→ Function :-

As we can assume by its name function is made to do some work.

We use/define function to structure and execute coad in better way.

Beside name of every function paranthesis ('()') is must.

Actully main() is also a function.

We can define function in two ways generally.

1. Directly defining function before main() and then using it in main.

ex

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

int sum(int a, int b)
{
    int c=a+b;
    return c;
}

int main()
{
    int num1, num2;
    cout<<"enter two integers "<<endl;
    cin>>num1>>num2;
    cout<<" sum is" << sum(num1, num2)
                    <<endl;

    return 0;
}
```

2. By function prototyping :-

By function prototyping you ~~create~~ declair prototype of function before main() but define

actual function after main.
It's make looks and structure of code better.

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

int sum(int a, int b);

int main()
{
    int num1, num2;
    cout<<"Two integers"<<endl;
    cin>>num1>>num2;
    cout<<"sum is"<<sum(num1, num2
                <<endl;

    return 0;
}

int sum(int a, int b)
{
    int c = a+b;
    return c;
}
```

NOTE:- Actual parameters are those parameters which are passed in function, for example num 1 & num 2 are actual parameters.

Formal parameters are those parameters which takes value from actual parameters, for example a & b are formal parameters.

→ Call by Value and Call by Refrence :-

In call by value, we pass value at the place of actual parameters.

It is easy to use but not work in all conditions like in swaping values from another function we can't use call by value it is better to use

call by reference in that case.

When we pass address of variables in place of actual parameters then it is called call by refrence.

There is two way to use call by reference.

1. By pointers:-

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

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main()
{
    int x, y;
```

```
cout<<"value of x & y is"<<endl;
cin>>x>>y;

cout<<"before swapping"<<x<<endl
                       <<y<<endl;

swap(&x, &y);
cout<<"after swapping"<<x<<endl
                      <<y<<endl;

return 0;
}
```

2. By reference variable.

As we know reference variables are variable referencing to other/different variable.

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

void swap(int &a, int &b)
{
    int temp=a;
    a=b;
```

```
    b = temp;
}

int main()
{
    int x, y;
    x = 10;
    y = 5;
    cout << "before swap" << x << y
                                << endl;
    swap(x, y);
    cout << "after swap" << x << y
                                << endl;
    return 0;
}
```

⇒ Goto statement in C :-

Goto is special statement in C, it is also called jump statement.

When you call go to statement it immediately stops all loops / function and go to that function / pointer where it is asked to go.

It is very important when you want to stop all loops together, unlike break which stops only one loop.

It ~~can very~~ should be used very carefully, because it makes program me very complicated to understand.

Syntax :-

```
goto end;
end:
    statements;
```

→ Inline function :-

When we call another function in main function, then first, compiler go to that function copy arguments if available, then do work then return value, this work takes time, it can be reduced if we call that fuction in main, compiler just copy & paste that function in the place where it is called.

We can do so, if that new function is inline function.

Inline function is type of function which when called, don't copy actual argument to formal argument but function itself copy to that place from where it is called, it reduces time.

But we should not make big function inline function because

pasting whole function from one to another point takes much more time than it takes when we call a normal function, so it is practical to make small function, (having one or two line of code) inline function.

Also making a function an inline function is request of user to compiler to make it inline, compiler may or may not accept it.

Sytex

inline function name;

ex.-

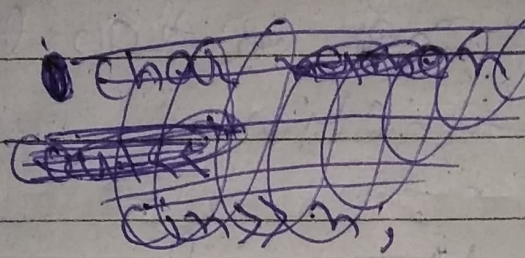
```
#include<iostream>
using namespace std;
inline void name();
int main()
{
```

cout<<"My name is"<<name();

}

inline void name()

{



cout<<"Anu";

}

→ Static variable :-

When we assign a variable as static, it means initialization of that variable only occurs one, also if it is in loop.

Also, variable memorise the previous value and if you come next time (in next iteration) it gives you that previous value.

Also note the by default static variable has value '0' in it, so we don't need to initialize it to zero.

Satic variable is useful in loops and calling of function when value of other variable changes when loops iterates next or function is called but you get that value you have given to him in previous iteration / function call.

ex. –

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

int product(int a, int b)
{
    static int c=0;
    c= c+1;
    return a*b+c;
}
int main()
{
    int a,b;
    cin>>a>>b;
for(int i=0;i<5;i++)
{
```

```
cout<<"Result from a & b is"<<
                    product(a,b)<<endl;

return 0;
}
```

→ Default argument :-

We have already learn about actual & formal argument. Here we will learn abaut default argument.

Let you called a function and forgot to pass actual arguments then programm didn't work as function needed actual argument so that it copy it in formal argument, we can solve this problem by passing value with formal argument. This value will work as default argument while function call we forgot to pass

actual argument.
If we pass actual argument then this default argument will not be used and actual argument will be used.

ex.-

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

int substract(int a, int b=0)
{
    signed int c;
    c = a-b;
    return c;
}

int main()
{
    int a,b;
    cout<<"Input two integers"<<endl;
    cin>>a>>b;
```

```
cout<<"Diff. btw"<<a<<"and"<<b<<"is"<<
        Substract(a)<<endl;

cout<<"Diff. btw"<<a<<"and"<<b<<"is"<<
        Substract(a,b)<<endl;

return 0;
}
```

Here note that in first cout we have given only one argument so, function will use default argument but in 2nd count we have given two argument so, function will use these two and will replace that default argument.

→ Constant arguments :-

We use constant arguments in a function when dont want that someone change value of those arguments we generally do that in case of reference or

const char

pointer variable.

ex.-

int len(const char *p);

It is declaration of len() function ~~whic~~ with constant argument.

→ Recursion & recursive function:-

Recursion is a process in which a function calls it self and the function doing this is called recursive function.

Recursion sometimes helps us to solve complication question which when we solve by loops takes much time but it can't be used in every place as its mechanism is hard to understand sometimes causes complications in programme.

Also when we use recursive function it calls itself until the base case not reached if the data is big or function is calling itself many times it gives overhead to programme and programmer. so, it is important to see, when we can use recursive approach & when we can use iterative approach.

ex.-
Finding factorial of any number by recursion.

```
#include<iostream>
using namespace std;
int factorial(int a)
{
    if(a==1 || a==0)
    {
        return 1;
    }
```

```
    return a * factorial(a-1);
}
int main()
{
    int a;
    cout<<" input number of which you
          want to find factorial"<<
                                endl;
    cin>>a;
    cout<<" factorial of"<<a<<" is "<<
                factorial(a) <<
                                endl;
    return 0;
}
```

→ Function overloading :-

Function overloading happens when there are more than one function with same name & you are calling one of them or all of them.

When such thing happens, it is called function overloading.

Now, question arises how compilar understand which function we are calling.

To know, which function we are calling compilar first matches no. of argument, if no. of arguments is matched then it matches type of data inputed and type of data asked in function, if it maches then ok, if not, compilar tries to change data type of input arguments to match the data type of formal arguments.

Q.-

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

int sum(int a, int b)
{
    return a+b;
}

int sum(int a, int b, int c)
{
    return a+b+c;
}

int main()
{
    int a,b,c;
    cout<<"input three integers"<<
                                endl;
    cin>>a>>b>>c;
    cout<<"sum of "<<a<<"and"<<b
    <<"is"<<sum(a,b)<<endl;
```

```
cout<<"sum of" <<a <<"and" <<b <<"and" <<c
    <<"is" <<sum(a, b, c) <<endl;

return 0;
}
```

Q. WAP to calculate volume of cubiod, cylinder and cube by one function only. (using function overloading)

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

float vol(int a)
{
  return (a*a*a);

}
float vol(int a, int b, int h);
{
   return (a*b*h);
}
float vol(int r, int h)
{
   return (3.14*r*r*h);
}
```

```
int main()
{
  int a, l, b, h1, h2, r;

  cout<<" input side of cube"<<endl;
  cin>>a;
  cout<<" input dimension of cuboid"<<
                                endl;

  cin>> l >>b>>h1;

  cout<<" input radius and height of
        cylinder"<<endl;
  cin>> r>>h2;

  cout<<" Volume of cube is"<<
        vol(a)<<endl;

  cout<<" Volume of cuboid is"<<
        vol(l,b,h1)<<end;

  cout<<" Volume of cylinder is"<<
        vol(r,h2)<<endl;

  return 0;
}
```