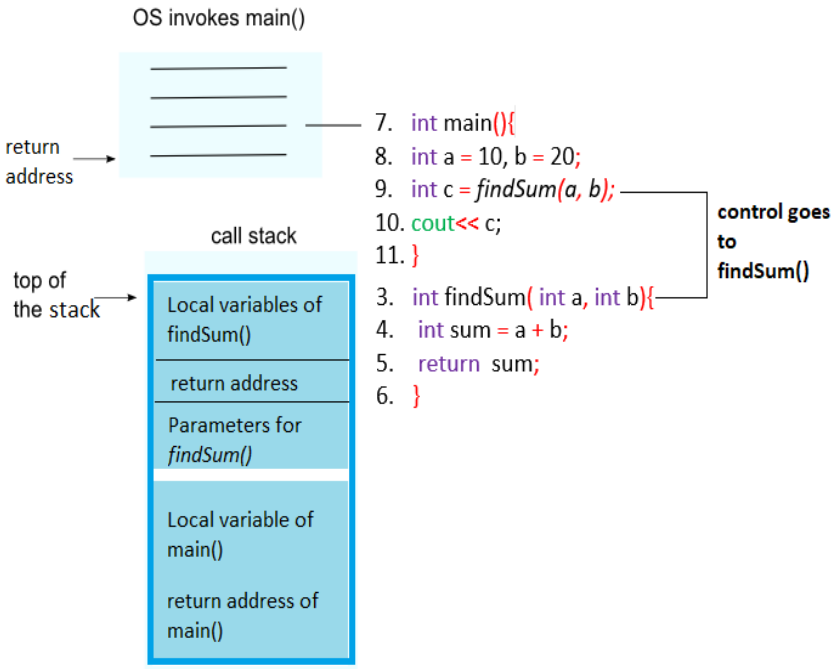
//Find Sum of two integer numbers

```
1.  #include<iostream>
2.  using namespace std;
3.  int findSum( int a, int b){  // return type of the function is int.
4.          int sum = a + b;
5.          return  sum;
6.  }
7.  int main () {
8.          int a = 10, b = 20;
9.          int c = findSum (a, b);
10.         cout<< c;  }
```



The function being called is called **callee(**here it is findsum function) and the function which calls the callee is called the **caller** (here main function is the caller) .
When a function is called, programme control goes to the entry point of the function. Entry point is where the function is defined. So focus now shifts to callee and the caller function goes in paused state .

For Example: In above code entry point of the function ***findSum ()*** is at line number 3. So when at line number 9 the function call occurs the control goes to line number 3, then after the statements in the function ***findSum ()*** are executed the programme control comes back to line number 9.



Role of stack in function calling (call stack)

A call stack is a storage area that store information about the *active* function and paused functions. It stores parameters of the function, return address of the function and variables of the function that are created statically.

Once the function statements are terminated or the function has returned a value, the call stack removes all the information about that function from the stack.

Benefits of functions

- **Modularisation**
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.
- **Neat code:** A code created with function is easy to read and dry run.

Variables and Scopes

Local Variables

Local variable is a variable that is given a local scope. Local variable belonging to a function or a block overrides any other variable with same name in the larger scope. Scope of a variable is part of a programme for which this variable is accessible.

Example:

```
1. #include<iostream>
2. using namespace std;
3. int main(){
4.       int a = 10;
5.       if (1){
6.             int a = 5;
7.             cout<< a<<endl;
8.       }
9.       cout<< a;
10.}
```

Output

5
10

In the above code the variable ***a*** declared inside the block after if statement is a local variable for this block and is different from the the variable ***a*** declared outside this block. Both these variable are allocated to different memory.

Lifetime of a Variable

The lifetime of a variable is the time period for which the declared variable has a valid memory. Scope and lifetime of a variable are two different concepts, scope of a variable is part of a programme for which this variable is accessible whereas lifetime is duration for which this variable has a valid memory.

Loop variable

Loop variable is a variable which defines the loop index for each iteration.

Example

“for (int *i* = 0; i < 3; i++) { // variable ***i*** is the loop variable
……;
……..;
statements;
} “

For this example, variable ***i*** is the loop variable.

Variables in the same scope

Scope is part of programme where the declared variable is accessible. In the same scope, no two variables can have name. However, it is possible for two variables to have same name if they are declared in different scope.

Example:

```
1. #include<iostream>
2. using namespace std;
3. int main () {
4.         int a = 10;
5.         int a = 5; // two variables with same name, the code will not
   compile
6.         cout<< a;
7. }
```

For the above code, there are two variables with same name ***a*** in the same scope of main () function. Hence the above code will not compile.

But the code given below will compile because the two variables with same name ***a*** are declared in different scope.

```
1. #include<iostream>
2. using namespace std;
3. int main () {
4.         int a = 10;
5.         if (1){
6.             int a = 5;
7.             cout<< a<<endl;
8.         }
9.         cout<< a;
10.}
```

Pass by value:

When the parameters are passed to a function by pass by value method, then the formal parameters are allocated to a new memory. These parameters have same value as that of actual parameters. Since the formal parameters are allocated to new memory any changes in these parameters will not reflect to actual parameters.

Example:

```
//Function to increase the parameters value
#include<iostream>
using namespace std;
void increment (int x, int y) {          // x and y are formal parameters
        x++;
        y = y + 2;
        cout<<x<<": "<<y<<endl;
}

int main () {
        int a = 10, b = 20;
        increment (a, b);                // a and b are actual parameters
        cout<< a<<": "<<b;
}
```

Output:

11: 22
10: 20

For the above code, changes in the values of ***x*** and ***y*** are not reflected to ***a*** and ***b*** because x and y are formal parameters and are local to function increment so any changes in their values here won't affect variables a and b inside main.

C++ Foundation with Data Structures

Lecture 6 : Arrays

What are arrays?

In cases where there is a need to use several variables of same type, for storing, example, names or marks of ‘n’ students we use a data structure called arrays. Arrays are basically collection of elements having same name and same data type. Using arrays, saves us from the time and effort required to declare each of the element of array individually.

Creating an array

The syntax for declaring an array is:

Data_type array_name [array_size] ;

Example :

```
float marks[5];
```

Here, we declared an array, marks, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Similarly, we can declare array of type int as follows:

```
int age[10];
```

How are arrays stored?

The elements of arrays are stored contiguously, i.e., at consecutive memory locations. The name of the array actually has the address of the first element of the array. Hence making it possible to access any element of the array using the starting address.

Example:

```
int age[ ] = {10, 14, 16, 18, 19};
```

Suppose the starting address of an array is 1000,then address of second and third element of array will be 1004, 1008 respectively and so on.

10	14	16	18	119
1000	1004	1008	1012	1016

Accessing elements of an array

An element of array could be accessed using indices. Index of an array starts from 0 to n-1, where n is the size of the array.

Syntax for accessing array element is :

Array_name[index]

Suppose we declare array age of size 5.

```
int age[10];

age[2] = 4      //  stores value 4 at index 2
```

		4		
age[0]	age[1]	age[2]	age[3]	age[4]

Here the first element is age[0], second element is age[1] and so on.

Default values

Arrays must always be initialized. If the array is not initialized the respective memory locations will contain garbage by default. Hence, any operation on uninitialized array will lead to unexpected results.

Initializing array at the time of declaration

Elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in curly braces {}.

Example

```
int age[5] = {5, 2, 10, 4, 12};
```

Alternatively,

```
int age[ ] = {5, 2, 10, 4, 12};
```

5	2	10	4	12
age[0]	age[1]	age[2]	age[3]	age[4]

If array is initialized like this -

```
int age[10] = {5, 2, 10, 4, 12};
```

Then, in memory an integer array of size 10 will be declared. That is, a continuous memory block of 40 bytes (to hold 10 integers) will be allocated. And first 5 values of age are provided, rest will be 0.

Here,

age[0] is equal to 5

age[1] is equal to 2

age[2] is equal to 10

age[3] is equal to 4

age[4] is equal to 12

And age[5] to age[9] is equal to 0.

sizeof operator

sizeof is an operator used to determine the length of the array, i.e., the number of elements in the array .

Example:

```
int main( ){
    int myArray[ ] = {5, 4, 3, 2, 1};
    int size = sizeof(myArray);
    cout << size;
```

```
    return 0;
}
```

Output :

20

Passing arrays to a function

Arrays can be passed to a function as an argument. When an array is passed as an argument, only the starting address of the array gets passed to the function as an argument. Since, only the starting address gets passed to the function as opposed to whole array, the size of the array cannot be determined in function using sizeof operator. Hence, the arrays that are passed as an argument to a function must always be accompanied by its size as another argument.

Syntax of the function call to pass an array :

function_name(array_name, array_size);

Syntax of the function definition that takes array as an argument :

return_type function_name(data_type array_name, int size, <other arguments>){

```
    //function body
```

}

NOTE : Square brackets '[ ]' are not used at the time of function call.

Example:

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

```
void printAge(int age[ ], int n){
        for(int i=0 ; i < n ; i++){
                cout << age[i] << endl;
        }
}
int main(){
        int age[5] = {11, 14, 15, 18, 20};
        printAge(age, 5);
        return 0;
}
```

Output:

11

14

15

18

20

Searching and Sorting

Searching

Searching means to find out whether a particular element is present in the given array/list. For instance, when you visit a simple google page and type anything that you want to know/ask about, basically you are searching that topic in the google's vast database for which google is using some technique in order to provide the desired result to you.

There are basically three types of searching techniques:

- Linear search
- Binary search
- Ternary search

We have already discussed the linear searching technique. In this module, we will be discussing binary search. We will not be discussing ternary search over here, for that You may explore the link mentioned below to know more about it.

Link for ternary search: (It is preferred to first-of-all clearly understand binary search and then move onto the ternary search)

https://cp-algorithms.com/num_methods/ternary_search.html

Linear Search

In this, we have to find a particular element in the given array and return the index of the same, in case it is not present the convention is to return -1. This can be simply done by traversing each element index one-by-one and then comparing the value at that index with the desired value.

Suppose you want to find a particular element in the sorted array, then following this technique, you have to traverse all the array elements for searching one element but guess if you only have to search at most half of the array indices for performing the same operation. This can be achieved through binary search.

Advantages of Binary search:

- This searching technique is fast and easier to implement.
- Requires no extra space.
- Reduces time complexity of the program to a greater extent. (The term **time complexity** might be new to you, you will get to understand this when you will be studying algorithmic analysis. For now, just consider it as the time taken by a particular algorithm in its execution and time complexity is determined by the number of operations that are performed by that algorithm i.e., time complexity is directly proportional to the number of operations in the program).

Binary Search

Now, let's look at what binary searching is.

Prerequisite: Binary search has one pre-requisite, the array must be sorted unlike the linear search where elements could be any order.

Let us consider the array:

0	1	2	3	4
1	**2**	**3**	**4**	**5**

You can see it is an array of size 5 with elements inside the boxes and indices above them. Our target element is 2.

Steps:

1. Find the middle index of the array. In case of even length of the array consider the index that appears first out of two middle ones.
2. Now, we will compare the middle element with the target element. In case they are equal then we will simply return the middle index.
3. In case they are not equal, then we will check if the target element is less than or greater than the middle element.
    - In case, the target element is less than the middle element, it means that the target element, if present in the given array, will be on the left side of the middle element as the array is sorted.
    - Otherwise, the target element will be on the right side of the middle element.
4. This helps us discard half of the length of the array and we have reduced the number of operations.
5. Now, since we know that the element is on the left, we will assume that the array's starting and ending indices to be:
    - **For left:** start = 0, end = n/2-1
    - **For right:** start = n/2, end = n-1

      where n are the total elements in the array.
6. We will repeat this process until we find the target element. In case start and end becomes equal or start becomes more than end, and we haven't found the target element till now, we will simply return -1 as that element is not present in the array.

In the example above, the middle element is 3 at index 2, and the target element is 2 which is greater than the middle element, so we will move towards the left part. Now marking start = 0, and end = n/2-1 = 1, now middle = (start + end)/2 = 0. Now comparing the 0-th index element with 2, we find that 2 > 1, hence we will be moving towards the right. Updating the start = 1 and end = 1, middle becomes 1, comparing 1-st index of the array with the target element, we find they are equal, meaning from here we will simply return 1 (the index of the element).

Now try implementing the approach yourself...

Implementation

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

int binarySearch(int arr[], int n, int x) {
        int start = 0, end = n - 1;
        while(start <= end) {
                int mid = (start + end) / 2;
                if(arr[mid] == x) {
                        return mid;
                }
                else if(x < arr[mid]) {
                        end = mid - 1;
                }
                else {
                        start = mid + 1;
                }
        }

        return -1;
}

int main() {
        // Take array input from the user
        int n;
        cin >> n;

        int input[100];

        for(int i = 0; i < n; i++) {
                cin >> input[i];
        }

        int x;
        cin >> x;

        cout << binarySearch(input, n, x) << endl;
}
```

Note: There are other approaches also to perform binary searching like Recursion which you will study in advanced topics.

Sorting

Sorting means an arrangement of elements in an ordered sequence either in increasing(ascending) order or decreasing(descending) order. Sorting is very important and many softwares and programs use this. Binary search also requires sorting. There are many different sorting techniques. The major difference is the amount of space and time they consume while being performed in the program.

For now, we will be discussing the following sorting techniques only:

- Selection sort
- Bubble sort
- Insertion sort

Let's discuss them one-by-one...

Selection sort:

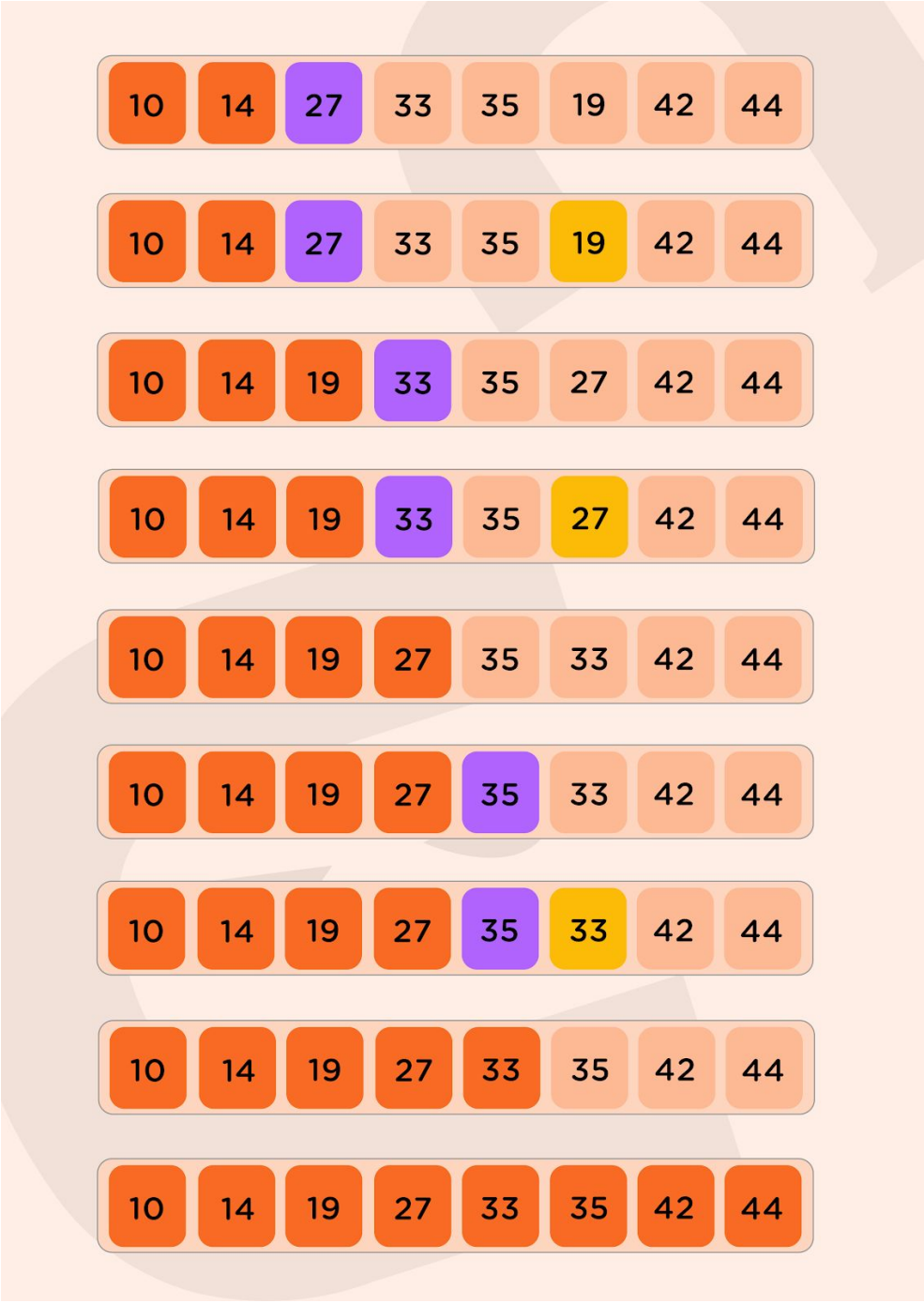
Steps: (sorting in increasing order)

1. First-of-all, we will find the smallest element of the array and swap that with the element at index 0.
2. Similarly, we will find the second smallest and swap that with the element at index 1 and so on...
3. Ultimately, we will be getting a sorted array in increasing order only.

Let us look at the example for better understanding:

Consider the following depicted array as an example. You want to sort this array in increasing order.

Following is the pictorial diagram for better explanation of how it works:



This is how we obtain the sorted array at last.

Now looking at the implementation of Selection Sort:

Implementation of selection sort

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

void selectionSort(int input[], int n) {
        for(int i = 0; i < n-1; i++ ) {
        // Find min element in the array
        int min = input[i], minIndex = i;
        for(int j = i+1; j < n; j++) {
                if(input[j] < min) {
                        min = input[j];
                        minIndex = j;
                }
        }
        // Swap
        int temp = input[i];
        input[i] = input[minIndex];
        input[minIndex] = temp;
        }
}

int main() {
        int input[] = {3, 1, 6, 9, 0, 4};
        selectionSort(input, 6);
        for(int i = 0; i < 6; i++) {
                cout << input[i] << " ";

        }
        cout << endl;
}
```

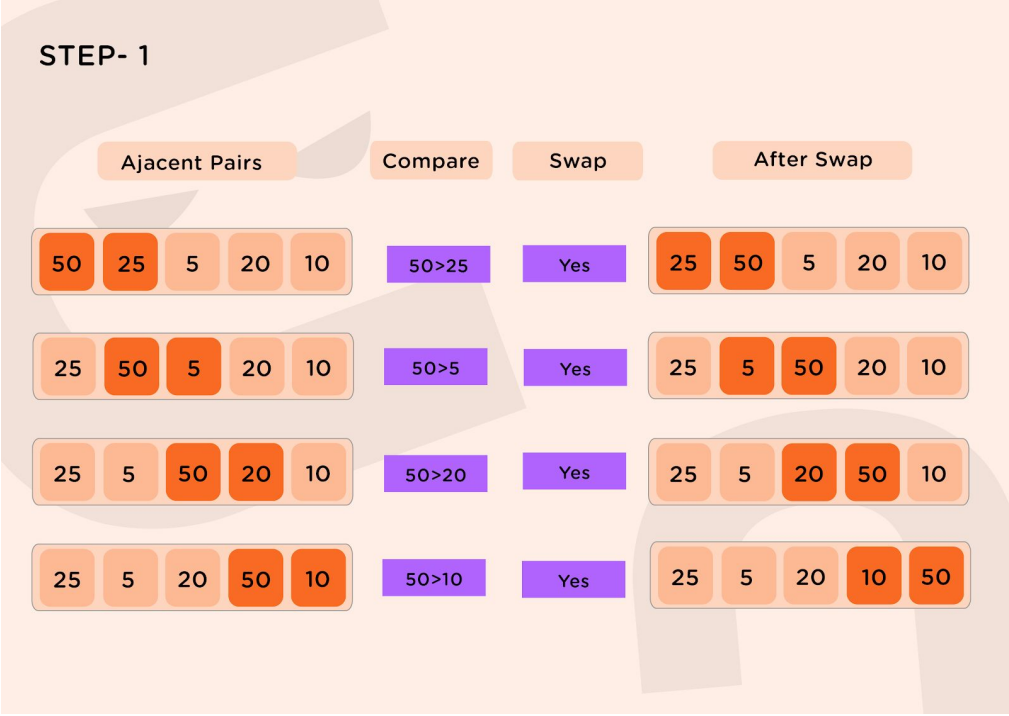
Bubble sort:

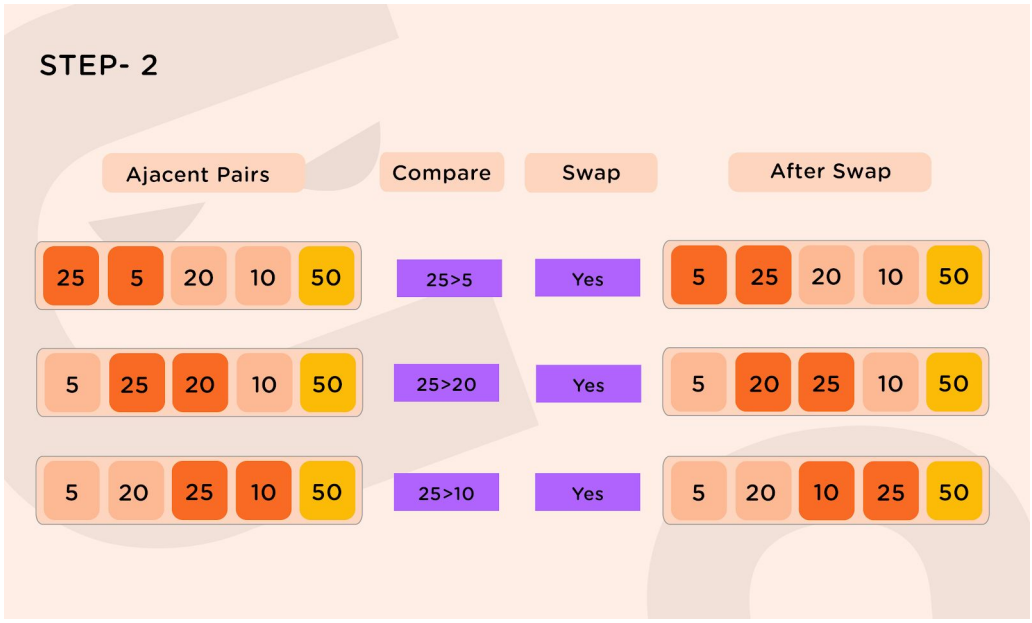
In selection sort the elements from the start get placed at the correct position first and then the further elements, but in the bubble sort, the elements start to place correctly from the end.

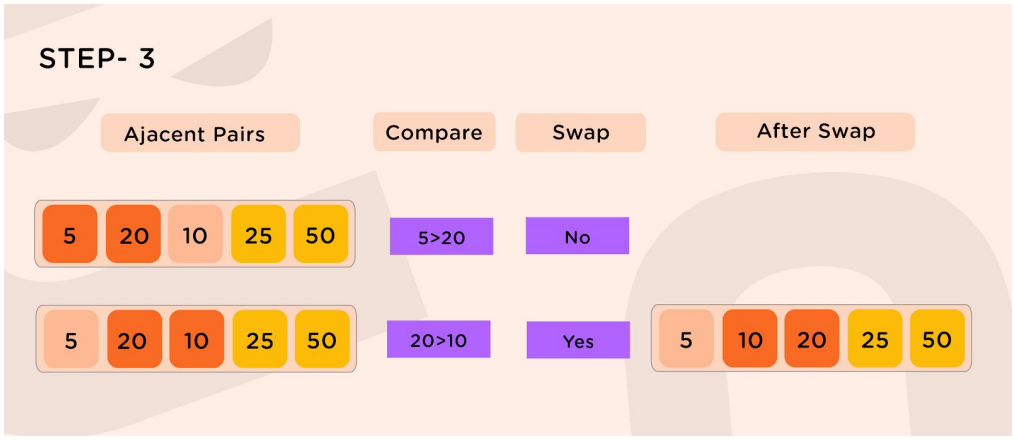
In this technique, we just compare the two adjacent elements of the array and then sort them manually by swapping if not sorted. Similarly, we will compare the next two elements (one from the previous position and the corresponding next) of the array and sort them manually. This way the elements from the last get placed in their correct position. This is the difference between selection sort and bubble sort. Consider the following depicted array as an example. You want to sort this array in increasing order.

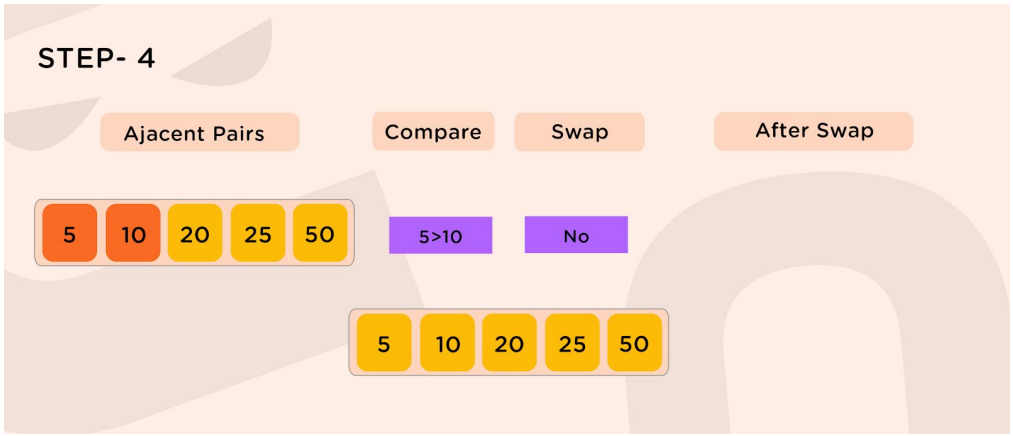
50	25	5	20	10

Following is the pictorial diagram for better explanation of how it works:




STEP- 2
Ajacent Pairs
Compare
Swap
After Swap
25 5 20 10 50
25>5
Yes
5 25 20 10 50
5 25 20 10 50
25>20
Yes
5 20 25 10 50
5 20 25 10 50
25>10
Yes
5 20 10 25 50


STEP- 3
Ajacent Pairs
Compare
Swap
After Swap
5 20 10 25 50
5>20
No
5 20 10 25 50
20>10
Yes
5 10 20 25 50


STEP- 4
Ajacent Pairs
Compare
Swap
After Swap
5 10 20 25 50
5>10
No
5 10 20 25 50

Implementation of bubble sort

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

void bubbleSort(int array[], int size) {

  // run loops two times: one for walking throught the array
  // and the other for comparison
  for (int step = 0; step < size - 1; ++step) {
    for (int i = 0; i < size - step - 1; ++i) {

      // To sort in descending order, change > to < in this line.
      if (array[i] > array[i + 1]) {

        // swap if greater is at the rear position
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// function to print the array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    cout << "  " << array[i];
  }
  cout << endl;
}

// driver code
int main() {
  int data[] = {-2, 45, 0, 11, -9};
  int size = sizeof(data) / sizeof(data[0]);
  bubbleSort(data, size);
  cout << "Sorted Array in Ascending Order:\n";
  printArray(data, size);
}
```

Insertion sort:

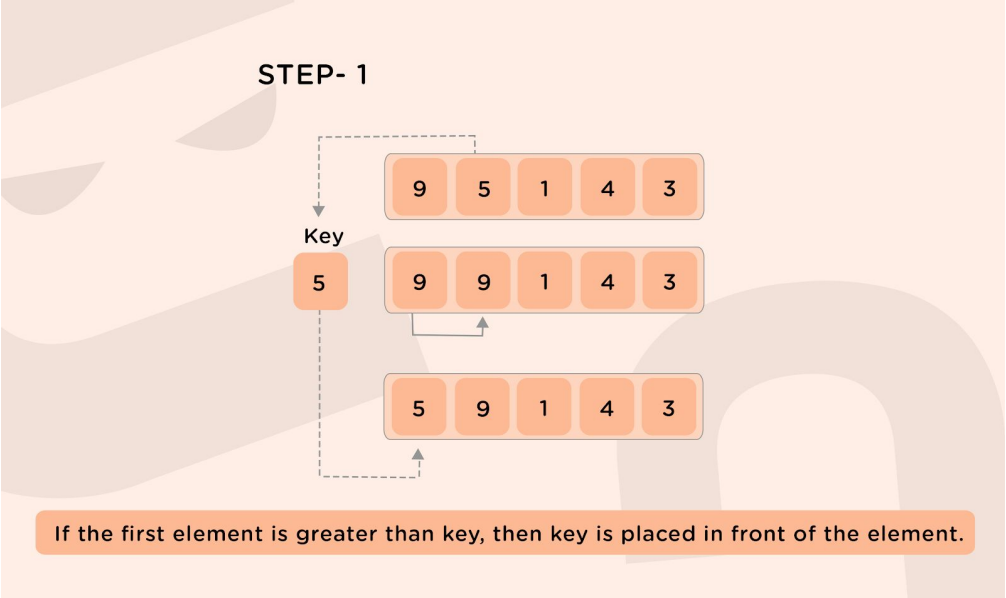
Insertion Sort works similar to how we sort a hand of playing cards.
Imagine that you are playing a card game. You're holding the cards in your hand, and these cards are sorted. The dealer hands you exactly one new card. You have to put it into the correct place so that the cards you're holding are still sorted. In selection sort, each element that you add to the sorted subarray is no smaller than the elements already in the sorted subarray. But in our card example, the new card could be smaller than some of the cards you're already holding, and so you go down the line, comparing the new card against each card in your hand, until you find the place to put it. You insert the new card in the right place, and once again, your hand holds fully sorted cards. Then the dealer gives you another card, and you repeat the same procedure. Then another card, and another card, and so on, until the dealer stops giving you cards. This is the idea behind **insertion sort**.
Loop over positions in the array, starting with index 1. Each new position is like the new card handed to you by the dealer, and you need to insert it into the correct place in the sorted subarray to the left of that position.

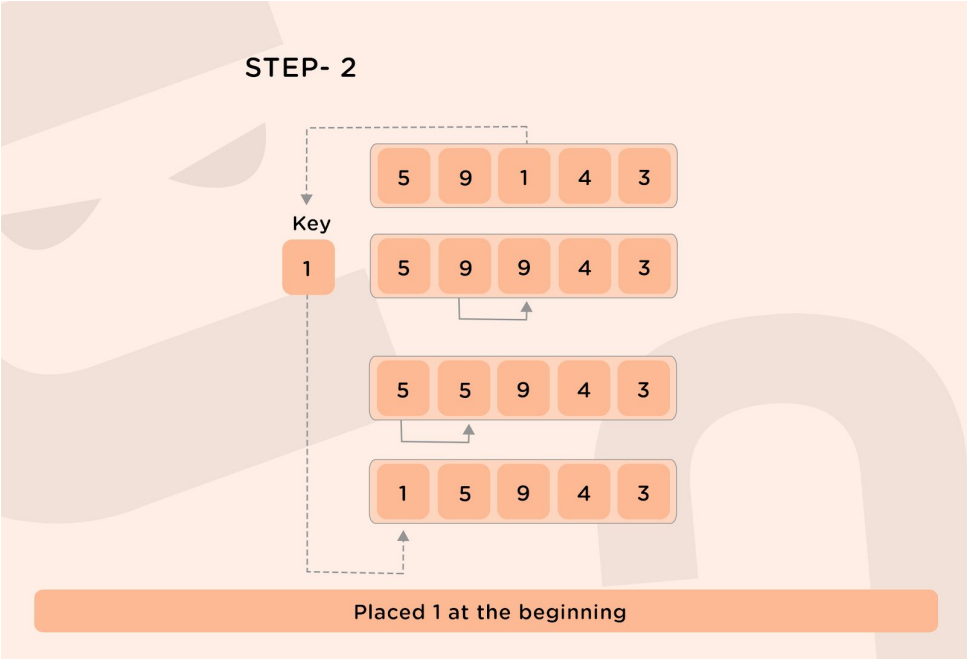
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

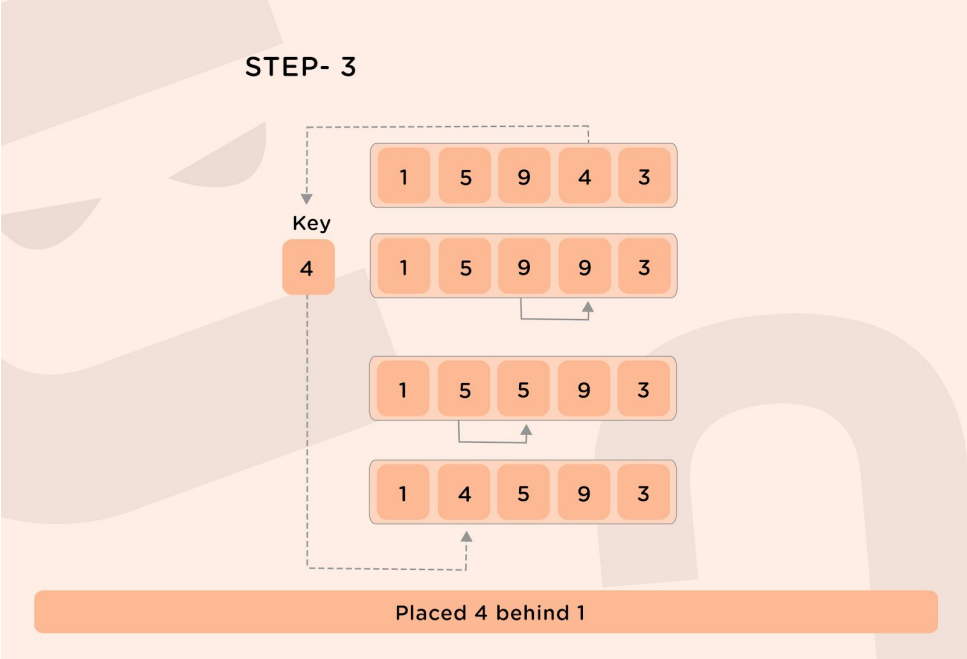
Consider the following depicted array as an example. You want to sort this array in increasing order.

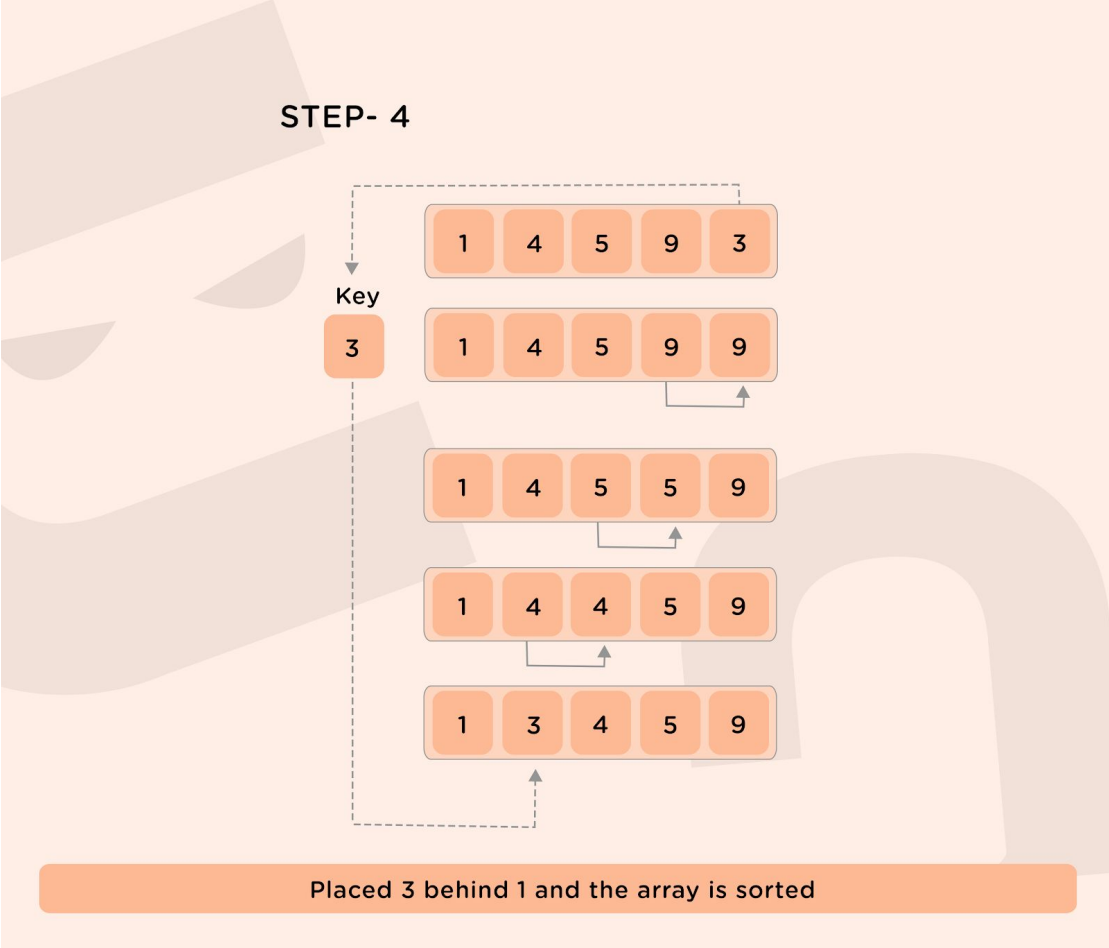
9	5	4	1	3

Following is the pictorial diagram for better explanation of how it works:




STEP- 2
Key
1
5 9 1 4 3
5 9 9 4 3
5 5 9 4 3
1 5 9 4 3
Placed 1 at the beginning


STEP- 3
Key
4
1 5 9 4 3
1 5 9 9 3
1 5 5 9 3
1 4 5 9 3
Placed 4 behind 1



Implementation of insertion sort

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

void display(int array[], int size) {
  for(int i = 0; i<size; i++)
    cout << array[i] << " ";
  cout << endl;
}
void insertionSort(int array[], int size) {
  int key, j;
  for(int i = 1; i<size; i++) {
    key = array[i];                    //take value
    j = i;
    while(j > 0 && array[j-1]>key) {
```

```
      array[j] = array[j-1];
      j--;
    }
    array[j] = key;                    //insert in right place
  }
}
int main() {
  int n;
  cout << "Enter the number of elements: ";
  cin >> n;
  int arr[n];                          //create an array with given number of elements
  cout << "Enter elements:" << endl;
  for(int i = 0; i<n; i++) {
    cin >> arr[i];
  }

  cout << "Array before Sorting: ";
  display(arr, n);

  insertionSort(arr, n);
  cout << "Array after Sorting: ";
  display(arr, n);
}
```

Now, practice different questions to get more familiar with the concepts. In the advanced course, you will study more types of sorting techniques.

Practice

- For binary search:

  https://www.hackerearth.com/practice/algorithms/searching/binary-search/practice-problems/

- For sorting:

  https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial/

  (In this link, you will find the questions for bubble sort, checkout the left tab, from there you can switch to other sorting techniques.)

Character arrays and 2D arrays

Strings

Till now, we have seen how to work with single characters but not with a sequence of characters. For storing a character sequence, we use strings. Strings are the form of 1D character arrays which are terminated by null character. Null character is a special character with a symbol '\0'.

Declaration syntax:

```
string  variableName;
```

string is the datatype in C++ to store continuous characters that terminates on encountering a space or a newline.

For example:

Suppose the input is:

```
Hello! How are you?
```

If you try to take input using string datatype over the given example, then it will only store **Hello!** in the form of an array.

To store the full sentence, we would need a total of four-string variables (though there are other methods to handle this (getline), we will soon read about them.)

Note: To use string datatype, don't forget to include the header file:

```
#include <cstring>
```

Character arrays

Declaration syntax:

```
char Name_of_array[Size_of_array];
```

All the ways to use them are the same as that of the integer array, just one difference is there. In case of an integer array, suppose we want to take 5 elements as input, an array of size 5 will be good for this.

In case of character arrays, if the length of the input is 5 then we would have to create an array of size at least 6, meaning **character arrays require one extra space in the memory from the given size.** This is because the character arrays store a NULL character at the last of the given input size as the mark of termination in the memory.

Now talking about the memory consumption, as each character requires 4 bytes in memory, the character array will require 4 multiplied by the character array size.

Also to take the character array as the input, you don't need to necessarily run the loop for each element. You can directly do the same as follows:

```
cin >> Name_of_array;
```

If we take above the example and run this statement over that and let the name of the array is arr and the size of array is at least 7, then the memory representation is as follows:

0	1	2	3	4	5	6												
'H'	**'e'**	**'l'**	**'l'**	**'o'**	**'!'**	**'\0'**												

Here, also the cin statement terminates taking input if it encounters any of the following:

- Space
- Tab
- \n

At the last you can notice some special character is placed, that is the NULL character about which we were talking earlier.

In the same way, you can directly use a cout statement to print the character array instead of running the loop.

Till now we haven't solved the "string with spaces" problem. Let's check that also...

Getline function:

In C++, we have another function named as **cin.getline()** which takes 3 arguments:

- Name of the string or character array name
- Length of the string
- Delimiter (optional argument)

Syntax:

```
cin.getline(string_name, length_of_string, delimiter);
```

Note: This function breaks at new line. This also initializes the last position to NULL.

Delimiter denotes the ending of each string. Generally, it is '\n' by default.

Let's look at some examples:

Suppose the input we want to take is: **Hello how are you?**

If the input commands are:

- **cin.getline(input, 100);**: this will input all the characters of input
- **cin.getline(input, 3);**: this will input only the first 3 characters of input and discard the rest instead of showing error.

In-built functions for character array:

- **strlen(string_name)**: To calculate the length of the string
- **strcmp(string1, string2)**: Comparison of two strings returns an integer value. If it returns 0 means equal strings. If it returns a positive value, it means that string2 is greater than string1 and if it returns a negative value, it means that string1 is greater than string2.

- **strcpy(destination_string, source_string) :** It copies the source_string to destination_string.
- **strncpy(destination_string, source_string, number_of_characters_to_copy):** it copies only a specified number of characters from source_string to destination_string.

2D arrays

Combination of many 1D arrays forms a 2D array.

Syntax to declare a 2D array:

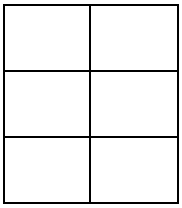
```
datatype array_name[rows][columns];
```

where rows imply the number of rows needed for the array and column implies the number of columns needed.

For example: If we want to declare an integer array of size 2x3, then:

```
int arr[2][3];
```

It looks like:



In this array you can store the values as required. Suppose in the above array you want to store 3 at every index, you can do so using the following code:

```
for (int i = 0; i < 2; i++)
{
        for (int j = 0; j < 3; j++)
        {
                arr[i][j] = 3;
        }
}
```

where, arr[i][j] invokes the element of the ith row and jth column.

How are 2D arrays stored in the memory?

They are actually stored as a 1D array of size (number_of_rows * number_of_columns).

Like if you have an array of size 5 x 5, so in the memory, a 1D array is created of size 25 and if you want to get the value of the element (2,1) in this array, it will invoke (2*5 + 1 = 11)th position in the array. We don't need to take care of this calculation, these are done by the compiler itself.

If we want to pass this array to a function, we can simply pass the name of the array as we did in the case of 1D arrays. But in the function definition, we can leave the first dimension empty, though the second dimension always needs to be specified.

Syntax for function call:

```
int fun (int arr[ ][number_of_columns], int n, int m) {
.................
}
```

Let's move to some questions now:

Practice problems:

Arrays and strings:

https://www.hackerearth.com/challenges/competitive/code-monk-array-strings/problems/

2D arrays:

https://www.hackerrank.com/challenges/2d-array/problem
https://www.techgig.com/practice/data-structure/two-dimensional-arrays

Rest there are many questions available for practice on codezen too. You can try them also...

C++ Foundation with Data Structures

Topic : Pointers

Address of Operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand
sign (&), known as *address-of operator*. For example:

```
cout << (&var) << endl;
```

This would print address of variable *var*; by preceding the name of the variable var with the *address-of operator* (&), we are no longer printing the content of the variable itself, but its address.

What are Pointers?

Pointers are one of the most important aspects of C++. Pointers are another type of variables in CPP and these variables store addresses of other variables.

While creating a pointer variable, we need to mention the type of data whose address is stored in the pointer. e.g. in order to create a pointer which stores address of an integer, we need to write:

```
int* p;
```

This means that p will contain address of an integer. So, if a pointer is going to store address of datatype X, it will be declared like this:

```
X* p;
```

Now let's say we have an integer i & an integer pointer p, we will use addressof(&) operator in order to put address of i in p. Address of operator:& as studied abpve is a unary operator which returns address of a variable. e.g. &i will give us address of variable i. Here is the code to put address of i in p.

```
int i = 10;
int* p;
p = &i;
```

Dereference Operator

As just seen, a variable which stores the address of another variable is called a *pointer*. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (*). The operator itself can be read as "value pointed to by".

Therefore, following with the values of the previous example, consider the following statement:

int a = *p;

So in this assignment we are assigning value pointed to by pointer p(i.e. value of to int i) to int variable a.

The reference and dereference operators are thus complementary:

- & is the *address-of operator*, and can be read simply as "address of"
- * is the *dereference operator*, and can be read as "value pointed to by"

Thus, they have sort of opposite meanings: An address obtained with & can be dereferenced with *.

Note that the asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the *dereference operator* seen above, but which is also written with an asterisk (*). They are simply two different things represented with the same sign.

Following

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

int main ()
{
 int firstvalue = 5, secondvalue = 15;
char thirdvalue = ‘a’;
 int * p1, * p2;
char *p3;
```

```
firstvalue is 10
secondvalue is
20
thirdvalue is b
```

```
 p1 = &firstvalue;  // p1 = address of firstvalue
 p2 = &secondvalue; // p2 = address of secondvalue
 p3 = &thirdvalue;    // p3 = address of thirdvalue
 *p1 = 10;         // value pointed to by p1 = 10
 *p2 = *p1;        // value pointed to by p2 = value
pointed to by p1
 p1 = p2;          // p1 = p2 (value of pointer is copied)
 *p1 = 20;         // value pointed to by p1 = 20
*p3 = ‘b’   // value pointed to by p3 = ‘b ’

 cout << "firstvalue is " << firstvalue << '\n';
 cout << "secondvalue is " << secondvalue << '\n';
 cout << "thirdvalue is " << thirdvalue << '\n';
 return 0;
}
```

Note: While solving pointers question, you should use pen and paper and draw things to get better idea.

Null Pointer

Consider the following statement –

int *p;

Here we have created a pointer variable that contains garbage value. In order to dereference the pointer, we will try reading out the value at the garbage stored in the pointer. This will lead to unexpected results or segmentation faults. Hence we should never leave a pointer uninitialized and instead initialize it to NULL, so as to avoid unexpected behavior.

int *p = NULL; // NULL is a constant with a value 0
int *q = 0;   // Same as above

So now if we try to dereference the pointer we will get segmentation fault as 0 is a reserved memory address.

Pointer Arithmetic

Arithmetic operations on pointers behave differently than they do on simple data types we studied earlier. Only addition and subtraction operations are

allowed; the others aren't allowed on pointers. But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.

For example: char always has a size of 1 byte, short is generally larger than that, and int and long are even larger; the exact size of these being dependent on the system. For example, let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

Suppose now that we define three pointers in this compiler:

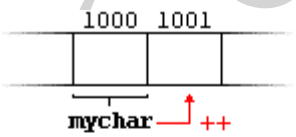
```
char *mychar;
short *myshort;
long *mylong;
```

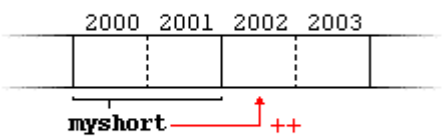
and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.

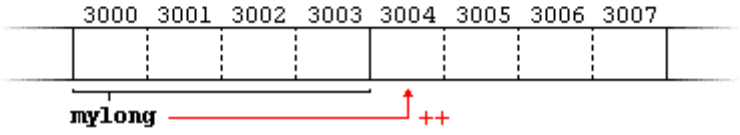
Therefore, if we write:

```
++mychar;
++myshort;
++mylong;
```

mychar, as one would expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.







This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we wrote:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

Pointer and Arrays

Pointers and arrays are intricately linked. An Array is actually a pointer that points to the first element of the array. Because the array variable is a pointer, you can dereference it, which returns array element 0.

Consider the following code –

```
int a[] = {1,2,3,4,5};
int *b = &a[0];
cout << b << endl;
cout << a << endl;
cout << *b << endl; // This will print 1
```

Both b and a will print same same address as they are referring to first element of the array.
Also in arrays - *a[i]* is same as **(a + i).*

Consider an example for the same–

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

int main(){

  int a[5] = {1,2,3,4,5};
  cout << *(a + 2) << endl;

}
```

Output:

3

Differences between arrays and pointers:

1. **the sizeof operator:**
   sizeof(array) returns the amount of memory used by all elements in array whereas sizeof(pointer) only returns the amount of memory used by the pointer variable itself.

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

int main(){

  int a[5] = {1,2,3,4,5};
  int *b = &a[0];
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
}
```

   Output :

```
20
8  // Size of pointer is compiler dependent. Here it is 8.
```

2. **the & operator**
   In the example above &a is an alias for &a[0] and returns the address of the first element in array
   &b returns the address of pointer.

3. **Pointer variable can be assigned a value whereas array variable cannot be.**

```
int a[10];
int *p;
p=a;        //legal
a=p;        //illegal
```

4. **Arithmetic on pointer variable is allowed, but not allowed on array variable.**

```
p++;        //Legal
a++;        //illegal
```

Double Pointer

As we know by now that pointers are variables that store address of other variables, so we can create variables that store address of pointer itself i.e. a pointer to a pointer. Let's see how can we create one.

```
int a = 10;
int *p = &a;
int **q = &p;
```

Here q is a pointer to a pointer i.e. a double pointer,as indicated by **.

Consider the following code for better understanding –

```
#include<iostream>
using namespace std;
int main(){
 int a = 10;
 int *p = &a;
 int **q = &p;

// Next three statements will print same value i.e. address of a
 cout << &a << endl;
 cout << p << endl;
 cout << *q << endl;

// Next two statements will print same value i.e. address of p
 cout << &p << endl;
 cout << q << endl;
```

```
// Next two statements will print same value i.e. value of a which is 10
  cout << a << endl;
  cout << *p << endl;
  cout << **q << endl;
}
```

Void Pointer

A void pointer is a generic pointer, it has no associated type with it. A void pointer can hold address of any type and can be typcasted to any type. Void pointer is declared normally the way we do for pointers.

```
void *ptr;
```

This statement will create a void pointer.

Example:

```
void *v;
int *i;
int ivar;
char chvar;
float fvar;
v = &ivar; // valid
v = &chvar; //valid
v = &fvar; // valid
i = &ivar; //valid
i = &chvar; //invalid
i = &fvar; //invalid
```

Thus we can use void pointer to store address of any variable.

Dynamic Allocation

Address typecasting

While declaring a pointer, why can't we just write like this:

```
int a = 5;
pointer p = &a;
```

Why do we have a complicated syntax like:

```
int a = 5;
int *p = &a;
```

It's generally because we need to specify that, when we invoke a particular pointer at any point, then how will the compiler know what type of value a pointer has stored and while invoking/transferring data, how much space needs to be allotted to it.

That is why while declaring a pointer, we start with the datatype and then assign the name to the pointer. That datatype specifies what type of value we are storing in the pointer.

The term **typecasting** means assigning one type of data to another type like storing an integer value to a char data type. For example:

```
int x= 65;
char c = x;
```

When you will check the value stored in variable 'c', it will print the ASCII value stored at that integer variable i.e., 'x' and will print 'A' (whose ASCII value is 65).

This type of typecasting done above is known as **implicit typecasting** as the compiler itself interprets the conversion of integer value to ASCII character value.

Now consider the example below:

```
int i = 65;
int *p = &i;
char *pc = (char*) p;
```

You can see that in the third line, we can't directly do like this:

char *pc = p;

This will give an error as we are trying to store a integer-type pointer value into a character-type value. To remove the error we have to type-cast by ourselves by providing (char*) on the right hand side as done in the code above. This type of typecasting is known as **explicit typecasting**.

Reference and pass by Reference

As you are familiar, that (&) symbol is known as a reference operator which means 'Address of operator'.

This operator is used to copy the values of any variable along with the guarantee that the reflected changes will also be visible in the copied variable.

For example:

```
int a = 5;
int b = a;
a++;
cout<<b<<endl;
```

Output:

```
5
```

Means that only the value is copied and when the value of the variable is increased by one, then the changes are not reflected in variable 'b'.

Now, look at the following piece of code:

```
int a = 5;
int &b = a;
a++;
cout<<b<<endl;
```

Here, (&b=a) means that now variables b and a are pointing to the same address and making changes in any of them gets reflected to both of the variables.

The above code outputs: 6

This concept of referencing the variables is useful in those cases where we want to update the value passed to the function. When we normally pass a value to a variable then a copy of those is created in the system and the original values remain unchanged. But by passing the reference, the changes are also reflected in the original variables as there is no extra copy created. The first type of argument passing is known as **pass by value** and the later one is known as **pass by reference.**

Syntax for pass by reference:

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

void fun(int &a) {
    a++;
}

int main() {
    int a = 5;
    fun(a);
    cout << a << endl;
}
```

Output:

```
6
```

Advantages of Pass by reference:

- Reduction in memory storage.
- Changes can be reflected easily.

Dynamic memory allocation

In the array, it is always advised that while declaring an array, we should always provide the size that is a constant value and not a variable in order to prevent Runtime error. Runtime error can happen if the variable contains a garbage value or some type of undefined value. Generally, we have two types of memory in our systems:

- **Stack memory:** It has a fixed value of size for which an array could be declared in the contiguous form.
- **Heap memory:** It is the memory where the array declared is not stored in a contiguous way. Suppose, if we want to declare an array of size 100000, but we do not have the contiguous space in the memory of the same size, then we will be declaring the array using heap memory as it necessarily don't require a contiguous allocation, it will allot the space wherever it gets and links all the memory blocks together.

  Note: Global variables are stored using heap memory.

The memory declaration using heap memory is known as **dynamic memory allocation** while the one using stack is known as **compile-time memory allocation**. So, how can we access heap memory?

The syntax is as follows:

```
int *arr = new int[size_of_array];
```

For simply allocating variables dynamically, use this:

```
int *var = new int;
```

You can see here, we are using a keyword **new** which is used to declare the dynamic memory in C++. There are other ways too but this one is the most efficient.

Note: Stack memory releases itself as the scope of the variable gets over, but in case of heap memory, it needs to be released manually at last otherwise the memory gets accumulated and in the end leads to memory leakage.

Syntax for releasing an array:

```
delete [ ] arr;
```

Syntax for releasing the memory of the variable:

```
delete var;
```

The keyword **delete** is used to clear the heap memory.

Now moving on how to create 2D arrays in the dynamic memory allocation...

Dynamic allocation of 2D arrays

To create a 2D array in the heap memory, we use the following syntax:

Suppose we want to create a 2D array of size n x m, with the name of the array as 'arr':

```
int **arr = new int*[n];
for (int i=0; i < n; i++) {
        arr[i] = new int[m];
}
```

You can see that first-of-all we have declared a pointer of pointers of size n and then at each pointer while traversing we allocated the array of size m using *new* keyword.

Similarly, we can release this memory, using the following syntax:

```
for (int i=0; i < n; i++) {
     delete [ ] arr[i];
}
delete [ ] arr;
```

First-of-all, we have cleared the memory allocated to each of the n pointers and then finally deleted those n pointers also.

This way, using *delete* keyword, we can release the memory of the 2D arrays.

Macros and global variable

Suppose in your code you are using the value pi(3.14) many times, instead of always writing 3.14 everywhere in the code, what we can do is define this value to some variable say pi in the beginning of the program and now everytime we need the value 3.14, we just need to write pi.

What we can do is that we can create a global variable with the value 3.14 and then use it anywhere as desired. There is one issue in using global variables and that is if someone changes the value of pi in our code at any point of time in any of the functions, then we could lose that original value.

To avoid such a situation, what we do is use macros. Syntax for using it:

```
#define pi 3.14
```

This is done after declaring the header files in the beginning itself so that this value can be used in the later encountered functions. Here, what we are actually doing is that we are declaring the value of the pi as 3.14 using the macros (#) define which basically locks the value and makes it unchangeable.

Another advantage is it prevents extra storage in the memory for declaring a new variable as by using macros we have specified to the compiler that the value we are using is a part of the compiler code, hence no need to create any extra memory.

Now, discussing about the advantages of the global variable:

- When we want to use the same variable with modified values in each of them, then we can use global variables as the changes done in one function are visible in all the other.
- It saves time for passing the values by reference in the functions.

Due to accidental changes, this method is not preferred much.

Inline and default arguments

Disadvantages of using a normal function:

- Uses time as it pauses some portion of code to get complete unless we are done with it we can't move forward.
- Creates a copy of variables each time while calling a function.

To prevent this what we can do is create inline functions. Inline functions replace the function call with its definition of when invoked.

Syntax:

```
inline fun() {
  ...
}
```

Just write the keyword inline before the function call.

When it is invoked like this:

```
fun();
```

Inside any other function, then what happens is it gets replaced with the function declaration hence preventing the pausing of the code at any point of time.

Disadvantages of using inline functions:

- Code becomes bulky.
- Time taken to copy code can be huge.

Now moving on to the use of default arguments...

Actually, sometimes we are unsure about any value(s) to be passed into the function as an argument but in the further calculations we need to use them as it is necessary to use them either by defined value or through some default value. This purpose is served by default arguments in C++. Using these we can specify a value to any variable in the function declaration to some default value that could be used if no value is passed to it by the function call.

Note: Be careful about using these arguments in the function call, default arguments can only be declared as the rightmost set of parameters.

For example: To calculate the sum of numbers, we are unsure that if we want to find the sum of two or three numbers, then:

```
int sum(int a, int b, int c = 0) {        // here, c is the default argument
    return a + b + c;
}

int main() {
    int a = 2, b = 3, c = 4;
    cout << sum(a, b) << endl;        // here, c will automatically be taken as 0
    cout << sum(a, b, c) << endl;     // as the value of c is provided, the value of c will
be 4
}
```

Output:

```
5
9
```

Constant variables

Now moving on to the constant variables which are identified by **const** keyword in C++. As the name suggests, if we declare any keyword as constant, then we can't change it's value throughout the program.

Note: The constant variable needs to be assigned during initialization only else it will store garbage value in it which can't be changed further.

Syntax:

```
const datatype variable_name = value;
```

For example:

```
const int a = 5;
```

C++ Foundation with Data Structures

Topic: Recursion

Recursion

a. What is Recursion?

In previous lectures, we used iteration to solve problems. Now, we'll learn about recursion for solving problems which contain smaller sub-problems of the same kind.

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. By same nature it actually means that the approach that we use to solve the original problem can be used to solve smaller problems as well.

So in other words in recursion a function calls itself to solve smaller problems. Recursion is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts and the code is also shorter and easier to understand.

b. How Does Recursion Work?

We can define the steps of a recursive solution as follows:

1. **Base Case:**
   A recursive function must have a terminating condition at which the function will stop calling itself. Such a condition is known as a base case.
2. **Recursive Call:**
   The recursive function will recursively invoke itself on the smaller version of problem. We need to be careful while writing this step as it is important to correctly figure out what your smaller problem is on whose solution the original problem's solution depends.
3. **Small Calculation:**
   Generally we perform a some calculation step in each recursive call. We can perform this calculation step before or after the recursive call depending upon the nature of the problem.

It is important to note here that recursion uses stack to store the recursive calls. So, to avoid memory overflow problem, we should define a recursive solution with minimum possible number of recursive calls such that the base condition is achieved before the recursion stack starts overflowing on getting completely filled.

Now, let us look at an example to calculate factorial of a number using recursion.

Example Code 1:

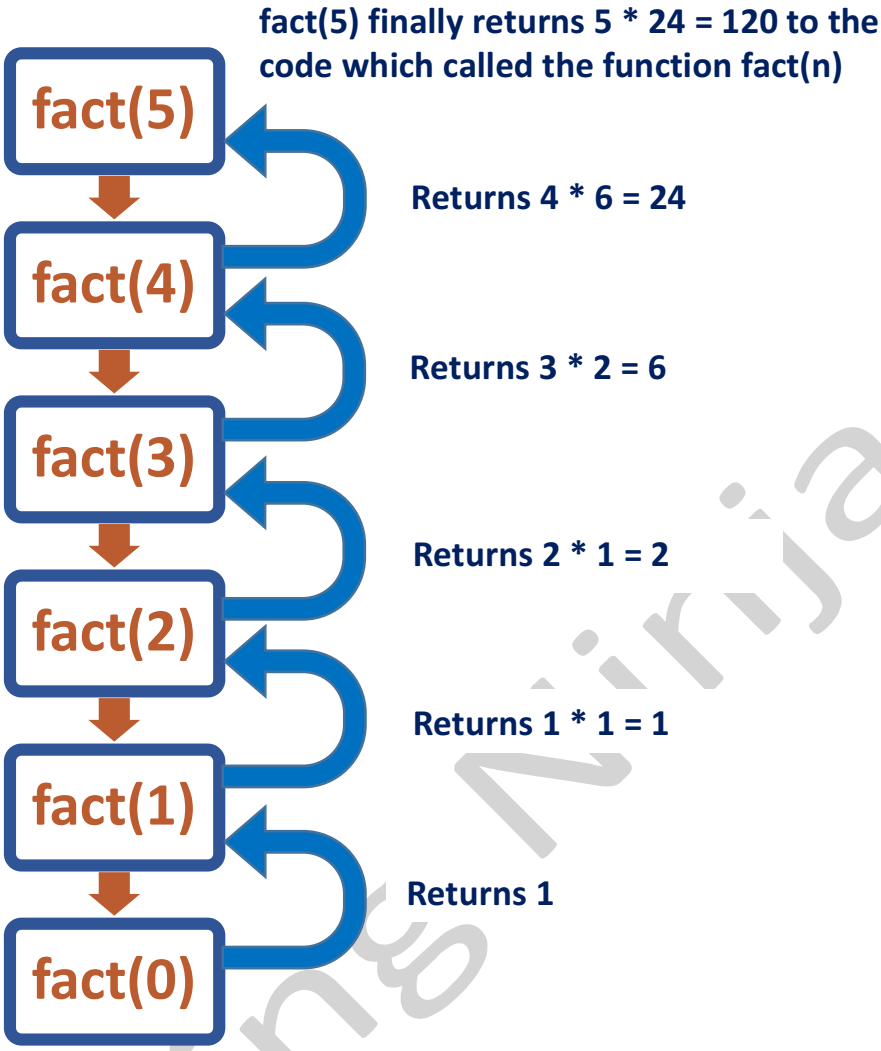
```
#include<iostream>
using namespace std;

int fact(int n)
{
   if(n==0)                          //Base Case
   {
      return 1;
   }
   return n * fact(n-1);             //Recursive call with small calculation
}

int main()
{
   int num;
   cin>>num;
   cout<<fact(num);
   return 0;
}
```

Output:

```
120          //For num=5
```

Explanation:

Here, we called factorial function recursively till number became 0. Then, the statements below the recursive call statement were executed. We can visualize the recursion tree for this function, where let n=5, as follows:



We are calculating the factorial of n=5 here. We can infer that the function recursively calls fact(n) till n becomes 0, which is the base case here. In the base case, we returned the value 1. Then, the statements after the recursive calls were executed which returned n*fact(n-1) for each call. Finally, fact(5) returned the answer 120 to main() from where we had invoked the fact() function.

Now, let us look at another example to find n^{th} Fibonacci number . In Fibonacci series to calculate nth Fibonacci number we can use the formula F(n) = F(n – 1) + F(n – 2) i.e. nth Fibonacci term is equal to sum of n-1 and n-2 Fibonacci terms. So let's use this to write recursive code for nth Fibonacci number.

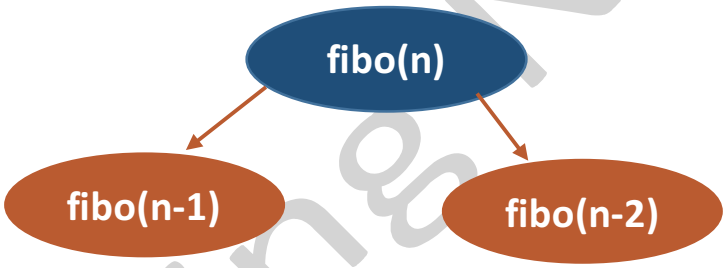
Example Code 2:

// Recursive function:

```
int fibo(int n) {
   if(n==0 || n==1) {          //Base Case
      return n;
   }
   int a = fibo(n-1);          //Recursive call
   int b = fibo(n-2);          //Recursive call
   return a+b;                 //Small Calculation and return statement
}
```

Explanation:
As we are aware of the Fibonacci Series (0, 1, 1, 2, 3, 5, 8,… and so on), let us assume that the index starts from 0, so, 5^{th} Fibonacci number will correspond to 5; 6^{th} Fibonacci number will correspond to 8; and so on.

Here, in recursive Fibonacci function, we have made two recursive calls which are depicted as follows:



Note: One thing that we should be clear about is that both recursive calls don't happen simultaneously. First fibo(n-1) is called, and only after we have its result and store it in "a" we move to next statement to calculate fibo(n – 2).

It is interesting to note here that the concept of recursion is based on the mathematical concept of **PMI** (Principle of Mathematical Induction). When we use PMI to prove a theorem, we have to show that the base case (usually for x=0 or x=1) is true and, the induction hypothesis for case x=k is true must imply that case x=k+1 is also true. We can now understand how the steps which we followed in recursion are based on the induction steps, as in recursion also, we have a base case while the assumption corresponds to the recursive call.

Recursion 2

In this lecture, we are going to understand how to solve different kinds of problems using recursion. It is strictly advised to complete the assignments for the earlier topics, if due.

Recursion and Strings

Here, we are going to discuss the different problems that can be solved using recursion on strings:

- Finding the length of the string

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

int length(char s[]) {
        if (s[0] == '\0') {                          // Base case
                return 0;
        }
        int smallStringLength = length(s + 1);       // Recursive call
        return 1 + smallStringLength;                // Small calculation
}


int main() {
        char str[100];
        cin >> str;

        int l = length(str);
        cout << l << endl;
}
```

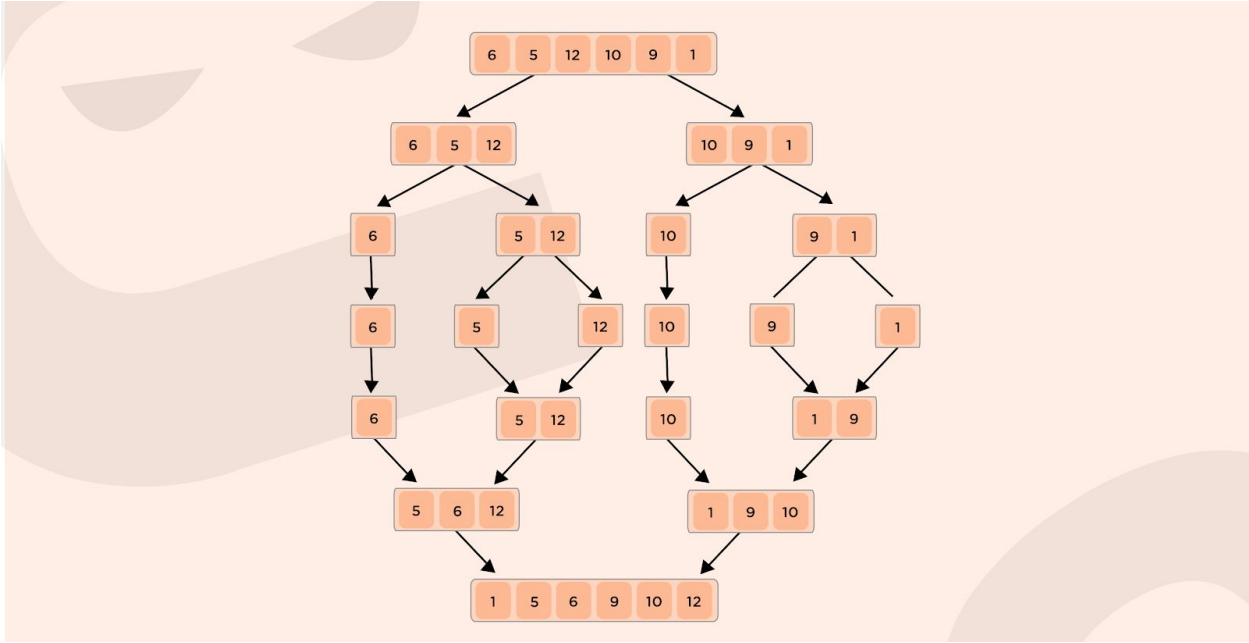
- Remove X from a given string

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

void removeX(char s[]) {
        if (s[0] == '\0') {                          // Base case
                return;
        }

        if (s[0] != 'x') {                           // Small calculation
                removeX(s + 1);                      // Recursive call
        } else {
                int i = 1;
                for (; s[i] != '\0'; i++) {
                        s[i - 1] = s[i];
                }
                s[i - 1] = s[i];
                removeX(s);
        }
}

int main() {
        char str[100];
        cin >> str;
        removeX(str);
        cout << str << endl;
}
```

Sorting Technique

Merge Sort

Consider the example below that shows the working of merge sort over the given array and also how it **divides and conquers** the array.



Let's now look at the algorithm associated with it...

Algorithm for Merge Sort:

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists, keeping the new list sorted too.

Step 1 − If it is only one element in the list it is already sorted, return.

Step 2 − Divide the list recursively into two halves until it can no more be divided.

Step 3 − Merge the smaller lists into new list in sorted order.

Now, you can code it easily by following the above three steps.

It has just one disadvantage and it is that it's **not an in-place sorting** technique, which means it creates a copy of the array and then works on that copy.

Time Complexity : O(nlogn)

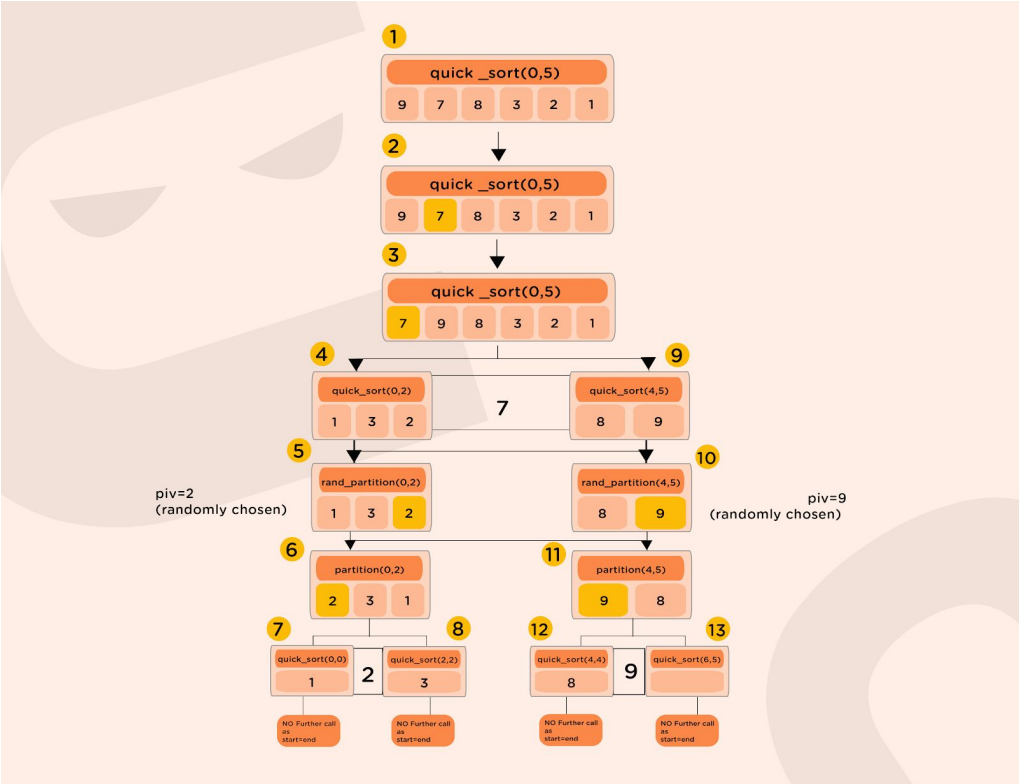
Extra Space Complexity : O(n)

Quick sort

Quick sort is based on the **divide-and-conquer approach** based on the idea of choosing one element as a pivot element and partitioning the array around it, such that: Left side of pivot contains all the elements that are less than the pivot element and Right side contains all elements greater than the pivot.

It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in an array results in an improved time complexity in most of the cases.

To get a better understanding of the approach consider the following example where a pictorial representation of the quick sort on an array is shown.



Algorithm for Quick Sort:

On the basis of Divide and Conquer approach, quicksort algorithm can be explained as:

- **Divide**

  The array is divided into subparts, taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right side.

- **Conquer**

  The left and right sub-parts are again partitioned by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.

- **Combine**

This part does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.

Advantage of quicksort over merge sort is that it does not require any extra space to sort the given array, therefore, it is an in-place sorting technique.

Time Complexity : O(nlogn)

Extra Space Complexity : O(1)

Strings

Let us think of string as a class, instead of a data type. We all already know that strings are the character arrays that ends with a NULL character itself.

Syntax for declaring a string:

```
string str;
```

Syntax for declaring a string dynamically:

```
string *str = new string;
```

To take input of the string, we can use the **getline()** function. Newline is the delimiter for the getline() function. Follow the syntax below:

```
getline(cin, str);        // where str is the name of the string
```

You can treat the string like any other character array as well. You can go to any specific index (indexing starts from 0), assign any string value by using equals to (=) operator, etc.

To concatenate two strings you can use '+' operator. For example, to concatenate two strings s1 and s2 and put it in another string str, syntax is as below:

```
string str = s1 + s2;
```

To calculate the length of the string you can use the in-built function (.length()) for that:

```
int len = str.length();
```

You can do the same using the following syntax also:

```
int len = str.size();
```

To extract a particular segment of the string, we can use .substr() function.

```
string s = str.substr(beginning_index, length_ahead_of_starting_index);
```

Here, if you omit the second argument, then automatically the function takes the complete string, after the specified starting index.

You can also find any particular substring in the given string using .find() function. It will return the starting index of the substring to be found in the given string.

```
int index = str.find(name_of_the_pattern_to_be_checked);
```

Note: .find() function finds the first occurring index of the given pattern.

These concepts would be much clearer after studying OOPs.

Let's practice some of the questions over the topics covered in this lecture...

Questions:

Visit the following links for practicing some questions...

- https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/practice-problems/
- https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/practice-problems/
- https://www.techiedelight.com/recursion-practice-problems-with-solutions (On this webpage, visit the subheading **String**)

C++ Foundation with Data Structures

Topic: Time Complexity

Introduction

An important question while programming is: How efficient is an algorithm or piece of code?

Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we are mostly concerned about CPU time.

Be careful to differentiate between:

1. Performance: how much time/memory/disk/etc. is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.

2. Complexity: how do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.
Complexity affects performance but not vice-versa.
The time required by a function/method is proportional to the number of "*basic operations*" that it performs.

Here are some examples of basic operations:

- one arithmetic operation (e.g. a+b / a*b)
- one assignment (e.g. int x = 5)
- one condition/test (e.g. x == 0)
- one input read (e.g. reading a variable from console)
- one output write (e.g. writing a variable on console)

Some functions/methods perform the same number of operations every time they are called.
For example, the size function/method of the string class always performs just one operation: return number of Items; the number of operations is independent of the size of the string. We say that functions/methods like this (that always perform a fixed number of basic operations) require constant time.

Other functions/methods may perform different numbers of operations, depending on the value of a parameter. For example, for the array implementation of the Vector/list(Java) class(vector/list classes are implemented similar to the dynamic class we have built), the remove function/method has to move over all of the items that were to the right of the item that was removed (to fill in the gap). The number of moves depends both on the position of the removed item and the number of items in the list. We call the important factors (the parameters and/or fields whose values affect the number of operations performed) the problem size or the input size.

When we consider the complexity of a function/method, we don't really care about the exact number of operations that are performed; instead, we care about how the number of operations relates to the problem size. If the problem size doubles, does the number of operations stay the same? double? increase in some other way? For constant-time functions/methods like the size function/method, doubling the problem size does not affect the number of operations (which stays the same).

Furthermore, we are usually interested in the worst case: what is the most operations that might be performed for a given problem size. For example, as discussed above, the remove function/method has to move all of the items that come after the removed item one place to the left in the array. In the worst case, all of the items in the array must be moved. Therefore, in the worst case, the time for remove is proportional to the number of items in the list, and we say that the worst-case time for remove is linear to the number of items in the array. For a linear-time function/method, if the problem size doubles, the number of operations also doubles.

Big-O Notation

We express complexity using big-O notation. For a problem of size N:
a constant-time function/method is "order 1": O(1)
a linear-time function/method is "order N": O(N)
a quadratic-time function/method is "order N squared": O(N^2)
Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time function/method will be faster than a linear-time

function/method, which will be faster than a quadratic-time function/method). See below for an example.
Formal definition:

A function T(N) is O(F(N)) if for some constant c and for all values of N greater than some value n_0:
T(N) <= c * F(N)
The idea is that T(N) is the exact complexity of a function/method or algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e. the actual time/space or whatever for a problem of size N will be no worse than F(N)). In practice, we want the smallest F(N) - the least upper bound on the actual complexity.
For example, consider T(N) = 3 * N^2 + 5. We can show that T(N) is O(N^2) by choosing c = 4 and n_0 = 2. This is because for all values of N greater than 2:

3 * N^2 + 5 <= 4 * N^2
T(N) is not O(N), because whatever constant c and value n0 you choose, I can always find a value of N greater than n_0 so that 3 * N^2 + 5 is greater than c * N.

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;
statement 2;
 ...
statement k;
```

The total time is found by adding the times for all statements:
total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

2. if-then-else statements

```
if (condition) {
   sequence of statements 1
}
else {
   sequence of statements 2
}
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities: max(time(sequence 1), time(sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

3. for loops

```
for (i = 0; i < N; i++) {
   sequence of statements
}
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

4. Nested loops

First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      sequence of statements
   }
}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M). In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
   for (j = i+1; j < N; j++) {
      sequence of statements
   }
}
```

Now we can't just multiply the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So let's think about how many iterations that inner loop has. That information is given in the following table:

Value of i	Number of iterations of inner loop
0	N
1	N-1
2	N-2
...	...
N-2	2
N-1	1

So we can see that the total number of times the sequence of statements executes is: N + N-1 + N-2 + ... + 3 + 2 + 1. the total is $O(N^2)$.

5. **Statements with function/method calls:**

When a statement involves a function/method call, the complexity of the statement includes the complexity of the function/method call. Assume that you know that function/method f takes constant time, and that function/method g takes time proportional to (linear in) the value of its parameter k. Then the statements below have the time complexities indicated.

```
f(k);  // O(1)
g(k);  // O(k)
```

When a loop is involved, the same rule applies. For example:

```
for (j = 0; j < N; j++) {
        g(N);
}
```

has complexity (N^2). The loop executes N times and each function/method call g(N) is complexity O(N).

Best-case and Average-case Complexity

Some functions/methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, consider the add function/method that adds an item to the end of the Vector/list. In the worst case (the array is full), that function/method requires time proportional to the number of items in the Vector/list (because it has to copy all of them into the new, larger array). However, when the array is not full, add will only have to copy one value into the array, so in that case its time is independent of the length of the Vector/list; i.e. constant time.

In general, we may want to consider the best and average time requirements of a function/method as well as its worst-case time requirements. Which is considered the most important will depend on several factors. For example, if a function/method is part of a time-critical system like one that controls an airplane, the worst-case times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average-case times for that computation are not relevant -- the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain).

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases.

Note that calculating the average-case time for a function/method can be tricky. You need to consider all possible values for the important factors, and whether they will be distributed evenly.

When do Constants Matter?

Recall that when we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter. However, this means that two algorithms can have the same big-O time

complexity, even though one is always faster than the other. For example, suppose algorithm 1 requires N^2 time, and algorithm 2 requires 10 * N^2 + N time. For both algorithms, the time is O(N^2), but algorithm 1 will always be faster than algorithm 2. In this case, the constants and low-order terms do matter in terms of which algorithm is actually faster.

However, it is important to note that constants do not matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles). Although an algorithm that requires N^2 time will always be faster than an algorithm that requires 10*N^2 time, for both algorithms, if the problem size doubles, the actual time will quadruple.

When two algorithms have different big-O time complexity, the constants and low-order terms only matter when the problem size is small. For example, even if there are large constants involved, a linear-time algorithm will always eventually be faster than a quadratic-time algorithm. This is illustrated in the following table, which shows the value of 100*N (a time that is linear in N) and the value of N^2/100 (a time that is quadratic in N) for some values of N. For values of N less than 104, the quadratic time is smaller than the linear time. However, for all values of N greater than 104, the linear time is smaller.

N	**100*N**	**N^2/100**
10^2	10^4	10^2
10^3	10^5	10^4
10^4	10^6	10^6
10^5	10^7	10^8
10^6	10^8	10^{10}
10^7	10^9	10^{12}

C++ Foundation with Data Structures

Topic : Object Oriented Programming - 1

What is Object Oriented Programming?

Object Oriented Programming could be best understood with help of an example. Consider a library management system. Using procedural programming, the problem will be viewed in terms of working happening in the library i.e., issuing of book, returning the book, adding new book etc. The OOP paradigm, however aims at the objects and their interface. Thus in OOP, library management problem will be viewed in terms of the objects involved. Objects are real world entities around which the system revolves. The objects involved are: librarian, book, member etc. Hence, the object-oriented approach views a problem in terms of objects involved rather than procedure for doing it.

Why use Object Oriented Programming?

In real life we deal with lot of objects such as people, car, account, etc. Hence, we need our software to be analogous to real-world objects. Real world objects have data-type properties such as name, age for people, model name for car and balance for account, etc. Moreover, real-world objects can also do certain things such as people talk, cars move, account accumulates, etc. We want our code to mimic the way these objects behave and interact. Hence, Object Oriented Programming allows the program to be closer to real world and thereby making it less complex. Also it makes the software reuse feasible and possible, for example, we don't want to define a student every time we use it, hence using OOP, we just create the blueprint for the student object and use it whenever required.

Classes and objects

The class is a single most important C++ feature that implements OOP concepts and ties them together. Classes are needed to represent real-world entities. A class is a way to bind the data describing an entity and its associated functions together. For instance, consider a ***user*** having characteristics ***username*** and ***password*** and some of its associated operations are ***sign up***, ***sign in*** and ***logout***.

Class is just a template, which declares and defines characteristics and behavior, hence we need to declare objects of the class for it to be usable. In other words class represents a group of similar objects.

Data members and member functions

Data members are the data-type properties that describe the characteristics of the class.

Member functions are the set of operations that may be applied to objects of that class, i.e., they represent the behavioral aspect of the object. They are usually referred as class interface.

Syntax for definition of a class :

```
class class_name {
        Access Modifier:
                Data members
                Member functions
};
```

The class body contains the declaration of members (data and functions).

Declaring objects of a class

We can declare objects of a class either statically or dynamically just as we declare variables of primitive data types.

Statically

Syntax for declaring objects of a class statically is:

```
class_name    object_name;
```

Example code :

```
class Student {
    public :
        int rollno;
        char name[20];
};
int main( ){
    Student s1;       // Declaration of object s1 of type Student
    Student s2;       // Declaration of object s2 of type Student
}
```

Dynamically

Syntax for declaring objects of a class dynamically is:

class_name *object_name = new class_name

Example code :

```
class Student {
    public :
        int rollno;
        char name[20];
};
int main( ){
    Student *s1 =new Student;   // Declaration of object of type Student
                                // dynamically
}
```

Access Modifiers

The class body contains the declaration of its members (data and functions). They are generally two types of members in a class: private and public, which are correspondingly grouped under two sections namely private: and public: .

The **private members** can be accessed only from within the class. These members are hidden from the outside world. Hence they can be used only by member functions of the class in which it is declared.

The **public members** can be accessed outside the class also. These can be directly accessed by any function, whether member function of the class or non-member function. These are basically the public interface of the class.
By default, members of a class are private if no access specifier is provided.

Example code :

```
class A {
    int x, y;
    int sqr(){
            return x * x;
    }
    public :
            int z;
            int twice(){
                    return 2 * y;
            }
```

```
        int test(int i){
                int q = sqr( );       // private function being
        invoked                                  // by member function
                return q + i;
        }
};
int main() {
    A obj;
    obj.z = 10;              // valid. z is a public member
    obj.x = 4;               // Invalid. x is a private member and hence can
                             // be   accessed only by member functions
                            // not  directly by using object
    int j = obj.twice( );   // valid. twice( ) is a public member function
    int k = obj.sqr( );     // Invalid. sqr() is a private member function
    int l = obj.test( );    // valid. test is a public member function
}
```

Getter and setters

The private members of the class are not accessible outside the class, although sometimes there is a necessity to provide access even to private members, in these cases we need to create functions called getters and setters. Getters are those functions that allow us to access data members of the object. However, these functions do not change the value of data members. These are also called accessor function.

Setters are the member functions that allow us to change the data members of an object. These are also called mutator function.

Example code :

```
class Student {
    int rollno;
    char name[20];
    float marks;
    char grade;
    public :
        int getRollno( ){
            return rollno;
        }
        int getMarks( ){
```

```
        return marks;
    }

    void setGrade( ){
        if (marks > 90) grade ='A';;
        else if (marks > 80) grade = 'B';
        else if (marks > 70) grade = 'C';
        else if (marks > 60) grade = 'D';
        else grade = 'E';
    }
};
```

getRollno( ) and getMarks( ) are getter functions and setGrade( ) is a setter function.

Defining Member functions outside the class

We can also define member functions outside the class using scope resolution operator :: .
For example lets move the definition of the two functions defined in student class above outside the class.

```
class Student {
    int rollno;
    char name[20];
    float marks;
    char grade;
    public :
        int getRollno( );

        int getMarks( );

        void setGrade( ){
            if (marks > 90) grade ='A';;
            else if (marks > 80) grade = 'B';
            else if (marks > 70) grade = 'C';
            else if (marks > 60) grade = 'D';
            else grade = 'E';
        }
};
```

```
int  Student::getMarks(){
  return marks;
}

int  Student::getRollNo(){
  return rollNo;
}
```

We can access member functions in similar manner via an object of class Student and using dot operator.

Constructors

A constructor in a class is means of initializing or creating objects of a class. A constructor allocates memory to the data members when an object is created. It may also initialize the object with legal initial value.
A constructor has following characteristics:

- Constructor is a member function of a class and has same name as that of the class.
- Constructor functions are invoked automatically when the objects are created.
- Constructor functions obey the usual access rules. That is, private constructors are available only for member functions, however, public constructors are available for all functions
- Constructor has no return type, not even void.

Default constructor

A constructor that accepts no parameter is called default constructor. The compiler automatically supplies a default constructor implicitly. This constructor is the public member of the class. It allocates memory to the data members of the class and is invoked automatically when an object of that class is created. Having a default constructor simply means that a program can declare instances of the class.

Example code :

```
class Sum {
    int a, b;
    public :
            int getSum(){
```

```
            return a + b;
        }
} ;
int main() {
    Sum obj;       //implicit default constructor invoked
}
```

Whenever an object of person class is created implicit default constructor is invoked automatically that assigns memory to its data members, i.e., ***name*** and ***age.***

- **Creating your own default constructor**

One can define their own default constructor. When a user-defined default constructor is created, the compiler's implicit default constructor is overshadowed.

Example code :

```
class Sum {
    int a, b;
    public :
        Sum( ) {              // user-defined default constructor
            cout << "constructor invoked";
            a = 10;
            b = 20;
        }
        int getSum(){
            return a + b;
        }
} ;
int main() {
    Sum obj;                  //explicitly defined default constructor invoked
}
```

Output :

constructor invoked

In this case, user-defined default constructor will be invoked when an object **obj** of person class is created and the data members of that object, ***a*** and ***b***, are initialized to default values 10 and 20.

Parameterized constructor

The constructors that can take arguments are called parameterized constructor.

Example code :

```
class Sum {
    int a, b;
    public :
            Sum(int num1, int num2 ) {          // parameterized constructor
                    a = num1;
                    b = num2;
            }
            int getSum(){
                    return a + b;
            }
 } ;
 int main() {
     Sum obj(4, 2);                  //parametrized constructor invoked
  }
```

Declaring a constructor with arguments hides the default constructor. Hence, the object declaration statement such as

Person obj;

may not work. It is necessary to pass the initial value arguments to the constructor function when an object is declared. This can be done in two ways :

1. By calling constructor explicitly.
   Sum obj = Sum(4 ,2) ;
2. By calling constructor implicitly.(as illustrated in example code)
   Sum obj(4, 2) ;

Destructors

Just as the objects are created, so are they destroyed. If a class can have constructor to set things up, it should have a destructor to destruct the objects. A destructor as the name itself suggests, is used to destroy the objects that have been created by a constructor. A destructor is also a member function whose name is the same as the class name but preceded by tilde ('~'). For instance, destructor for class Sum will be ~Sum( ).

A destructor takes no arguments, and no return types can be specified for it, not even void. It is automatically called by the compiler when an object is destroyed. A destructor frees up the memory area of the object that is no longer accessible.

Example code:

```
class Sum {
    int a, b;
    public :
        Sum(int num1, int num2 ) {          // parameterized constructor
            cout << "Constructor at work" << endl;
            a = num1;
            b = num2;
        }
        ~Sum( ){                            //destructor
            cout<< "Destructor at work" << endl;
        }
        int getSum(){
            return a + b;
        }
}
int main() {
    Sum obj(4, 6);
 }
```

Output :

Constructor at work
Destructor at work

As soon as obj goes out of scope, destructor is called and obj is destroyed releasing its occupied memory.

NOTE:

- If we fail to define a destructor for a class, the compiler automatically generates one for static allocations.
- Destructors are invoked in the reverse order in which the constructors were called.
- Only the function having access to the constructor and destructor of a class, can define objects of this class types otherwise compiler reports an error.

this keyword

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which this function was called. For example, the function call for obj.getSum() will set the

pointer **this** to the address of the object obj. This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions.

Example code:

```
class Sum {
    int a, b;
    public :
            Sum(int a, int b ) {
                    this->a = a;
                    this->b = b;
            }
            int getSum(){
                    return a + b;
            }
 }
```

In the constructor of the Sum class, since the data members and data members have the same name, this keyword is used to differentiate between the two. Here, ***this->a*** refers to the data member ***a*** of the object obj.

OOPs 2

Shallow and Deep Copy

Shallow Copy:

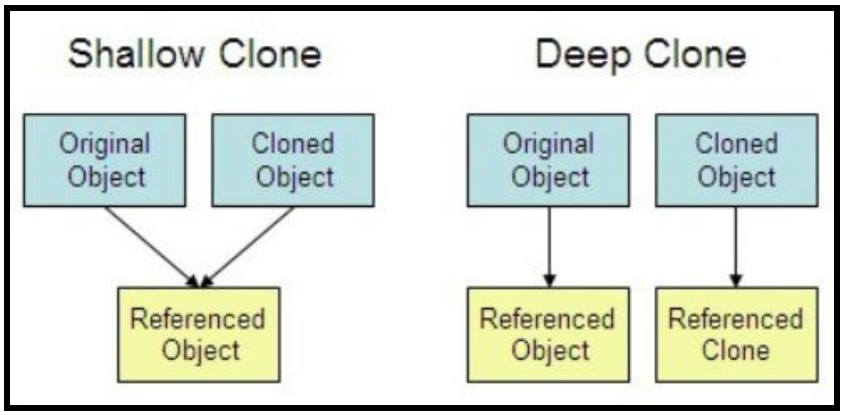
It copies all of the member field values. Here, the pointer will be copied but not the memory it points to. It means that the original object and the created copy will now point to the same memory address, which is generally not preferred.

Note: Assignment operator and the default copy constructor makes a shallow copy.

Deep Copy:

It copies all the fields but creates a copy of dynamically allocated memory pointed to by the fields. Here, we need to make our copy constructor and overload the assignment operator. Advantages of the deep copy are:

- When we need to initialise the class variables to some value or NULL, then a parameterized constructor can be used.
- A destructor that can be used to delete the dynamically allocated memory.



Kindly refer to the code below:

```
class Student {
```

```
        int age;
        char *name;
        public :
        Student(int age, char *name) {   // parameterized constructor
                this -> age = age;
                // Shallow copy
                // this -> name = name;

                // Deep copy
                this -> name = new char[strlen(name) + 1];  // Created a new memory
                strcpy(this -> name, name);
        }
};
```

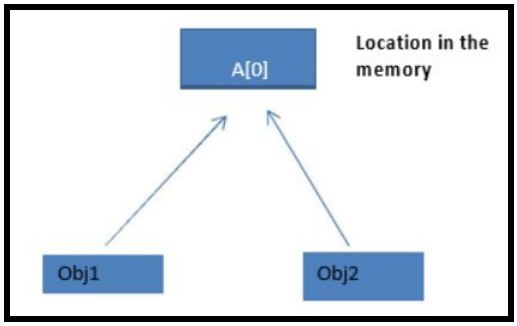
Copy Constructor

The copy constructor is used to create a copy of an already existing object of class type. We can make this copy in two ways:

Shallow copy constructor:

Let's take an example to understand it better. Suppose, two students are entering their details simultaneously from two different machines that are connected over the same network. It will reflect the changes made by both of them in the database because they both are entering their details in the same database.

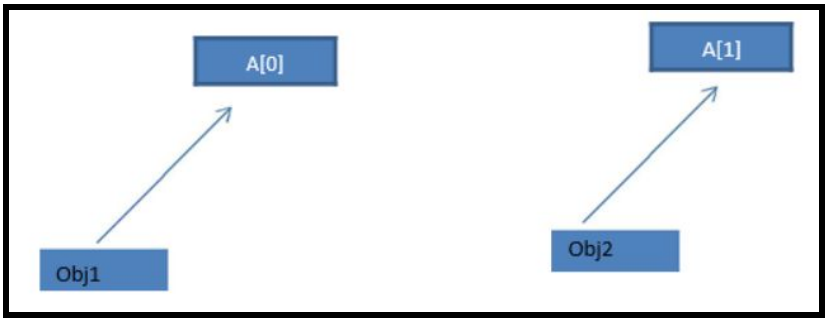
The same is the case with the shallow copy as both share the same locations. It copies references to the original object. **Default copy constructor** uses the same way of copying. It is generally preferred in the cases when the class does not contain any dynamically allocated memory.



Deep copy constructor:

Suppose you have to submit the homework but are short of time. So, you decide to copy the same from your friend. Now, you and your friend have the same work but possess different copies. Now, you choose to modify yours's a bit to prevent the risk of the teacher figuring it out. But the changes you did will not affect your friend's work. Deep copy follows the same concept.

Deep memory allocates different memory for the copied task. It is generally used when we assign the dynamic memory using pointers.



Kindly refer to the code below:

```
// Copy constructor
Student(Student const &s) {      // keyword const assured us that original copy is
                                 // unchanged
        this -> age = s.age;
        // this -> name = s.name;          // Shallow Copy

        // Deep copy
        this -> name = new char[strlen(s.name) + 1];
        strcpy(this -> name, s.name);
}
```

Initialisation list

Using the initialisation list, we can directly assign the values to the data members of the class. Though we can use the initialisation list as needed, in the following cases, we mandatorily need to use the same:

- When no base class default constructor is present
- When the data members are of type **const**
- When the data member and parameter have the same name
- When the data member of the reference type is used.

The syntax begins with a colon(:) and then each variable along with its value separated by a comma. It does not end in a semicolon.

Kindly refer to the code below for better understanding:

```
class Student {
        public :

                int age;
                const int rollNumber;     // Const type data member
                int &x;                   // age reference variable
                // Parameterised list is used
                Student(int r, int age) : rollNumber(r), age(age), x(this -> age) {
                        //rollNumber = r;
                }
};
```

Constant functions

Constant functions are those which don't change any property of current objects. Only constant objects of the class could invoke these.

Syntax:

```
datatype function_name const();
```

Static Members

Suppose, we have a student database and we are creating objects for each student specifying their name, roll number and school name. As discussed earlier, deep copy constructor will be used for the same. For each student, their names and roll number can be different so we will be allotting distinct memories to each. But the school name should be the same. By using a deep copy, we will allocate each student a different memory for the school name. To overcome this, we will be using the **static** keyword.

Syntax for declaring static members:

```
static datatype variable_name;
```

Since it is a property of the class and not of an object, we can't simply access these values from the object name followed by the variable name.

Syntax for invoking static members:

```
Class_name :: variable_name;
```

Note: (::) is called the Scope Resolution Operator.

Apart from variables, functions can also be made static. By declaring a function as static, we make it independent of any particular class object. A static function can be invoked even if no objects of the class exist and the static functions are accessed using only the class name and the (::) operator.

Kindly refer to the code below for better understanding:

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

class Student {
        static int totalStudents;          // total number of students present

        public :
        int rollNumber;
```

```
        int age;

        Student() {
                totalStudents++;
        }

        int getRollNumber() {
                return rollNumber;
        }

        static int getTotalStudent() {          // Static member function
                return totalStudents;
        }
};
int Student :: totalStudents = 0;   // initialize static data members

int main() {
        Student s1;
        Student s2;
        Student s3, s4, s5;

        cout << Student :: getTotalStudent() << endl;  // static function invoked
}
```

Operator Overloading

In C++, we can specify more than one definition of an operator in the same scope, which is called operator overloading. It makes the program more intuitive. Example, if we want to add two fraction numbers, then we can do the same by calling over the numerator and denominator of both fraction object F1 and F2. Using operator Overloading, it can be simply done by F1 + F2. You can see that we are adding two class objects, which is primitively not possible but is possible using operator overloading of '+' operator.

Syntax:

```
return_type operator symbol_to_be_overloaded(arguments){}
```

Here, **operator** is a keyword.

Points to remember while overloading an operator:

- It can be used only for user-defined operators(objects, structures) but cannot be used for in-built operators(int, char, float, etc.).
- Operators = and & are already overloaded in C++, so we can avoid overloading them.
- Precedence and associativity of operators remain intact.

List of operators that can be overloaded in C++:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

List of operators that cannot be overloaded in C++:

::	.*	.	?:

Kindly refer to the code below:

```
class Fraction {
        private :
                int numerator;
                int denominator;

        public :
                Fraction(int numerator, int denominator) {
                        this -> numerator = numerator;
                        this -> denominator = denominator;
```

```
    }

    void print() {
        cout << this -> numerator << " / " << denominator << endl;
    }

    void simplify() {
        int gcd = 1;
        int j = min(this -> numerator, this -> denominator);
        for(int i = 1; i <= j; i++) {
            if(this -> numerator % i == 0 && this -> denominator % i
== 0) {
                gcd = i;
            }
        }
        this -> numerator = this -> numerator / gcd;
        this -> denominator = this -> denominator / gcd;
    }

    Fraction add(Fraction const &f2) {
        int lcm = denominator * f2.denominator;
        int x = lcm / denominator;
        int y = lcm / f2.denominator;

        int num = x * numerator + (y * f2.numerator);

        Fraction fNew(num, lcm);
        fNew.simplify();
        return fNew;
    }

    Fraction operator+(Fraction const &f2) const {
        int lcm = denominator * f2.denominator;
        int x = lcm / denominator;
        int y = lcm / f2.denominator;

        int num = x * numerator + (y * f2.numerator);

        Fraction fNew(num, lcm);
        fNew.simplify();
        return fNew;
    }

    Fraction operator*(Fraction const &f2) const {
        int n = numerator * f2.numerator;
        int d = denominator * f2.denominator;
        Fraction fNew(n, d);
```

```
                fNew.simplify();
                return fNew;
        }

        bool operator==(Fraction const &f2) const {
                return (numerator == f2.numerator && denominator ==
f2.denominator);
        }

        void multiply(Fraction const &f2) {
                numerator = numerator * f2.numerator;
                denominator = denominator * f2.denominator;
                simplify();
        }

        // Pre-increment
        Fraction& operator++() {
                numerator = numerator + denominator;
                simplify();

                return *this;
        }
        // Post-increment
        Fraction operator++(int) {
                Fraction fNew(numerator, denominator);
                numerator = numerator + denominator;
                simplify();
                fNew.simplify();
                return fNew;
        }
        // Short-hand addition operator overloaded
        Fraction& operator+=(Fraction const &f2) {
                int lcm = denominator * f2.denominator;
                int x = lcm / denominator;
                int y = lcm / f2.denominator;

                int num = x * numerator + (y * f2.numerator);

                numerator = num;
                denominator = lcm;
                simplify();

                return *this;
        }
};
```

Dynamic array class

Let us now implement a dynamic array class where we will be creating functions to add an element to the array, extract the array element using the provided index and print the complete array. Kindly refer to the code below:

```
class DynamicArray {
        int *data;
        int nextIndex;
        int capacity;                   // total size

        public :
        DynamicArray() {
                data = new int[5];          // Starting with capacity  = 5
                nextIndex = 0;
                capacity = 5;
        }

        DynamicArray(DynamicArray const &d) {
                //this -> data = d.data;                // Shallow copy

                // Deep copy
                this -> data = new int[d.capacity];
                for(int i = 0; i < d.nextIndex; i++) {
                        this -> data[i] = d.data[i];
                }
                this -> nextIndex = d.nextIndex;
                this -> capacity = d.capacity;
        }
       // operator overloading: We are simply equating two arrays
        void operator=(DynamicArray const &d) {
                this -> data = new int[d.capacity];
                for(int i = 0; i < d.nextIndex; i++) {
                        this -> data[i] = d.data[i];
                }
                this -> nextIndex = d.nextIndex;
                this -> capacity = d.capacity;
        }

       // inserting a new element to the array
        void add(int element) {
                if(nextIndex == capacity) {         // If the capacity is not enough
                        int *newData = new int[2 * capacity];   // We doubled the
                                                                // capacity of the array
```

```
                for(int i = 0; i < capacity; i++) {
                        newData[i] = data[i];    // Elements copied to the array
                }
                delete [] data;
                data = newData;
                capacity *= 2;
        }
        data[nextIndex] = element;
        nextIndex++;
    }

    int get(int i) const {     // For returning the element at particular index
        if(i < nextIndex) {
                return data[i];
        }
        else {
                return -1;
        }
    }

    void add(int i, int element) {
                if(i < nextIndex) {
                        data[i] = element;
                }
                else if(i == nextIndex) {
                        add(element);
                }
                else {
                        return;
                }
    }

    void print() const {                    // For printing the complete array
        for(int i = 0; i < nextIndex; i++) {
                cout << data[i] << " ";
        }
        cout << endl;
    }
};
```

Linked List 1

Data Structures

Data structures are just a way to store and organise our data so that it can be used and retrieved as per our requirements. Using these can affect the efficiency of our algorithms to a greater extent. There are many data structures that we will be going through throughout the course, linked list is a part of them.

Introduction to linked list

Let's first discuss what are the disadvantages of using an array:

- Arrays have a fixed size that is required to be initialised at the time of declaration i.e., the addition of extra elements would throw an error (Index out of range) in C++.
- Arrays could store a maximum of 10^6 - 10^7 elements . If we try to store more elements, then the program might lead to memory overflow.
- Even deletion of elements from the array is an expensive task as it would require the complete shift of further elements by some positions to the left as the data is stored in the form of contiguous memory blocks.

In order to overcome these problems, we can use linked lists...

- A linked list is a linear data structure where each element is a separate object.
- Each element or node of a list is comprising of two items:
    - Data
    - Pointer(reference) to the next node.
- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
- The first node of a linked list is known as head.
- The last node of a linked list is known as tail.
- The last node has a reference to null.

Linked list class

```
class Node {
        public :
        int data;                       // to store the data stored
        Node *next;                     // to store the address of next pointer

        Node(int data) {
                this -> data = data;
                next = NULL;
        }
};
```

Note: The first node in the linked list is known as **Head** pointer and the last node is referenced as **Tail** pointer. We must never lose the address of the head pointer as it references the starting address of the linked list and is, if lost, would lead to losing of the list.

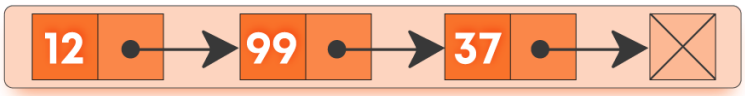
Printing of the linked list

To print the linked list, we will start traversing the list from the beginning of the list(head) until we reach the NULL pointer which will always be the tail pointer. Follow the code below:

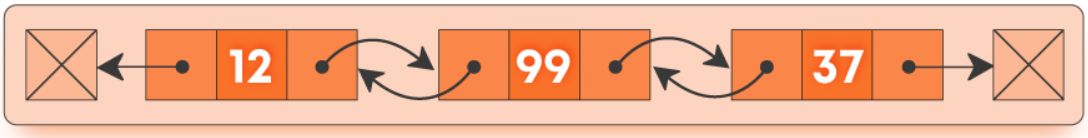
```
void print(Node *head) {
      Node *tmp = head;
      while(tmp != NULL) {
                  cout << tmp->data << " ";
                  tmp = tmp->next;
      }
      cout << endl;
}
```

There are generally three types of linked list:

- **Singly:** Each node contains only one link which points to the subsequent node in the list.



- **Doubly:** It's a two-way linked list as each node points not only to the next pointer but also to the previous pointer.



- **Circular:** There is no tail node i.e., the next field is never NULL and the next field for the last node points to the head node.

Let's now move onto all other operations like insertion, deletion, input in linked list...

Taking input in list

Refer the code below:

```
Node* takeInput() {
        int data;
        cin >> data;
        Node *head = NULL;
        Node *tail = NULL;
        while(data != -1) {                                    // -1 is used for terminating
                Node *newNode = new Node(data);
                if(head == NULL) {
                        head = newNode;
                        tail = newNode;
                }
                else {
                        tail -> next = newNode;
                        tail = tail -> next;
                        // OR
                        // tail = newNode;
                }

                cin >> data;
        }
        return head;
}
```

To take input in the user, we need to keep few things in the mind:

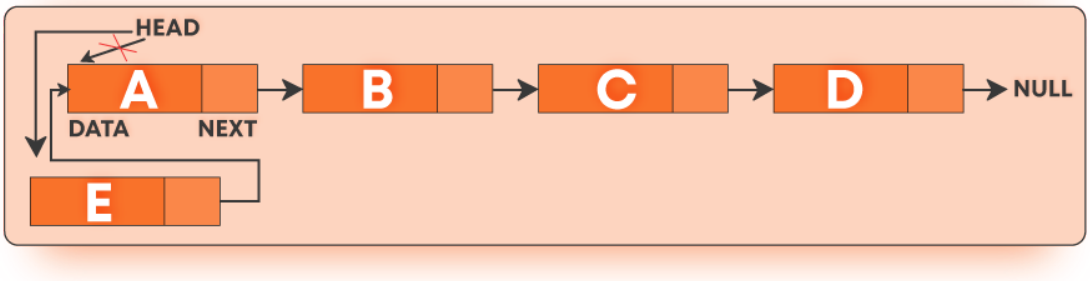
- Always use the first pointer as the head pointer.
- When initialising the new pointer the next pointer should always be referenced to NULL.
- The current node's next pointer should always point to the next node to connect the linked list.

Insertion of node

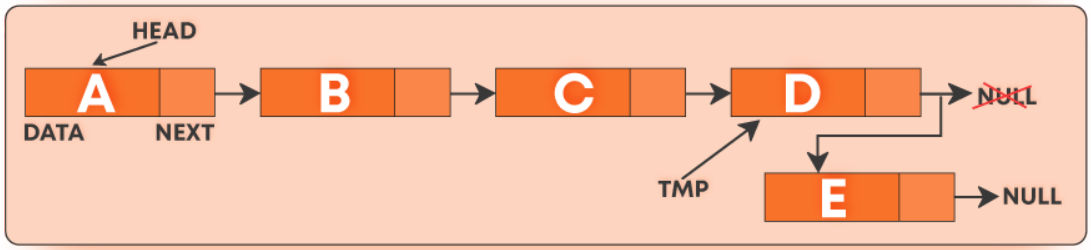
There are 3 cases:

- **Case 1: Insert node at the last**

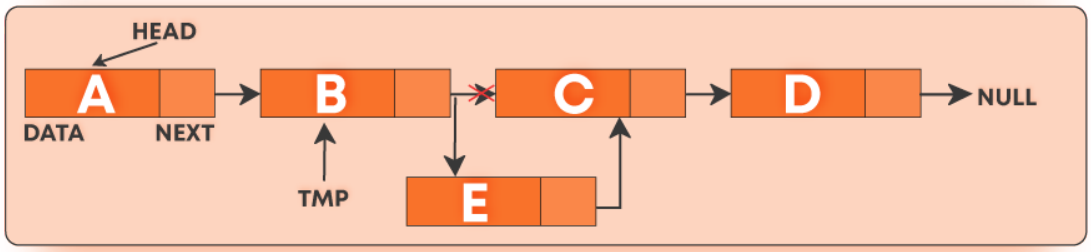
  This can be directly done by normal insertion as discussed above.



- **Case 2: Insert node at the beginning**
  - First-of-all store the head pointer in some other pointer.
  - Now, mark the new pointer as the head and store the previous head to the next pointer of the current head.
  - Update the new head.



- **Case 3: Insert node anywhere in the middle**
    - For this case, we always need to store the address of the previous pointer as well as the current pointer of the location at which new pointer is to be inserted.
    - Now let the new inserted pointer be curr. Point the previous pointer's next to curr and curr's next to the original pointer at the given location.
    - This way the new pointer will be inserted easily.



Let's now check out the code for all the above cases...

```
Node* insertNode(Node *head, int i, int data) {
        Node *newNode = new Node(data);
        int count = 0;
```

```
        Node *temp = head;

        if(i == 0) {                                        //Case 2
                newNode -> next = head;
                head = newNode;
                return head;
        }

        while(temp != NULL && count < i - 1) {              //Case 3
                temp = temp -> next;
                count++;
        }
        if(temp != NULL) {
                Node *a = temp -> next;
                temp -> next = newNode;
                newNode -> next = a;
        }
        return head;                    //Returns the new head pointer after insertion
}
```

Deletion of node

There are 2 cases:

- **Case 1: Deletion of the head pointer**

  In order to delete the head node, we can directly remove it from the linked list by pointing the head to the next.

- **Case 2: Deletion of any node in the list**

  In order to delete the node from the middle/last, we would need the previous pointer as well as the next pointer to the node to be deleted. Now directly point the previous pointer to the current node's next pointer.

Try to code it yourself and then check for the solution provided against the corresponding problem in the course curriculum.

Now, let's move on to the recursive approach for insertion and deletion of the node in a linked list.

Insert node recursively

Follow the steps below and try to implement it yourselves:

- If Head is null and position is not 0. Then exit it.
- If Head is null and position is 0. Then insert a new Node to the Head and exit it.
- If Head is not null and position is 0. Then the Head reference set to the new Node. Finally, a new Node set to the Head and exit it.
- If not, iterate until finding the Nth position or end.

For the code, refer to the Solution section of the problem...

Delete node recursively

Follow the steps below and try to implement it yourselves:

- If the node to be deleted is root, simply delete it.
- To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if the position is not zero, we run a loop position-1 times and get a pointer to the previous node.
- Now, simply point the previous node's next to the current node's next and delete the current node.

For the code, refer to the Solution section of the problem...

You will find more practice problems from other sources in the Linked List 2 module.

Linked List 2

Now moving further with the topic, let's try to solve some problems now...

Here we are only gonna discuss the approach, but you will have to implement it yourselves and in case you are stuck then refer to the solution section of the corresponding problem.

Midpoint of LL

Midpoint of a linked list can be found out very easily by taking two pointers. One named **slow** and the other named **fast**. As their names suggest, they will move in the same way respectively. Fast pointer will move ahead two pointers at a time, while the slower one will move at a speed of a pointer at a time. In this way, when the fast pointer will reach the end, by that time the slow pointer will be at the middle position of the array.

They will move like these:

slow = slow -> next;

fast = fast -> next -> next;

Also, be careful with the even length scenario of the linked lists. For odd length there will be only one middle element, but for the even length there will be two middle elements. The above approach will return the first middle element and the other one(in case of even length list) will be the direct next of the first middle element.

Merge Two sorted linked lists

We will be merging the linked list, similar to the way we performed merge over two sorted arrays.

We will be using the two head pointers, compare their data and the one found smaller will be directed to the new linked list and increase the head pointer of the corresponding linked list. Just remember to maintain the head pointer separately for the new sorted list. And also if one of the linked list's length ends and the other one's not, then the remaining linked list will directly be appended to the final list. Now try to code it...

Mergesort over linked list

Like the merge sort algorithm is applied over the arrays, the same way we will be applying it over the linked list. just the difference is that in case of arrays, the middle element could be easily figured out, but here you have to find the middle element, each time you send the linked list to split into two halves using the above approach and merging part of the divided lists can also be done using the merge sorted linked lists code as discussed above. Basically the functionalities of this code have already been implemented by you, just use them directly in your functions at the specified places.

Reverse the linked list

Recursive approach:

Basically, we will store the last element of the list in the small answer, and then update that by adding the next last node and so on. Finally, when we will be reaching the first element, we will assign the **next** to NULL. Follow the code below, for better understanding...

```
Node* reverseLL(Node *head) {
        if(head == NULL || head -> next == NULL) {          //Base case
                return head;
        }

        Node *smallAns = reverseLL(head -> next);           // Recursive call

        Node *temp = smallAns;                              // small answer that
        while(temp -> next != NULL) {                       // stores the reversed
                temp = temp -> next;                        // list by traversing the
        }                                                   // reversed list and
                                                            // then appending the
        temp -> next = head;                                // next element to it.
        head -> next = NULL;
        return smallAns;
}
```

After calculation you can see that this code has a time complexity of O(n^2). Now let's think on how to improve it...

There is another recursive approach in which we can simply use the O(n) approach. What we will be doing is creating a pair class that will be storing the reference of not only the head but also the tail pointer, which can save our time in searching over the list to figure out the tail pointer for appending or removing. Checkout the code for your reference...

```
class Pair {
        public :                        //Pair class about which we were talking above
                Node *head;
                Node *tail;
};

Pair reverseLL_2(Node *head) {
        if(head == NULL || head -> next == NULL) {          //Base case
                Pair ans;
                ans.head = head;
                ans.tail = head;
                return ans;
        }

        Pair smallAns = reverseLL_2(head -> next);          //Recursive call

        smallAns.tail -> next = head;                       // you can see that the time
        head -> next = NULL;                                // is reduced as we do
        Pair ans;                                           //not need to find the tail
        ans.head = smallAns.head;                           // pointer each time
        ans.tail = head;
        return ans;
}
```

Now improving this code further...

A simple observation is that the tail is always the head->next. By making the recursive call we can directly use this as our tail pointer and reverse the linked list by tail->next = head.

Refer to the code below...

```
Node* reverseLL_3(Node *head) {
        if(head == NULL || head -> next == NULL) {          //Base case
                return head;
        }
        Node *smallAns = reverseLL_3(head -> next);         //Recursive call

        Node *tail = head -> next;                          //Small calculation
        tail -> next = head;                                //discussed above
        head -> next = NULL;
        return smallAns;
}
```

Iterative approach:

We will be using three-pointers in this approach: **previous, current and next.** Initially, the previous pointer would be NULL as in the reversed linked list, we want the original head to be the last element pointing to NULL. Current pointer will be the current node whose next will be pointing to the previous element but before pointing it to the previous element, we need to store the next element's address somewhere otherwise we will lose that element. Similarly, iteratively, we will keep updating the pointers as current to the next, previous to the current and next to current's next.

You will be solving this problem yourself now...

Variations of the linked list

In the lecture notes of Linked list - 1, we have already seen the three different types of linked list and discussed them diagrammatically also. Prefer to that section for the reference...

Practice problems

Try over the following link to practice some good questions related to linked lists:

https://www.hackerrank.com/domains/data-structures?filters%5Bsubdomains%5D%5B%5D=linked-lists

Abstract Data Type (ADT)

Data Abstraction & ADT

When you switch on the fan and it starts rotating at its desired speed, you don't care about the process of its internal working. Fan works unless there is a power cut. This real-world example highlights the programming concept of **data abstraction**, which allows a programmer to protect/hide the implementation of a process and only give the "keys" to other functions or user data. Similarly, in the case of programming when a user tries to find out some information, it is provided to him/her without showing the internal working of the functions in our program.

Data types created using the same concept are known as abstract data types. An **abstract data type** (or ADT) is a class that has a defined set of operations and values. If we consider the above example, then making the fan's switch as the entire abstract data type will prevent the user from knowing all of its internal working.

Features of ADT

- Prevents users from knowing the working of the program.
- Provides only specific data asked for if it's asking for an accurate set of data.

Methods of Abstraction

There are mainly three ways of data abstraction:

Procedural Abstraction

- Performs already stored repeated tasks, if asked for.
- Follows a fixed procedure of instructions provided.
- Generally, used in programs by invoking them. For example: While implementing a square root function in C++, we prefer to directly call the in-built **sqrt()** function and provide arguments to it, but we are not allowed to view the complete program written for this function.
- **Disadvantage:** Data is public, hence could be modified by any programmer using it.

Modular (File) Abstraction

- Contains both public and private sections and hence, can be used as per requirements.
- **Disadvantage:** Data can either be public or private at a single time.

Object-Oriented Programming

- The most efficient method of data abstraction.
- Data and procedures are instantiated together within an object.
- Multiple instantiations possible overcoming the disadvantage of modular abstraction.
- Accessing data can be controlled by the programmer.
- The program becomes more robust and readable using it.

Templates

Templates are a good way of making classes more abstract by letting you define the behavior of the class without actually knowing what data type will be handled by the operations of the class. The advantage of having a single class - that can handle several different data types is that it makes the code easier to maintain, and it makes classes more reusable.

Syntax for one type of variable only:

```
template <typename T>
```

Syntax for two types of variables:

```
template <typename T, typename V>
```

For more than two types of variables, more type names can be added within the angular brackets.

For Example: You want to create a pair class and use it for different types of variables, like int, char, double etc... Follow the code below:

```
template <typename T>

class Pair {
        T x;            //variables x and y are declared of type T which
        T y;            //could take place of any known data type

        public :

        void setX(T x) {
                this -> x = x;
        }

        T getX() {
                return x;
        }
```

```
        void setY(T y) {
                this -> y = y;
        }

        T getY() {
                return y;
        }
};
```

Now suppose you want to create *x* of one type (like int) and *y* of another type (for eg. double), then you can do it in the following way:

```
template <typename T, typename V>

class Pair {
        T x;          //variables x and y are declared of type T and V respectively,
        V y;          //which could take place of any known data types(different/same)

        public :

        void setX(T x) {
                this -> x = x;
        }

        T getX() {
                return x;
        }

        void setY(V y) {
                this -> y = y;
        }

        V getY() {
                return y;
        }
};
```

To use these in the main() function, simply call the class like this:

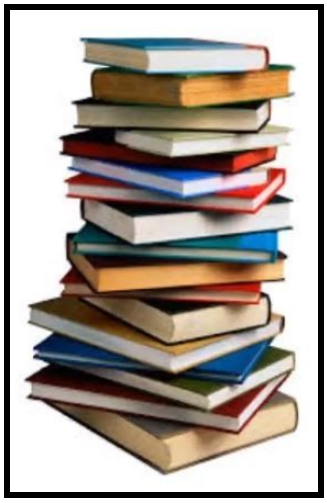
```
Pair<int, double> p1;
```

and the pair of the required types will be created.

Stacks

Introduction

- It is also a linear data structure like arrays and linked lists.
- It is an abstract data type(ADT).
- This is also known as recursion type data structure.
- It follows LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out.**
- Consider the example of a pile of books:



Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

Like deletion, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as **LIFO.**

Operations on the stack:

- **Insertion**: This is known as **push** operation.
- **Deletion**: This is known as **pop** operation.
- **Access to the topmost element:** This operation is performed by the **top** function.
- **Size of the stack:** This operation is performed by the **size** function which returns the number of elements in the stack.
- **To check for filled/empty stack:** This operation is performed by **isEmpty** function which returns boolean value. It returns true for empty stack and false otherwise.

Implementing stack using array

As in the arrays we can access any elements but here we want to use it as a stack which means we should be only allowed to access the topmost element of it. Hence, to use an array as a stack we will be keeping the access to the former as **private** in the class. This way we can put restrictions on the various operations discussed above.

Check the code below for better understanding of the approach (follow-up in the comments)...

```
#include <climits>
class StackUsingArray {
        //Privately declared
        int *data;                          // Dynamic array created serving as stack
        int nextIndex;                      // To keep the track of current top index
        int capacity;                       // To keep the track of total size of stack

        public :
        StackUsingArray(int totalSize) {    //Constructor to initialise the values
                data = new int[totalSize];
                nextIndex = 0;
```

```
        capacity = totalSize;
    }

    // return the number of elements present in my stack
    int size() {
        return nextIndex;
    }

    bool isEmpty() {
        /*
        if(nextIndex == 0) {
            return true;
        }
        else {
            return false;
        }
        */

        return nextIndex == 0;        //Above program written in short-hand
    }

    // insert element
    void push(int element) {
        if(nextIndex == capacity) {
            cout << "Stack full " << endl;
            return;
        }
        data[nextIndex] = element;
        nextIndex++;                  //Size incremented
    }
    // delete element
    int pop() {
//Before deletion we checked if it was initially not empty to prevent underflow
        if(isEmpty()) {
            cout << "Stack is empty " << endl;
            return INT_MIN;
        }
        nextIndex--;                  //Conditioned satisfied so deleted
        return data[nextIndex];
    }
   //to return the top element of the stack
    int top() {
        if(isEmpty()) {               // checked for empty stack to prevent overflow
            cout << "Stack is empty " << endl;
            return INT_MIN;
        }
        return data[nextIndex - 1];
```

```
    }
};
```

Dynamic stack

There is one limitation to the above approach, which is the size of the stack is fixed. In order to overcome this limitation, whenever the size of the stack reaches its limit we will simply double its size. To get the better understanding of this approach, look at the code below...

```
#include <climits>

class StackUsingArray {
    int *data;
    int nextIndex;
    int capacity;

    public :
    StackUsingArray() {
        data = new int[4];              //initially declared with a small size of 4
        nextIndex = 0;
        capacity = 4;
    }

    // return the number of elements present in my stack
    int size() {
        return nextIndex;
    }
    bool isEmpty() {
        return nextIndex == 0;
    }

    // insert element
    void push(int element) {
        if(nextIndex == capacity) {
            int *newData = new int[2 * capacity];       //Capacity doubled
            for(int i = 0; i < capacity; i++) {
                newData[i] = data[i];                   //Elements copied
            }
            capacity *= 2;
            delete [] data;
            data = newData;
            /*cout << "Stack full " << endl;
            return;*/
```

```
            }
            data[nextIndex] = element;
            nextIndex++;
        }

        // delete element
        int pop() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return INT_MIN;
            }
            nextIndex--;
            return data[nextIndex];
        }
        int top() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return INT_MIN;
            }
            return data[nextIndex - 1];
        }
};
```

Stack using templates

While implementing the dynamic stack, we kept ourselves limited to the data of type integer only, but what if we want a generic stack(something that works for every other data type as well). For this we will be using templates. Refer the code below(based on the similar approach as used while creating dynamic stack):

```
#include <climits>

template <typename T>                    // Templates initialised
class StackUsingArray {
        T *data;                         // Template type of data used
        int nextIndex;
        int capacity;

        public :

        StackUsingArray() {
```

```
            data = new T[4];
            nextIndex = 0;
            capacity = 4;
        }
        // return the number of elements present in my stack
        int size() {
            return nextIndex;
        }
        bool isEmpty() {
            return nextIndex == 0;
        }

        // insert element
        void push(T element) {
            if(nextIndex == capacity) {
                T *newData = new T[2 * capacity];
                for(int i = 0; i < capacity; i++) {
                    newData[i] = data[i];
                }
                capacity *= 2;
                delete [] data;
                data = newData;
            }
            data[nextIndex] = element;
            nextIndex++;
        }

        // delete element
        T pop() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return 0;
            }
            nextIndex--;
            return data[nextIndex];
        }
        // For extracting top element
        T top() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return 0;
            }
            return data[nextIndex - 1];
        }
};
```

You can see that every function whose return type was int initially now returns T type (i.e., template-type).

Generally, the template approach of stack is preferred as it can be used for any data type irrespective of it being int, char, float, etc...

Stack using LL

Till now we have learnt how to implement a stack using arrays, but as discussed earlier, we can also create a stack with the help of linked lists.

All the five functions that stacks can perform could be made using linked lists. Below is the template for you to work upon the same In case you are unable to come up with the program, kindly refer to the code in the solution section of the problem.

```
#include <iostream>
using namespace std;
template <typename T>
class Node {                    //Node class for Linked list
        public :
                T data;
                Node<T> *next;
                Node(T data) {
                        this -> data = data;
                        next = NULL;
                }
                ~Node() {
                        delete next;
                }
};

template <typename T>
class Stack {
        Node<T> *head;
         Node<T> *tail;
        int size;               // number of elements present in stack
```

```
        public :
        Stack() {                 // Constructor to initialize the head and tail to NULL and
                                  // size to zero
        }

        int getSize() {           // traverse the whole linked list and return its length

        }

        bool isEmpty() {          // Just check if the head pointer is NULL or not

        }

        void push(T element) {    // insert the newNode at the end of the list and
                                  // update the tail node
        }

        T pop() {    // remove the tail node from the list and then update the tail
                     // pointer to the previous position by traversing the list again.
        }

        T top() {    //return the value at the tail pointer. No change needed.

        }
};
```

In-built stack using STL

C++ provides the in-built stack in it's **standard temporary library (STL)** which can be used instead of creating/writing a stack class each time. To use this stack, we need to use the header file:

```
#include <stack>
```

To declare a stack use the following syntax:

```
stack <datatype_of_variables_that_will_be_stored> Name_of_stack;
```

For example: to create a stack of integer type with name 'st':

stack <int> st;

Now, you can simply perform all the operations using the in-built stack functions:

- **st.push(value_to_be_inserted)** : To insert a value in the stack
- **st.top()** : Returns the value at the top of the stack
- **st.pop()** : Deletes the value at the top from the stack.
- **st.size()** : Returns the total number of elements in the stack.
- **st.isEmpty()** : Returns a boolean value (True for empty stack and vice versa).

Practice Problems:

- https://www.hackerrank.com/challenges/equal-stacks/problem
- https://www.hackerrank.com/challenges/simple-text-editor/problem
- https://www.techiedelight.com/design-a-stack-which-returns-minimum-element-without-using-auxiliary-stack/
- https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/practice-problems/algorithm/monk-and-order-of-phoenix/

Queues

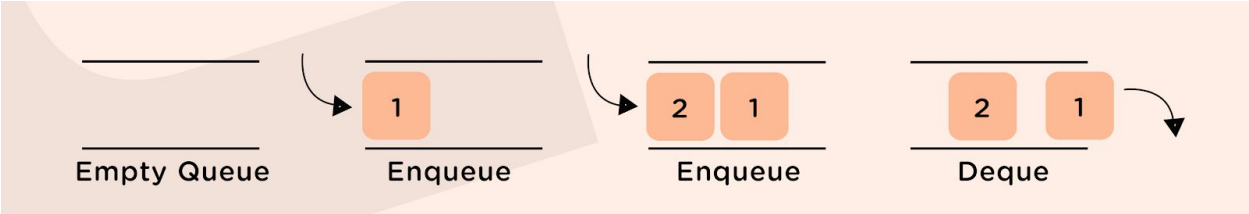
Introduction

Like stack, the queue is also an abstract data type. As the name suggests, in queue elements are inserted at one end while deletion takes place at the other end. Queues are open at both ends, unlike stacks that are open at only one end(the top).

Let us consider a queue at a movie ticket counter:

Here, the person who comes first in the queue is served first with the ticket while the new seekers of tickets are added back in the line; this order is known as **First In First Out (FIFO)**, a principle that queue data structure follows

In programming terminology, the operation to add an item to the queue is called "enqueue", whereas removing an item from the queue is known as "dequeue".

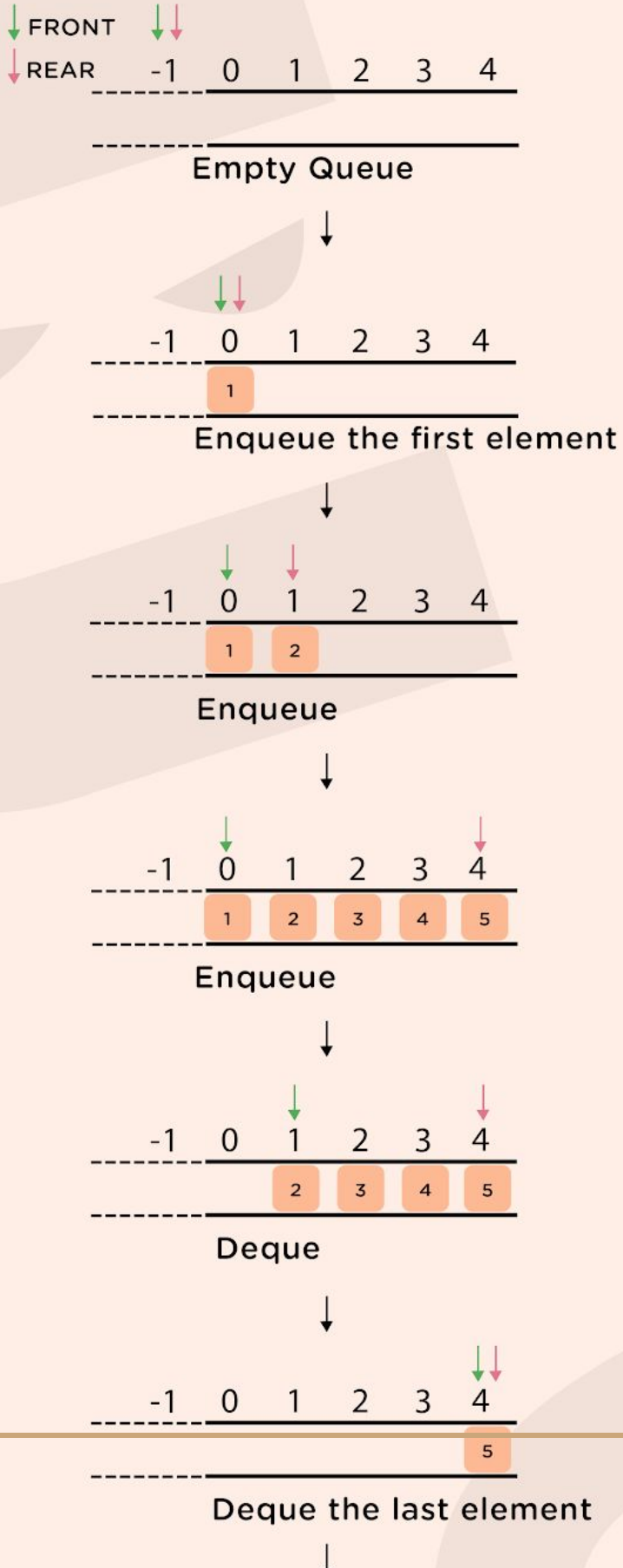


How does the queue work?

Queue operations work as follows:

1. Two pointers called **FRONT** and **REAR** are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On **enqueuing** an element, we increase the value of the REAR index and place the new element in the position pointed to by REAR.
4. On **dequeuing** an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueuing, we check if the queue is already full.
6. Before dequeuing, we check if the queue is already empty.
7. When enqueuing the first element, we set the value of FRONT to 0.
8. When dequeuing the last element, we reset the values of FRONT and REAR to -1.

Refer to the pictorial representation below:


FRONT
REAR
-1 0 1 2 3 4
Empty Queue
-1 0 1 2 3 4
1
Enqueue the first element
-1 0 1 2 3 4
1 2
Enqueue
-1 0 1 2 3 4
1 2 3 4 5
Enqueue
-1 0 1 2 3 4
2 3 4 5
Deque
-1 0 1 2 3 4
5
Deque the last element

Applications of queue:

- CPU Scheduling, Disk Scheduling.
- When data is transferred asynchronously between two processesQueue is used for synchronization. eg: IO Buffers, pipes, file IO, etc.
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people in order of their calling.

Queue using array

Queue contains majorly these five functions that we will be implementing:

- **enqueue()**: Insertion of element
- **dequeue()**: Deletion of element
- **front()**: returns the element present in the front position
- **getSize()**: returns the total number of elements present at current stage
- **isEmpty()**: returns boolean value, TRUE for empty and FALSE for non-empty.

Now, let's implement these functions in our program.

NOTE: We will be using templates in the implementation, so that it can be generalised.

Implementation of queue using array

Follow up the code along with the comments below:

```
template <typename T>                          // Generalised using templates
class QueueUsingArray {
        T *data;                               // to store data
        int nextIndex;                         // to store next index
        int firstIndex;                        // to store the first index
        int size;                              // to store the size
        int capacity;                          // to store the capacity it can hold

        public :
        QueueUsingArray(int s) {               // Constructor to initialize values
                data = new T[s];
                nextIndex = 0;
                firstIndex = -1;
                size = 0;
                capacity = s;
        }

        int getSize() {                        // Returns number of elements present
                return size;
        }

        bool isEmpty() {                       // To check if queue is empty or not
                return size == 0;
        }

        void enqueue(T element) {              // Function for insertion
                if(size == capacity) {         // To check if the queue is already full
                        cout << "Queue Full ! " << endl;
                        return;
                }
                data[nextIndex] = element;     // Otherwise added a new element
                nextIndex = (nextIndex + 1) % capacity ;    // in cyclic way
                if(firstIndex == -1) {         // Suppose if queue was empty
                        firstIndex = 0;
                }
                size++;                        // Finally, incremented the size
        }
```

```
        T front() {                                 //  To return the element at front position
                if(isEmpty()) {                     // To check if the queue was initially empty
                        cout << "Queue is empty ! " << endl;
                        return 0;
                }
                return data[firstIndex];    // otherwise returned the element
        }

        T dequeue() {                               // Function for deletion
                if(isEmpty()) {                     // To check if the queue was empty
                        cout << "Queue is empty ! " << endl;
                        return 0;
                }
                T ans = data[firstIndex];
                firstIndex = (firstIndex + 1) % capacity;
                size--;                             // Decrementing the size by 1
                if(size == 0) {                     // If queue becomes empty after deletion, then
                        firstIndex = -1;    // resetting the original parameters
                        nextIndex = 0;
                }
                return ans;
        }
};
```

Like stack, we can also make dynamic queues (discussed in the further sections)...

Dynamic queue

In the dynamic queue. we will be preventing the condition where the queue becomes full and we were not able to insert any further elements in that. Let's now check the implementation of the same.

Implementation is pretty similar to the static approach discussed above. A few minor changes are there which could be followed with the help of comments in the code below.

```
template <typename T>

class QueueUsingArray {
        T *data;
        int nextIndex;
        int firstIndex;
        int size;
        int capacity;

        public :
        QueueUsingArray(int s) {
                data = new T[s];
                nextIndex = 0;
                firstIndex = -1;
                size = 0;
                capacity = s;
        }

        int getSize() {
                return size;
        }

        bool isEmpty() {
                return size == 0;
        }

        void enqueue(T element) {
                if(size == capacity) {                          // When size becomes full
                        T *newData = new T[2 * capacity];       // we simply doubled the
                                                                // capacity
                        int j = 0;
                        for(int i = firstIndex; i < capacity; i++) {  // Now copied the
                                                                      Elements to new one
                                newData[j] = data[i];
                                j++;
                        }
                        for(int i = 0; i < firstIndex; i++) {       // Overcoming the initial
                                                                    //  cyclic insertion by copying
                                                                    //  the elements linearly
                                newData[j] = data[i];
                                j++;
                        }
                        delete [] data;
                        data = newData;
```

```
                firstIndex = 0;
                nextIndex = capacity;
                capacity *= 2;                  // Updated here as well
                //cout << "Queue Full ! " << endl;
                // return;
        }
        data[nextIndex] = element;
        nextIndex = (nextIndex + 1) % capacity ;
        if(firstIndex == -1) {
                firstIndex = 0;
        }
        size++;
    }

    T front() {
        if(isEmpty()) {
                cout << "Queue is empty ! " << endl;
                return 0;
        }
        return data[firstIndex];
    }

    T dequeue() {
        if(isEmpty()) {
                cout << "Queue is empty ! " << endl;
                return 0;
        }
        T ans = data[firstIndex];
        firstIndex = (firstIndex + 1) % capacity;
        size--;
        if(size == 0) {
                firstIndex = -1;
                nextIndex = 0;
        }
        return ans;
    }
};
```

The STL queues in C++ are implemented in a similar fashion.

We can also implement the queues with the help of linked lists.

Queues using LL

Check the function description below and try to implement it yourselves.

```
template <typename T>
class Node {                                    // Node class for linked list, no change needed
        public :
                T data;
                Node<T> *next;
                Node(T data) {
                        this -> data = data;
                        next = NULL;
                }
};
template <typename T>
class Queue {
        Nod<T> *head;                   // for storing front of queue
        Node<T> *tail;                  // for storing tail of queue
        int size;                       // number of elements in queue

        public :
        Queue() {                       // Constructor to initialise head, tail to NULL
                                        // and size to 0
        }

        int getSize() {                 // just return the size of linked list

        }

        bool isEmpty() {                // just check if head is NULL or not

        }

        void enqueue(T element) {   // Simply insert the new node at the tail of LL

        }

        T front() {                     // Returns the head pointer of LL. Be careful for
                                        // the case when size is 0
        }

        T dequeue() {                   // moves the head pointer one position ahead
                                        // and deletes the head pointer. Also decrease the
```

```
        }                                   // size by 1
};
```

In-built queue

C++ provides the in-built queue in it's **standard temporary library (STL)** which can be used instead of creating/writing a queue class each time.

Header file used:

```
#include <queue>
```

Syntax for declaration of queue:

```
queue <datatype> name_of_queue
```

Key functions of this in-built queue:

- **.push(element_value)** : Used to insert the element in the queue
- **.pop()** : Used to delete the element from the queue
- **.front()** : Returns the element at front of the queue
- **.size()** : Returns the total number of elements present in the queue
- **.empty()** : Returns TRUE if the queue is empty and vice versa
-

Let us now consider an example to implement queue using STL:

Problem Statement: Implement the following parts using queue:

1. Declare a queue of integers and insert the following elements in the same order as mentioned: 10, 20, 30, 40, 50, 60.
2. Now tell the element that is present at the front position of the queue
3. Now delete an element from the front side of the queue and again tell the element present at the front position of the queue.

4. Print the size of the queue and also tell if the queue is empty or not.

5. Now, print all the elements that are present in the queue.

Solution: Check the code for the above stated...

```
#include <iostream>
#include <queue>                        // Header file for using in-built queue
using namespace std;

int main() {
        queue<int> q;                   // queue declared of type int with name q
        q.push(10);                     // Now inserted elements using .push()
        q.push(20);                     // Part 1
        q.push(30);
        q.push(40);
        q.push(50);
        q.push(60);

        cout << q.front() << endl;      // Part 2
        q.pop();                        // Part 3
        cout << q.front() << endl;      // Part 3
        cout << q.size() << endl;       // Part 4
        cout << q.empty() << endl;          // prints 1 for TRUE and 0 for FALSE(Part 4)

        while(!q.empty()) {                 // prints all the elements until the queue
                                            // is empty (Part 5)
                cout << q.front() << endl;
                q.pop();
        }
}
```

Output of the following code is:

```
10
20
5
0
20
30
40
50
60
```

Practice problems:

- https://www.spoj.com/problems/ADAQUEUE/
- https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/practice-problems/algorithm/number-recovery-0b988eb2/
- https://www.codechef.com/problems/SAVJEW
- https://www.hackerrank.com/challenges/down-to-zero-ii/problem

Trees

Introduction

There are various systems around us which are hierarchical in nature; military, government organisations, corporations, they all have a hierarchical form of command chain. There are various systems around us which are hierarchical in nature; military, government organisations. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, those subordinates further have people reporting to them and so on. If we wanted to model this company with a data structure, it would be natural to think of the president as the head of all, the vice presidents at level 1, and their subordinates at lower levels as we go down the organizational hierarchy.

The examples of hierarchical models that we discussed above cannot be represented/stored using a linear data structure. For this very scenario, a data structure called a tree comes to our rescue. Now there are various types of tree, depending on the rule/property they follow. The simplest variant of tree is called a **general tree** (or N-ary tree). As in above example, the number of vice presidents is likely to be more than zero, we need to use a data structure that represents the data in the form of a hierarchy.

In this module we will examine general tree terminology and define a basic ADT for general trees.

Trees are non-linear hierarchical data structures. It is a collection of nodes connected to each other by means of “edges” which are either directed or undirected. One of the nodes is designated as “Root node” and the remaining nodes are called child nodes or the leaf nodes(nodes with no chid nodes).

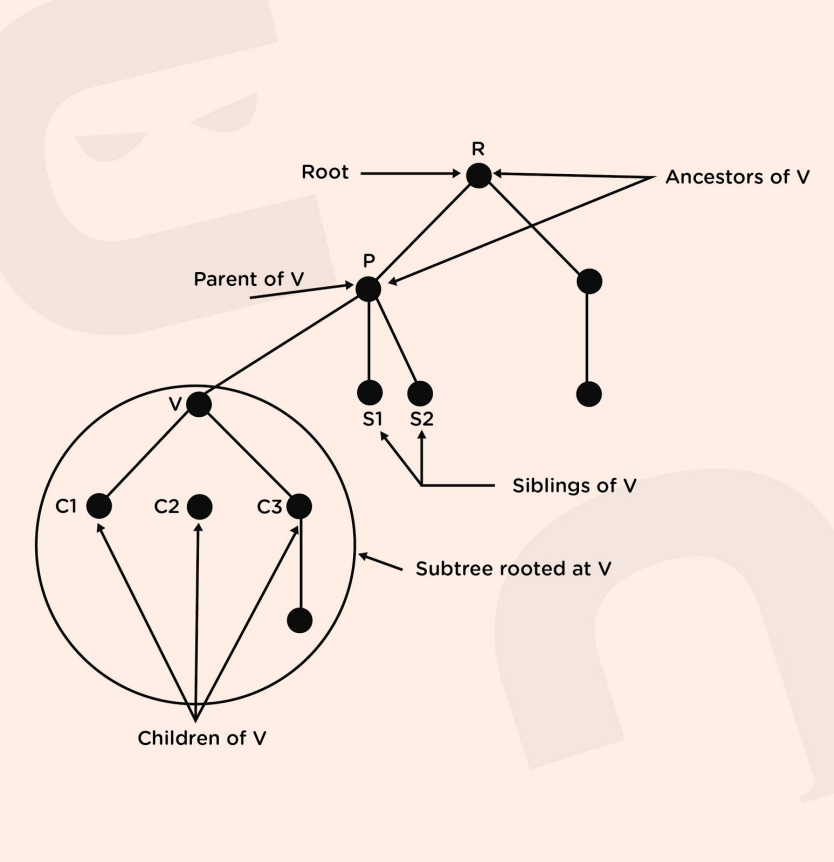
In general, each node can have as many children but only one parent node.

More formally, a tree T is a finite set of one or more nodes such that there is one designated node R, called the root of the tree. If the set (T - {R}) is not empty, these nodes are partitioned into n > 0 disjoint sets T_0, T_1, T_2,..., T_{n-1}, each of which is a tree, and whose roots R_1, R_2,..., R_n, respectively, are children of R. The subsets T_i (($0 \leq i < n$) are said to be ***subtrees*** of T. These subtrees, as ordered in that T_i set, are said to come before T_j if $i < j$. By convention, the subtrees are arranged from left to right with subtree T_0 called the leftmost child of R.

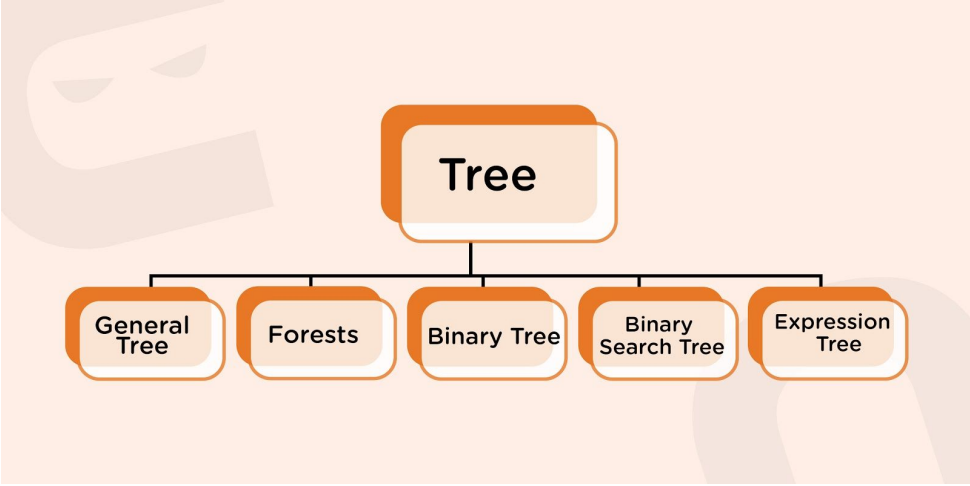
Let us go through some basic terminology associated with trees:

- **Root node:** This is the topmost node in the tree hierarchy.
- **Leaf node:** These are the bottommost nodes in a tree hierarchy. The leaf nodes do not have any child nodes. They are also known as external nodes.
- **Subtree:** Subtree represents various descendants of a node when the root is not null. A tree usually consists of a root node and one or more subtrees.
  Parent node: Any node except the root node that has a child node and an edge upward towards the parent.
- **Ancestor Node:** It is any predecessor node on a path from the root to that node. Note that the root does not have any ancestors.
- **Key:** It represents the data stored in a node.
- **Level:** Represents the generation of a node. A root node is always at level 1. Child nodes of the root are at level 2, grandchildren of the root are at level 3 and so on. In general, each node is at a level higher than its parent.
- **Path:** The path is a sequence of consecutive edges.
- **Degree:** Degree of a node indicates the number of children that a node has.

Below is provided the pictorial representation of above:



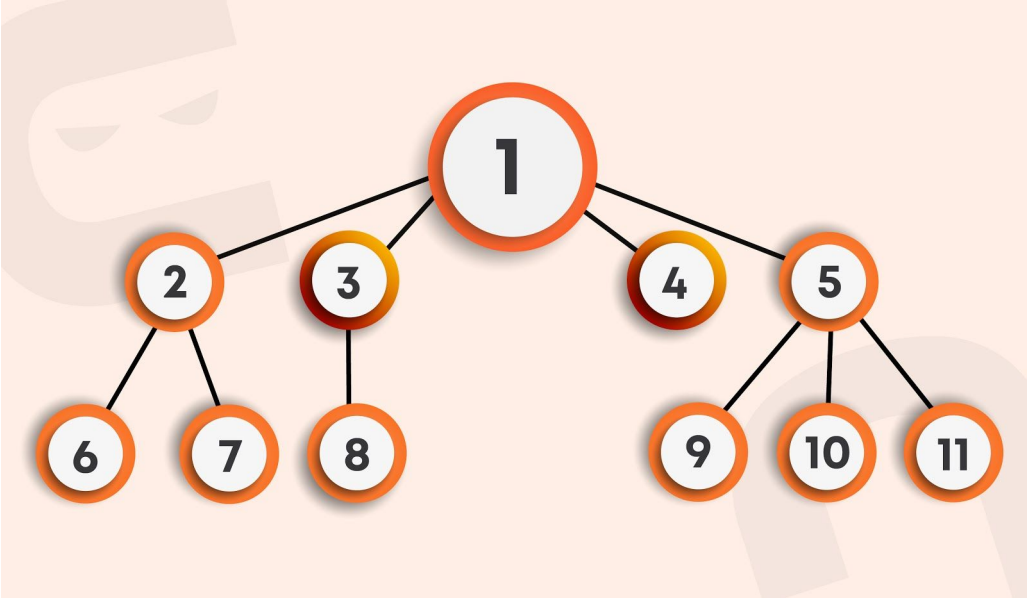
Types of Trees



In this course, we will discuss only General trees, Binary trees and Binary Search trees.

Tree node class

Before discussing general tree implementations, we should first make precise what operations such implementations must support. Firstly, any implementation must be able to initialize a tree. Given a tree, we need access to the root of that tree. There must be some way to access the children of a node. In the case of the ADT for binary tree nodes, this was done by providing member functions that give explicit access to the left and right child pointers. Unfortunately, because we do not know in advance how many children a given node will have in the general tree, we cannot give explicit functions to access each child. An alternative must be found that works for an unknown number of children.



One choice would be to provide a function that takes as its parameter the index for the desired child. That combined with a function that returns the number of children for a given node would support the ability to access any node or process all children of a node. Unfortunately, this view of access tends to bias the choice for

node implementations in favor of an array-based approach, because these functions favor random access to a list of children. In practice, an implementation based on a linked list is often preferred.

An alternative is to provide access to the first (or leftmost) child of a node, and to provide access to the next (or right) sibling of a node.

Here are the class declarations for general trees and their nodes. Based on these two access functions, the children of a node can be traversed like a list. Trying to find the next sibling of the rightmost sibling would return null.

Kindly refer below for code- (we will name this file as **TreeNode.h** to use it further in our program)

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

template <typename T>
class TreeNode {
        public:
        T data;                                 // To store data
        vector<TreeNode<T>*> children;          // To store children for each node

        TreeNode(T data) {                      // Constructor to initialize data
                this->data = data;
        }
};
```

Taking input and print Recursive

Kindly follow the comments in the code to understand it better...

```
#include <iostream>
#include "TreeNode.h"                   // TreeNode.h file included as told above
using namespace std;
```

```
TreeNode<int>* takeInput() {   // Function that returns root node after taking input
        int rootData;                // To store root data
        cout << "Enter data" << endl;
        cin >> rootData;
        TreeNode<int>* root = new TreeNode<int>(rootData);
        // Dynamically created a root node and initialized with constructor

        int n;                    // To store number of children of the node
        cout << "Enter num of children of " << rootData << endl;
        cin >> n;
        for (int i = 0; i < n; i++) {
                TreeNode<int>* child = takeInput();        // Input taken recursively for
                                                           // each child node of the current node
                root->children.push_back(child);      // Each child node is inserted into
                                                           // the list of children nodes'
        }
        return root;
}

void printTree(TreeNode<int>* root) {    // Function to print the tree that takes the
                                         //  root node as its argument

        if (root == NULL) {                    // Base case
                return;
        }

        cout << root->data << ":";
        for (int i = 0; i < root->children.size(); i++) {    // Traversing over the vector of
                                                             // its child nodes and printing each of it
                cout << root->children[i]->data << ",";
        }
        cout << endl;
        for (int i = 0; i < root->children.size() ; i++) {      // Now recursively calling print
                printTree(root->children[i]);           // function over each child
        }
}

int main() {
        TreeNode<int>* root = takeInput();
        printTree(root);
      // TODO : Delete tree
}
```

We learnt that the memory that is created dynamically also needs to be deleted. The same will be followed here also. We suggest you implement it yourself as an exercise.

HINT: Idea will be recursively traversing over each node of the tree and first delete each child and then delete the root node.

Take input level-wise

For taking input level-wise, we will use **queue data structure.** Follow the comments in the code below:

```
TreeNode<int>* takeInputLevelWise() {           // Function to take level-wise input
        int rootData;
        cout << "Enter root data" << endl;
        cin >> rootData;
        TreeNode<int>* root = new TreeNode<int>(rootData);

        queue<TreeNode<int>*> pendingNodes; // Queue declared of type TreeNode
        pendingNodes.push(root);                // Root data pushed into queue at first
        while (pendingNodes.size() != 0) {      // Runs until the queue is not empty
                TreeNode<int>* front = pendingNodes.front();  // stores front of queue
                pendingNodes.pop();         // deleted that front node stored previously
                cout << "Enter num of children of " << front->data << endl;
                int numChild;
                cin >> numChild;    // get the number of child nodes
                for (int i = 0; i < numChild; i++) {  // iterated over each child node to
                                                      //  input it
                        int childData;
                        cout << "Enter " << i << "th child of " << front->data << endl;
                        cin >> childData;
                        TreeNode<int>* child = new TreeNode<int>(childData);
                        front->children.push_back(child);   // Each child node is pushed
//into the queue as well as the list of child nodes as it is taken input so that next
// time we can take its children as input while we kept moving in the level-wise
// fashion
                        pendingNodes.push(child);
                }
```

```
        }
        return root;                // Finally returns the root node
}
```

Similarly, we can also print the child nodes using a queue itself. Now, try doing the same yourselves and for solution refer to the solution tab of the respective question.

Count total nodes in a tree

To count the total number of nodes in the tree, we will just traverse the tree recursively starting from the root node until we reach the leaf node by iterating over the vector of child nodes. As the size of the child nodes vector becomes 0, we will simply return. Kindly check the code below:

```
int numNodes(TreeNode<int>* root) {
        if(root == NULL) {                          // Edge case
                return 0;
        }
        int ans = 1;                                // To store total count
        for (int i = 0; i < root->children.size(); i++) {      // iterating over children vector
                ans += numNodes(root->children[i]);   // recursively storing the count
                                                      // of children's children nodes.
        }
        return ans;                 // ultimately returning the final answer
}
```

Height of the tree

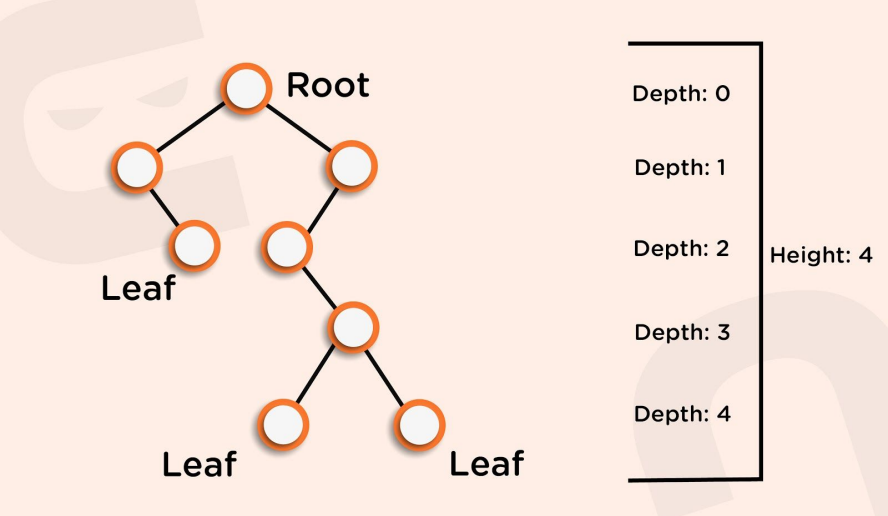
Height of a tree is defined as the length of the path from the tree's root node to any of its leaf nodes. Just think what should be the height of a tree with just one node? Well, there are a couple of conventions; we can define the height of a tree with just one node to be either 1 or zero. We will be following the convention where the height of a NULL tree is zero and that with only one node is one. This has been left

as an exercise for you, if need be you may follow the code provided in the solution tab of the topic corresponding to the same question.

Approach: Consider the height of the root node as 1 instead of 0. Now, traverse each child of the root node and recursively traverse over each one of them also and the one with the maximum height is added to the final answer along by adding 1 (this 1 is for the current node itself).

Depth of a node

Depth of a node is defined as it's distance from the root node. For example, the depth of the root node is 0, depth of a node directly connected to root node is 1 and so on. Now we will write the code to find the same... (Below is the pictorial representation of the depth of a node)



If you observe carefully, then the depth of the node is just equal to the level in which it resides. We have already figured out how to calculate the level of any node,using a similar approach we will find the depth of the node as well. Suppose, we want to find all the nodes at level 3, then from the root node we will tell its children to find the node that is at level 3 - 1 = 2, and similarly keep this up

recursively until we reach the depth = 0. Look at the code below for better understanding...

```
void printAtLevelK(TreeNode<int>* root, int k) {
        if(root == NULL) {                                  // Edge case
                return;
        }

        if(k == 0) {                                        // Base case: when the depth is 0
                cout << root->data << endl;
                return;
        }

        for(int i = 0; i < root->children.size(); i++) {    // Iterating over each child and
                printAtLevelK(root->children[i], k - 1);    // recursively calling with with 1
                                                            // depth less
        }
}
```

Count Leaf nodes

To count the number of leaves, we can simply traverse the nodes recursively until we reach the leaf nodes (the size of the children vector becomes zero). Following recursion, this is very similar to finding the height of the tree. Try to code it yourself and for the solution refer to the solution tab of the same.

Traversals

Traversing the tree is the manner in which we move on the tree in order to access all its nodes. There are generally 4 types of traversals in a tree:

- Level order traversal
- Preorder traversal
- Inorder traversal

- Postorder traversal

We have already discussed level order traversal. Now let's discuss the other traversals.

In Preorder traversal, we visit the current node first(starting with root) and then traverse the left sub-tree. After covering all nodes there, we will move towards the right subtree and visit in a similar manner. Refer the code below:

```
void preorder(TreeNode<int>* root) {
        if(root == NULL) {
            return;
        }
        cout << root->data << endl;
        for(int i = 0; i < root->children.size(); i++) {
            preorder(root->children[i].size());
        }
}
```

In postorder traversal, we visit all the child nodes first (from left to right order) and then we visit the current node. You will be coding this yourself (very similar to preorder traversal) and for solution, refer the solution tab in the corresponding questions.

We will study inorder traversal in further sections...

Destructor

Now, let's check the code to delete the tree that we left earlier. We will first-of-all delete the child nodes and then delete their root nodes and ultimately the main root node of the tree. If we simply delete the root node, then we will lose the references to its child nodes and hence only the root node will be deleted.

Kindly, refer to the code below for your reference:

```
void deleteTree(TreeNode<int> *root) {
        for(int i = 0; i < root->children.size(); i++) {
                deleteTree(root->children[i]);
        }
        delete root;
}
```

We will call this function in the main() and the tree will be deleted. But to make the code robust and easy to understand, we will be using destructors. We just want to call **delete root;** in the main() and it should delete the complete tree. Let's check the destructor part below:

```
~TreeNode() {
        for(int i = 0; i < children.size(); i++) {      // We will call delete on all its
                delete children[i];                     // children which will invoke
        } // corresponding destructor and ultimately delete the root node itself.
}
```

This destructor works in the same way as the recursive functions but is a better choice as code looks clean and easy-to interpret.

Vectors in C++

Introduction

Vectors are very similar to dynamic arrays for which we don't have to determine the size as done in case of arrays.

Vectors are sequence containers representing arrays that can change in size.

They can be used in the same way as the arrays and use the contiguous storage locations for storing data as in case of arrays. Just the difference is that these containers can change their size as per the requirements.

Dynamic implementation of arrays is used by vectors to store the elements with some initial size. In case of arrays, we need to reallocate the memory and then copy back these elements which is a time-expensive task. This is easily avoidable in case of vectors as these directly allocates some extra storage to accommodate the elements. There is a possibility that a vector's capacity is more than the number of elements present in it as it increases it's space using square function.

This is achieved by *amortized constant time* complexity which is internally achieved in C++.

Syntax for vector declaration:

```
vector<data_type> vector_name;
```

For example: To declare a vector of integer with a name v:

```
vector<int> v;
```

To dynamically allocate the same:

```
vector<int> *v = new vector<int>();
```

Header file used is : **#include<vector>**

Some important functions of vector:

push_back	Add element at the end
pop_back	Delete last element
insert	Insert elements
size	Return size
capacity	Return size of allocated storage capacity

Practice to implement a vector in C++

Statement: Perform the following tasks:

1. Create a vector of integer.
2. Insert the elements from 1 to 10 and print size along with capacity at each step.

Solution Code:

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

int main() {
        vector<int> v;                          // Part 1

        for(int i = 1; i <= 10; i++) {          // Part 2
                cout << v.capacity() << " " << v.size() << endl;
                v.push_back(i);
        }
        return 0;
}
```

Output:

```
0 0
1 1
2 2
4 3
4 4
8 5
8 6
8 7
8 8
16 9
```

Observation:

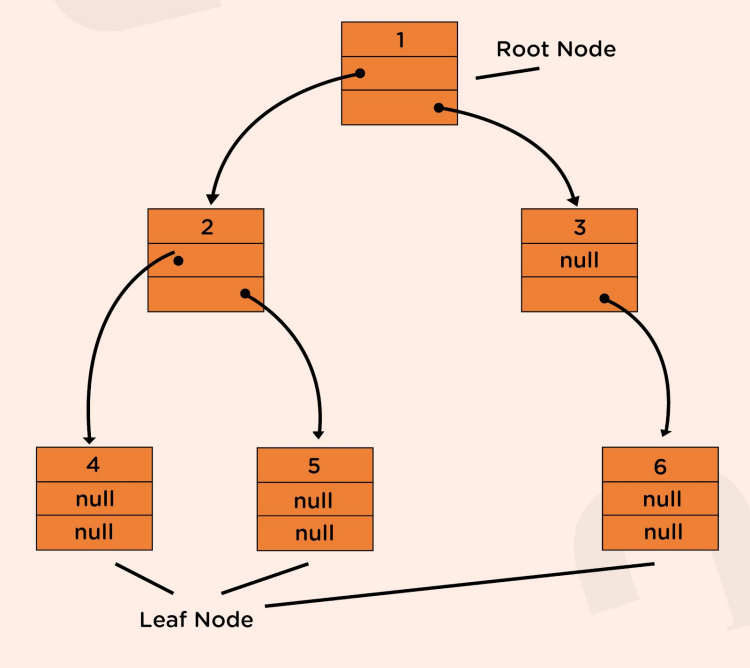
You can see that capacity doubles each time as the vector gets an extra element. But size represents the total number of elements present in the vector irrespective of its capacity.

Binary Trees

Introduction

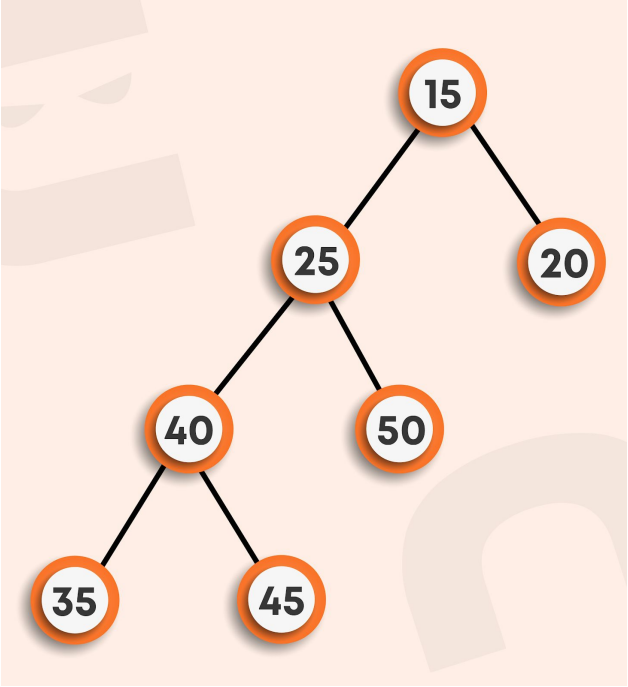
A generic tree with at most two child nodes for each parent node is known as a binary tree.

A binary tree is made of nodes that constitute a **left** pointer, a **right** pointer, and a data element. The **root** pointer is the topmost node in the tree. The left and right pointers recursively point to smaller **subtrees** on either side. A null pointer represents a binary tree with no elements, i.e., an empty tree. A formal definition is: a **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.



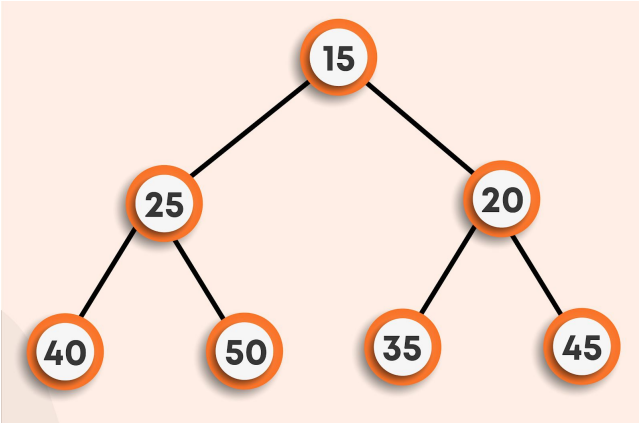
Types of binary trees:

- **Full binary trees:** A binary tree in which every node has 0 or 2 children is termed as a full binary tree.

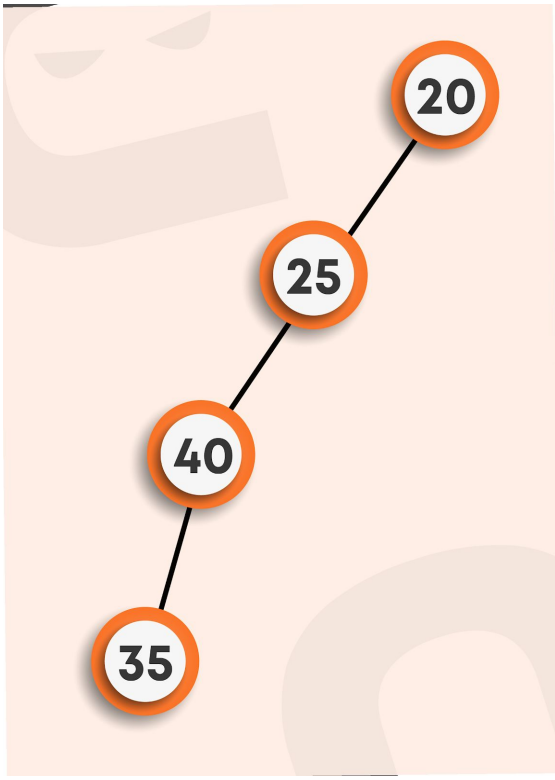


- **Complete binary tree:** A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.

- **Perfect binary tree:** A binary tree is termed perfect when all its internal nodes have two children along with the leaf nodes that are at the same level.

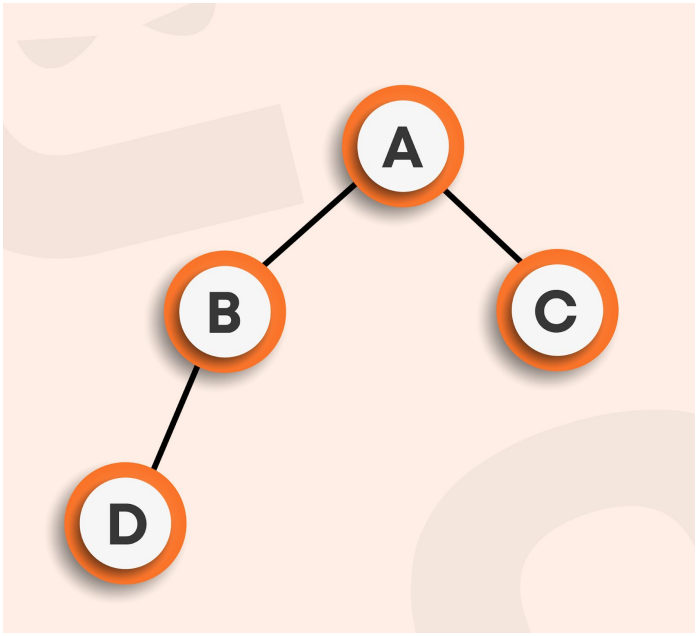


- **A degenerate tree:** In a degenerate tree, each internal node has only one child.



The tree shown above is degenerate. These trees are very similar to linked-lists.

- **Balanced binary tree:** A binary tree in which the difference between the depth of the two subtrees of every node is at most one is called a balanced binary tree.



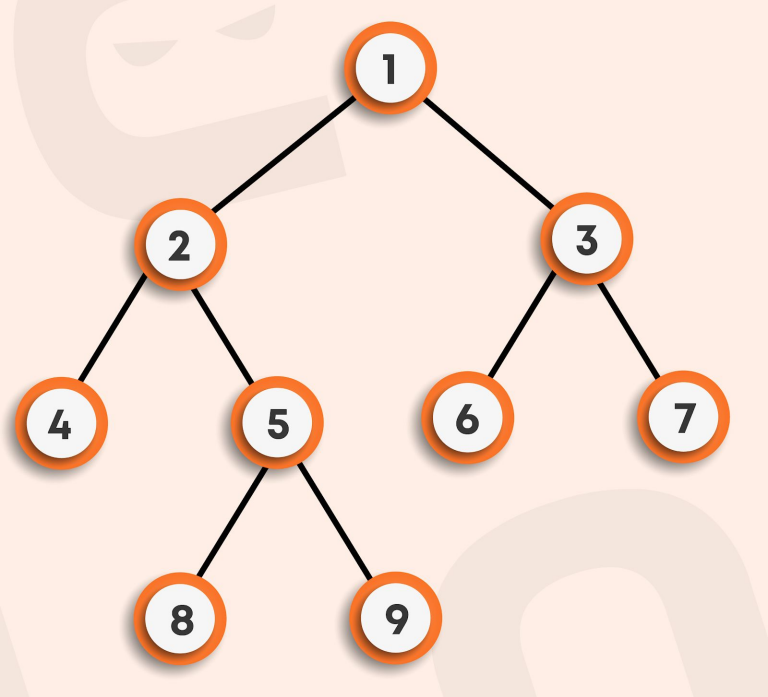
Binary tree representation:

Binary trees can be represented in two ways:

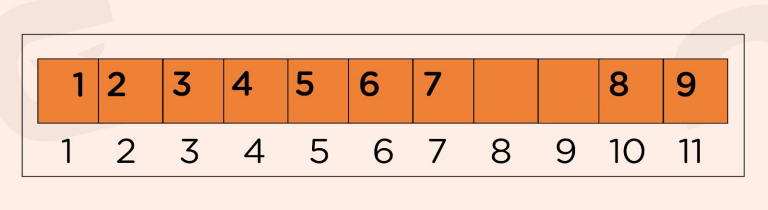
- **Sequential representation:** This is the most straightforward technique to store a tree data structure. An array is used to store the tree nodes. The number of nodes in a tree defines the size of the array. The root node of the tree is held at the first index in the array.

  In general, if a node is stored at the ith location, then its left and right child is kept at 2i and 2i+1 locations, respectively.

Consider the following binary tree:



The array representation of the above binary tree is as follows:



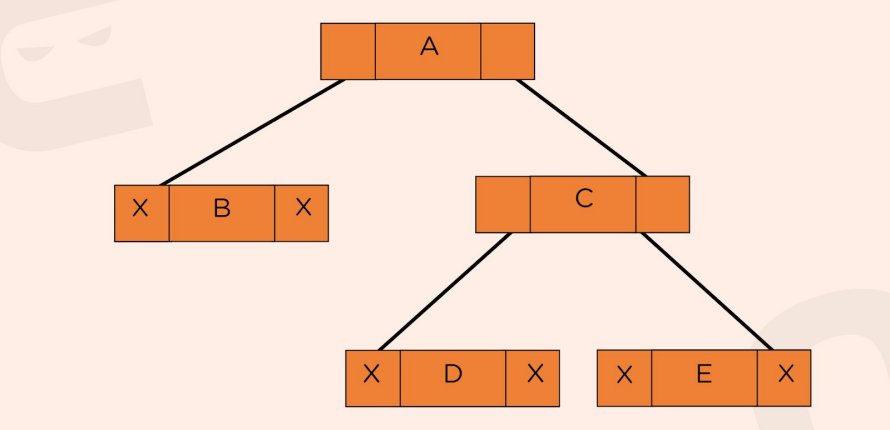
Here, we see that the left and right child of each node is stored at locations 2*(node_position) and 2*(node_position)+1, respectively.

For Example: The location of node 3 in the array is 3. So its left child will be placed at 2*3 = 6. Its right child will be at the location 2*3 +1 = 7. As we can see in the array, children of 3, which are 6 and 7, are placed at locations 6 and 7 in the array.

The sequential representation of the tree is not preferred due to the massive amount of memory consumption by the array.

Linked list representation: In this type of model, a linked list is used to store the tree nodes. The nodes are connected using the parent-child relationship like a tree.

The following diagram shows a linked list representation for a tree.



As shown in the above representation, each linked list node has three components:

- Left pointer
- Data part
- Right pointer

The left pointer has a pointer to the left child of the node; the right pointer has a pointer to the right child of the node whereas the data part contains the actual data of the node. If there are no children for a given node (leaf node), then the left and right pointers for that node are set to null . Let's now check the implementation of binary tree class. Like TreeNode class, here also we will be creating a separate file with **.h extension** and then use it wherever necessary. (Here the name of the file will be: **BinaryTreeNode.h**)

```
template <typename T>
```

```
class BinaryTreeNode {
        public:
        T data;                                 // To store data
        BinaryTreeNode* left;                   // for storing the reference to left pointer
        BinaryTreeNode* right;                  // for storing the reference to right pointer
        // Constructor
        BinaryTreeNode(T data) {
                this->data = data;              // Initializes data of the node
                left = NULL;                    // initializes left and right pointers to NULL
                right = NULL;
        }
        // Destructor
        ~BinaryTreeNode() {
                delete left;                    // Deletes the left pointer
                delete right;                   // Deletes the right pointer
        }                                       // As it ends, deletes the node itself
};
```

Take input and print recursively

Let's first check on printing the binary tree recursively. Follow the comments in the code below...

```
void printTree(BinaryTreeNode<int>* root) {
        if (root == NULL) {                     // Base case
                return;
        }
        cout << root->data << ":";  // printing the data at root node
        if (root->left != NULL) {     // checking if left not NULL, then print it's data also
                cout << "L" << root->left->data;
        }

        if (root->right != NULL) { // checking if right not NULL, then print it's data also
                cout << "R" << root->right->data;
        }
        cout << endl;
```

```
        printTree(root->left);    // Now recursively, call on the left and right subtrees
        printTree(root->right);
}
```

Now, let's check the input function: (We will be following the level-wise order for taking input and -1 denotes the NULL pointer or simply, it means that a pointer over the same place is a NULL pointer.

```
BinaryTreeNode<int>* takeInput() {
        int rootData;
        cout << "Enter data" << endl;
        cin >> rootData;                        // taking data as input
        if (rootData == -1) {                   // if the data is -1, means NULL pointer
                return NULL;
        }
       // Dynamically create the root Node which calls constructor of the same class
        BinaryTreeNode<int>* root = new BinaryTreeNode<int>(rootData);
       // Recursively calling over left subtree
        BinaryTreeNode<int>* leftChild = takeInput();
       // Recursively calling over right subtree
       BinaryTreeNode<int>* rightChild = takeInput();
        root->left = leftChild;     // now allotting left and right childs to the root node
        root->right = rightChild;
        return root;
}
```

Taking input iteratively

We have already discussed a recursive approach for taking input in a binary tree in a level order fashion. Now, let's discuss the iterative procedure for the same using queue as we did in Generic trees. Follow the code along with the comments...

```
BinaryTreeNode<int>* takeInputLevelWise() {
```

```
        int rootData;
        cout << "Enter root data" << endl;
        cin >> rootData;                                    // Taking node's data as input
        if (rootData == -1) {                               // As -1 refers to NULL pointer
                return NULL;
        }
       // Dynamic allocation of root node
        BinaryTreeNode<int>* root = new BinaryTreeNode<int>(rootData);
       // Using queue to level wise traversal for iterative approach
        queue<BinaryTreeNode<int>*> pendingNodes;
        pendingNodes.push(root);                // root node pushed into the queue
        while (pendingNodes.size() != 0) {      // process continued until the size of queue ≠ 0
                BinaryTreeNode<int>* front = pendingNodes.front();//front of queue is stored
                pendingNodes.pop();
                cout << "Enter left child of " << front->data << endl;
                int leftChildData;
                cin >> leftChildData;       // Left child of current node is taken input
                if (leftChildData != -1) {    // If the value of left child is not -1 that is, not NULL
                        BinaryTreeNode<int>* child = new BinaryTreeNode<int>(leftChildData);
                        front->left = child;            // Value assigned to left part and then pushed
                        pendingNodes.push(child);   // to the queue
                }
               // Similar work is done for right subtree
                cout << "Enter right child of " << front->data << endl;
                int rightChildData;
                cin >> rightChildData;
                if (rightChildData != -1) {
                       BinaryTreeNode<int>* child = new BinaryTreeNode<int>(rightChildData);
                        front->right = child;
                        pendingNodes.push(child);
                }
        }
        return root;
}
```

Count nodes

Unlike the Generic trees, where we need to traverse the children vector of each node, in binary trees, we just have at most left and right children for each node. Here, we just need to recursively call on the right and left subtrees independently with the condition that the node pointer is not NULL. Kindly follow the comments in the upcoming code...

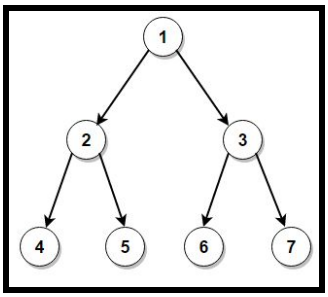
```
int numNodes(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // Condition to check if the node is not NULL
                return 0;                       // counted as zero if so
        }
        return 1 + numNodes(root->left) + numNodes(root->right);   // recursive calls
            // on left and right subtrees with addition of 1(for counting current node)
}
```

Binary tree traversal

Following are the ways to traverse a binary tree and their orders of traversal:

- **Level order traversal**: Moving on each level from left to right direction
- **Preorder traversal** : ROOT -> LEFT -> RIGHT
- **Postorder traversal** : LEFT -> RIGHT-> ROOT
- **Inorder traversal** : LEFT -> ROOT -> RIGHT

Some examples of the above-stated traversal methods:



- **Level order traversal:** 1, 2, 3, 4, 5, 6, 7
- **Preorder traversal:** 1, 2, 4, 5, 3, 6, 7
- **Post order traversal:** 4, 5, 2, 6, 7, 3, 1
- **Inorder traversal:** 4, 2, 5, 1, 6, 3, 7

Let's look at the code for inorder traversal, below:

```
void inorder(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // Base case when node's value is NULL
                return;
        }
        inorder(root->left);                   //Recursive call over left part as it needs
                                               // to be printed first
        cout << root->data << " ";             // Now printed root's data
        inorder(root->right);      // Finally a recursive call made over right subtree
}
```

Now, from this inorder traversal code, try to code preorder and postorder traversal yourselves. For the answer, refer to the solution tab for the same.

Construct a binary tree from preorder and inorder traversal

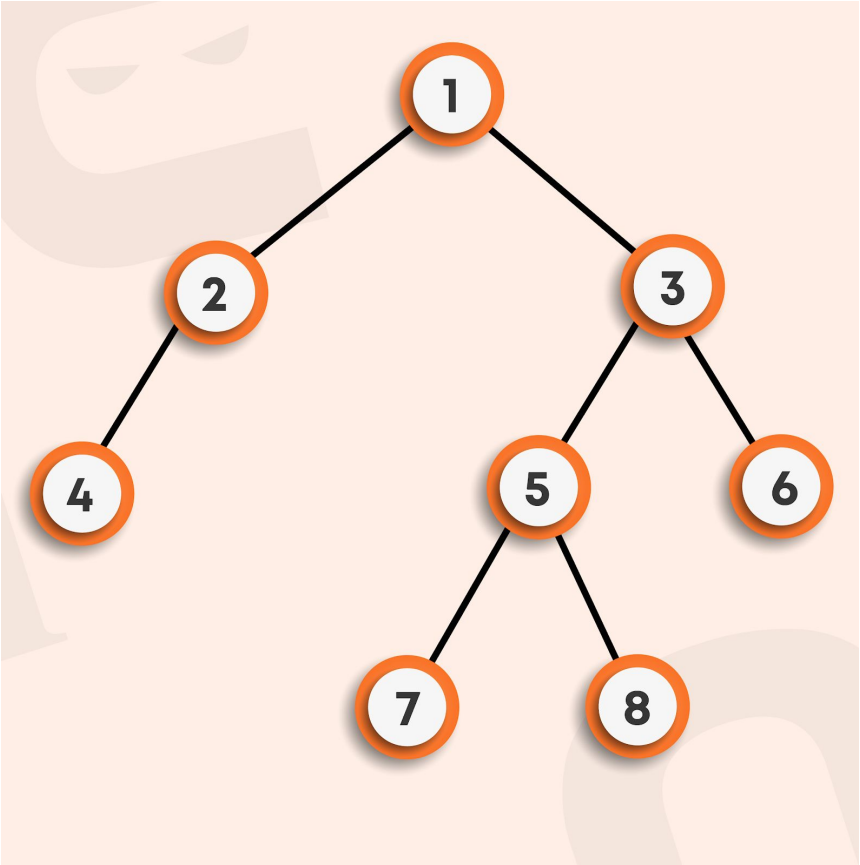
Consider the following example to understand this better.

Input:

Inorder traversal : {4, 2, 1, 7, 5, 8, 3, 6}

Preorder traversal : {1, 2, 4, 3, 5, 7, 8, 6}

Output: Below binary tree...



The idea is to start with the root node, which would be the first item in the preorder sequence and find the boundary of its left and right subtree in the inorder array. Now all keys before the root node in the inorder array become part of the left subtree, and all the indices after the root node become part of the right subtree. We repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider below inorder and preorder sequence-

Inorder: {4, 2, 1, 7, 5, 8, 3, 6}

Preorder: {1, 2, 4, 3, 5, 7, 8, 6}

The root will be the first element in the preorder sequence, i.e. 1. Next, we locate the index of the root node in the inorder sequence. Since 1 is the root node, all

nodes before 1 must be included in the left subtree, i.e., {4, 2}, and all the nodes after one must be included in the right subtree, i.e. {7, 5, 8, 3, 6}. Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Left subtree:	**Right subtree:**
Inorder : {4, 2}	**Inorder :** {7, 5, 8, 3, 6}
Preorder : {2, 4}	**Preorder :** {3, 5, 7, 8, 6}

Follow the above approach until the complete tree is constructed. Now let us look at the code for this problem:

```
BinaryTreeNode<int>* buildTreeHelper(int* in, int* pre, int inS, int inE, int preS, int preE) {
        if (inS > inE) {                        // Base case
                return NULL;
        }

        int rootData = pre[preS];   // Root's data will be first element of the preorder array
        int rootIndex = -1;             // initialised root's index to -1 and searched for it's value
        for (int i = inS; i <= inE; i++) {      // in inorder list
                if (in[i] == rootData) {
                        rootIndex = i;
                        break;
                }
        }
       // Initializing the left subtree's indices for recursive call
        int lInS = inS;
        int lInE = rootIndex - 1;
        int lPreS = preS + 1;
        int lPreE = lInE - lInS + lPreS;
      // Initializing the right subtree's indices for recursive call
        int rPreS = lPreE + 1;
        int rPreE = preE;
        int rInS = rootIndex + 1;
        int rInE = inE;
      // Recursive calls follows
        BinaryTreeNode<int>*  root = new BinaryTreeNode<int>(rootData);
```

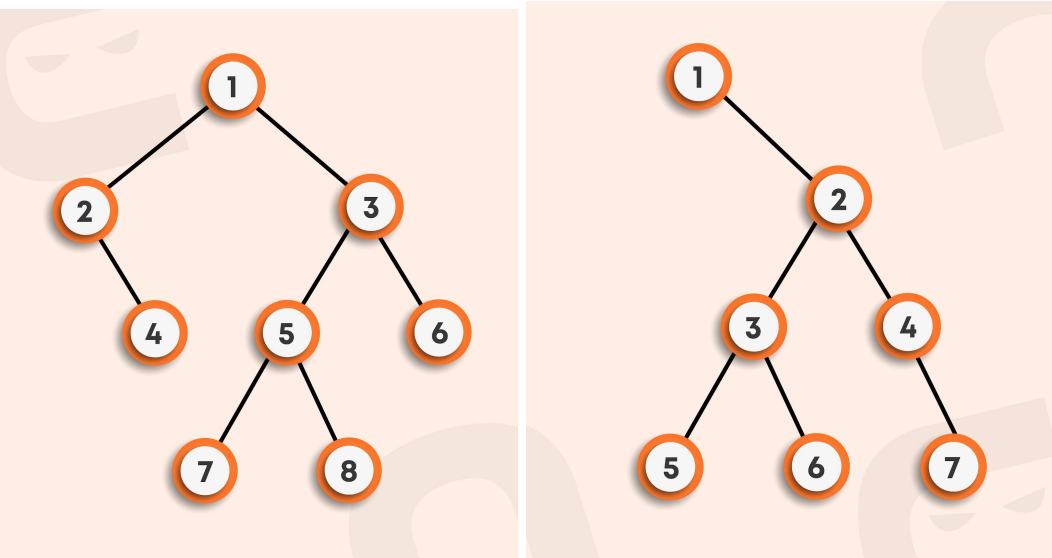
```
        root->left = buildTreeHelper(in, pre, lInS, lInE, lPreS, lPreE);    // over left subtree
        root->right = buildTreeHelper(in, pre, rInS, rInE, rPreS, rPreE); // over right subtree
        return root;    // finally returned the root
}
BinaryTreeNode<int>* buildTree(int* in, int* pre, int size) {    // this is the function called
 // from the main() with inorder and preorder traversals in the form of arrays and their
// size which is obviously same for both
        return buildTreeHelper(in, pre, 0, size - 1, 0, size - 1);  // These arguments are of the
   // form (inorder_array, preorder_array, inorder_start, inorder_end, preorder_start,
    // preorder_end) in the helper function for the same written above.
}
```

Now, try to construct the binary tree when inorder and postorder traversals are given...

The diameter of a binary tree

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two child nodes. The diameter of the binary tree may pass through the root (not necessary).

For example, the Below figure shows two binary trees having diameters 6 and 5, respectively (nodes highlighted in blue color). The diameter of the binary tree shown on the left side passes through the root node while on the right side, it doesn't.



There are three possible paths of the diameter:

1. The diameter could be the sum of the left height and the right height.
2. It could be the left subtree's diameter.
3. It could be the right subtree's diameter.

We will pick the one with the maximum value.

Now let's check the code for this...

```
int height(BinaryTreeNode<int>* root) {   // Function to calculate height of tree
        if (root == NULL) {
                return 0;
        }
        return 1 + max(height(root->left), height(root->right));
}

int diameter(BinaryTreeNode<int>* root) {   // Function for calculating diameter
        if (root == NULL) {                 // Base case
                return 0;
        }

        int option1 = height(root->left) + height(root->right);   // Option 1
        int option2 = diameter(root->left);                       // Option 2
        int option3 = diameter(root->right);                      // Option 3
```

```
        return max(option1, max(option2, option3));   // returns the maximum value
}
```

The time complexity for the above approach:

- Height function traverses each node once; hence time complexity will be O(n).
- Option2 and Option3 also traverse on each node, but for each node, we are calculating the height of the tree considering that node as the root node, which makes time complexity equal to O(n*h). (worst case with skewed trees, i.e., a type of binary tree in which all the nodes have only either one child or no child.) Here, h is the height of the tree, which could be $O(n^2)$.

This could be reduced if the height and diameter are obtained simultaneously, which could prevent extra n traversals for each node. To achieve this, move towards the other sections...

The Diameter of a Binary tree: Better Approach

In the previous approach, for each node, we were finding the height and diameter independently, which was increasing the time complexity. In this approach, we will find height and diameter for each node at the same time, i.e., we will store the height and diameter using a pair class where the **first** pointer will be storing the height of that node and the **second** pointer will be storing the diameter. Here, also we will be using recursion.

Let's focus on the **base case**: For a NULL tree, height and diameter both are equal to 0. Hence, pair class will store both of its values as zero.

Now, moving to **Hypothesis**: We will get the height and diameter for both left and right subtrees, which could be directly used.

Finally, the **induction step**: Using the result of the Hypothesis, we will find the height and diameter of the current node:

Height = max(leftHeight, rightHeight)

Diameter = max(leftHeight + rightHeight, leftDiameter, rightDiameter)

Note: C++ provides an in-built pair class, which prevents us from creating one of our own. The Syntax for using pair class:

```
pair<datatype1, datatype2> name_of_pair_class;
```

For example: To create a pair class of int, int with a name p, follow the syntax below:

pair<int, int> p;

To access this pair class, we will use **.first** and **.second** pointers.

Follow the code below along with the comments to get a better grip on it...

```
pair<int, int> heightDiameter(BinaryTreeNode<int>* root) {
                                                    // pair class return-type function
        if (root == NULL) {          // Base case
                pair<int, int> p;
                p.first = 0;
                p.second = 0;
                return p;
        }
        // Recursive calls over left and right subtree
        pair<int, int> leftAns = heightDiameter(root->left);
        pair<int,int> rightAns = heightDiameter(root->right);
        // Hypothesis step
        // Left diameter, Left height
        int ld = leftAns.second;
        int lh = leftAns.first;
        // Right diameter, Right height
        int rd = rightAns.second;
        int rh = rightAns.first;

       // Induction step
        int height = 1 + max(lh, rh);                    // height of current root node
        int diameter = max(lh + rh, max(ld, rd));  // diameter of current root node
        pair<int, int> p;                                // Pair class for current root node
```

```
        p.first = height;
        p.second = diameter;
        return p;
}
```

Now, talking about the time complexity of this method, it can be observed that we are just traversing each node once while making recursive calls and rest all other operations are performed in constant time, hence the time complexity of this program is O(n), where n is the number of nodes.

Practice problems:

- https://www.hackerrank.com/challenges/tree-top-view/problem
- https://www.codechef.com/problems/BTREEKK
- https://www.spoj.com/problems/TREEVERSE/
- https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-trees/practice-problems/approximate/largest-cycle-in-a-tree-9113b3ab/