Ans. to the que. no.1

To describe a circle and a rectangle by referencing the concepts of OOP, we use **Polymorphism** which is an important feature of OOP.

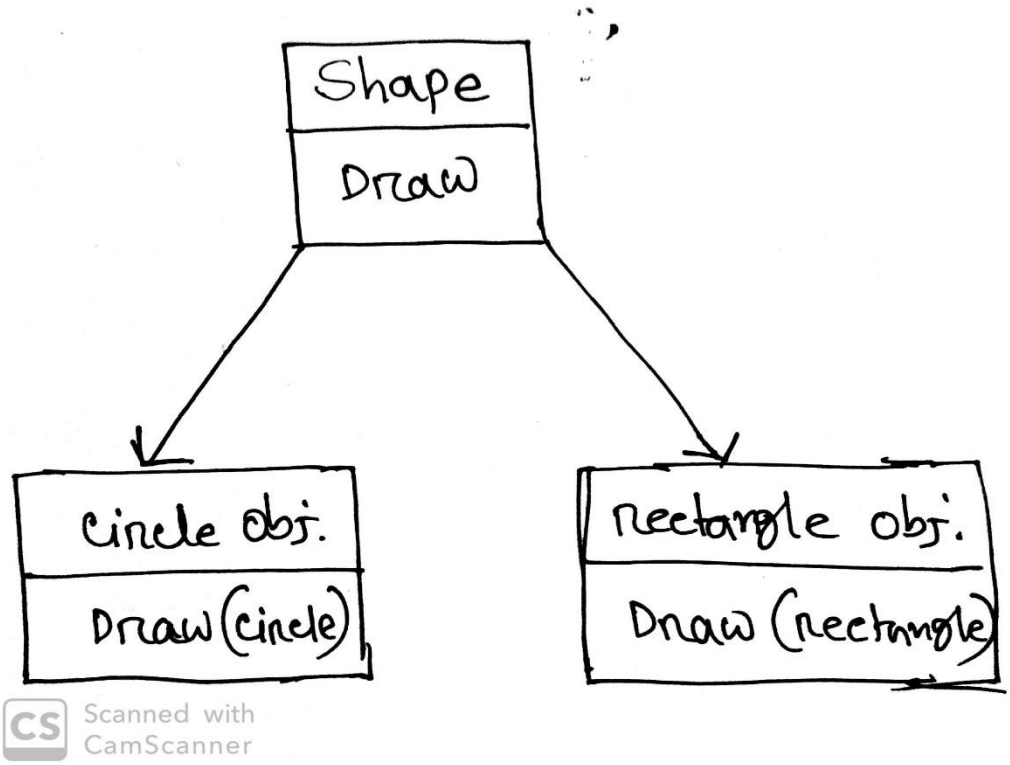

Fig.1: Polymorphism

Fig.1 shows how a single function name may be used to handle a variety of numbers and types of arguments. This is analogous to how a single word may have multiple different meanings depending on context. Function overloading occurs when a single function name is used to execute several tasks which is a one of the task of polymorphism.

The above figure shows a Rectangle and a Circle. Both of them have a shape. Here, Polymorphism is extensively used in implementing inheritance.

Runtime polymorphism and ways to achieve it:

Runtime polymorphism, also known as dynamic polymorphism or late binding, is a kind of polymorphism. The function call is resolved at runtime under runtime polymorphism.

Runtime polymorphism occurs when an object is associated with functionality during runtime. **Function overriding** can be used to provide runtime polymorphism. It is the behavior when we have two classes parent class/superclass and child class/sub-class with same method signature but different implementation.

when we call the method using the reference of superclass and call the method, then the method of subclass will be called.

```
superClass s = new SubClass();
s.someMethod();        // this will call the method of subclass.
```

It is called runtime polymorphism because the method to run is only known at the runtime as opposed to compile time polymorphism, in which it is decided at compile time.

Ans. to the que. no.2

Stack:

It's a type of transient memory. The memory of the variable will be automatically deleted after the computational operation is completed. Methods, local variables, and reference variables are primarily found in the stack section.

The allocation and deallocation for stack memory is automatically done.

Heap:

The heap is a kind of memory that computer languages employ to keep track of global variables. All global variables are kept in heap memory space by default. It has the ability to allocate memory dynamically.

The heap isn't handled automatically for you, and the CPU doesn't have as much control over it. It's more like a memory's free-floating area.

Unlike stack memory, heap memory is allocated explicitly by programmers and it won't be deallocated until it is explicitly freed.

For a very large array (100 MB), I want to allocate memory on heap with malloc because stack can't hold that much data. I have to allocate big arrays on the heap as follows:

```
int *array = malloc (sizeof(int)*size);
```

Ans. to the que. no.3

If I want to perform element-wise matrix multiplication, i don't prefer the two for-loop approach because time complexity of matrix multiplication is **O(n^3) using normal matrix multiplication**.

I do prefer **Strassen algorithm** which improves it and its time complexity is **O(n^(2.8074)).** It uses the divide and conquer techniques to perform multiplication. It is faster than the standard matrix multiplication algorithm for large matrices, with a better asymptotic complexity, although the naive algorithm is often better for smaller matrices.

by using Multi-threading the improvement of the performance can be done.

Ans. to the que. no.4

Given recursive approach is seen below:

```
struct Node
{
Node* left;
Node* right;
}
int traverse(Node* node){
//base case
 if (node == null)
return 0;
 count++;
count += traverse (node->left) //recursive call
count += traverse (node->right) //recursive call
return count;
 }
```

This recursive approach is not going to work because,

Here in the traverse method, there has no presence of the **recursive case.** That's why the recursive call has no value.

The alternative ways to count the nodes is seen below:

```
struct Node
{
   int data;
   struct Node* left, *right;
};


unsigned int traverse(struct Node* root)
{
   if (root == NULL)
      return 0;

   int res = 0;
   if (root->left && root->right)
      res++;

   res += (traverse(root->left) +
         traverse(root->right));
   return res;
}


struct Node* newNode(int data)
{
   struct Node* node = new Node;
   node->data = data;
   node->left = node->right = NULL;
   return (node);
}
```

Programming Questions Answer

1.BAD_URLS

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
	string st;
	cin >> st;
	string ans;
	for (int i = 0; i < st.size(); i++)
	{
		if (st[i] == '/')
		{
			if (ans.size() > 0 && ans[ans.size() - 1 ] != '/' )
				ans.push_back(st[i]);
		}
		else ans.push_back(st[i]);
	}
	cout << ans << endl;
 return 0;
}
```

Thought process: We will use extra space to store the resulting string. If we find a new ‘/’ , we will check whether the resulting string’s last character is ‘/’ or not. If the last character is ‘/’, we don’t need to push another ‘/’ to the resulting string. Thus we can remove repeating ‘/’ from the given string.

2.BIG_NUMBERS

```
#include<bits/stdc++.h>
using namespace std;

string summation (string num1, string num2)
{

   if (num1.size() > num2.size())
   {
      reverse(num2.begin(), num2.end());
      while (num1.size() > num2.size()) num2.push_back('0');
      reverse(num2.begin(), num2.end());
   }
   else
   {
      reverse (num1.begin(), num1.end());
      while (num2.size() > num1.size()) num1.push_back('0');
      reverse (num1.begin(), num1.end());
```

```
    }
    int carry = 0;
    string sum;
    for (int i = num1.size() - 1; i >= 0; i--)
    {
        int val = num1[i] - '0' + num2[i] - '0' + carry;
        if (val > 9)
        {
            sum.push_back(val - 10 + '0');
            carry = 1;
        }
        else
        {
            sum.push_back (val + '0');
            carry = 0;
        }
    }
    if (carry == 1) sum.push_back('1');
    reverse(sum.begin(), sum.end());

    return sum;
}
int main()
{
    string num1, num2;
```

```
    cin >> num1 >> num2;
    string sum = summation(num1, num2);
    cout << sum << endl;
    return 0;
}
```

Thought process: We need to store two big integers and their summation in string. We have to add these values in the way we used to add two numbers in primary school. We will start from the rightmost digits of the numbers one by one and add them with carry value. Set the carry value 1 if summation of two digits is greater or equal than 10 and carry this to the next same operation.