C++ FOR PROBLEM SOLVING

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    cout << "Hello";
    return 0;
}
```

→ contains all internal headers so while using vector, map etc. we don't have to import them separately

→ helps us execute operators like cin, cout

C namespaces are areas where certain keywords, variables, operators are defined for usage)

main fn() where the execution of the program begins

operator used to output. (Here we are outputting a string which is why we put it in " ")

* More Usage

→ `cin>>a;`

This statement is taking an input in variable a in runtime i.e you can interact with console in real time to provide input.

→ `cout << '\n';` ← works faster than endl

This is to print a new line. i.e. if you want to output "Hello" and integer a=5. if you don't include '\n' then the output will look something like (Hello5) or you can include a new line then the 5 will come in a new line. i.e,

Hello
5

→ `cout << endl;` ← its slower bcz it flushes out the entire output buffer.

endl stands for end line. It also acts like '\n'

→ There can be many other functions than main. Ex:

does not return anything to function call in main →

```
void f(int a, int b){
    cout << a+b << '\n';
}
```

// Here we are simply outputting the addition of a & b.

```
int main(){ // main is taking input of a and b and passing to function f
    int a, b;
    cin >> a >> b;
    f(a, b); // function call with parameter a, b.
    return 0;
}
```

First call → Main() —main() calls→ f()

OR we can update the function to return the value of addition to the function call.

Since now we are returning an integer →

```
int f(int a, int b){
    return a+b; ← value is calculated & returned
}
int main(){
    int a, b;
    cin >> a >> b;
    cout << f(a,b); ← Here, we are outputting the result
    return 0;
}
```

- What will be the effect if we change the value of variables a and b in function f? Will the change reflect in main()?

```
int f (int a, int b) {
    a++;        // a = 6
    b++;        // b = 7
    return a+b;     // 13
}
int main() {
    int a, b;
// a=5, b=6  cin>>a>>b;
    cout<<f(a,b);
    cout<<a<<b;
// 5+6=12  cout<<a+b<<'\n';
    return 0;
}
```

This is pass by value which means that when the function call provide the parameters, the function f creates its own copy which is then used for computation. It is as if the fxn f has its own set of a & b.

a+b ≠ f(a,b)
12 ≠ 13

→ Pass by Reference If you want changes in any of your variables anywhere in the program to be consistent (i.e, changes in f() should reflect in main())

Then we use call by Reference method. Here, you are basically passing & sharing the memory location of variables a & b. It is a single memory

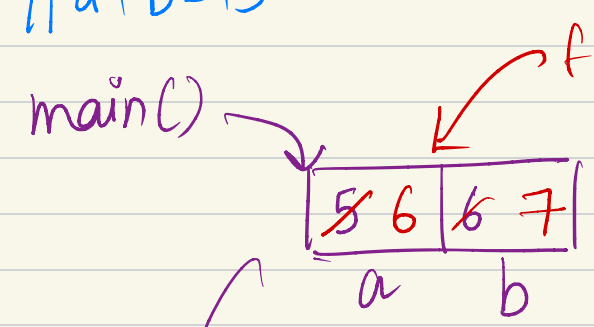
location with pointers from main() & f()

Implementation:

Passing variables by reference.

```
int f(int &a, int &b)
    a++;        //a=6
    b++;        //b=7
    return a+b; //a+b=13
}
int main()
{
    int a,b;
//a=5, b=6  cin>>a>>b;
//f(a,b)=13 cout<<f(a,b);
//a+b=13    cout<<a+b;
}
```

6 7



shared variable memory, helping to reflect changes.

→ Call by Pointers

A pointer is used to store address of another variable. Ex: int *a = &b.

```
int f(int *a, int *b){
    cout<<*a<<" "<<*b<<'\n';
    return *a+*b;
}
```

Pointers storing addresses

How to access value of a & b?

→ using dereferencing operator

accessing value of a & b

outputting the addition

→ Functional Scope : A variables life ends when the function returns or ends i.e a T in f() is not a variable in another function g(). T can only be accessed by f() as it is declared inside it.

→ Local Scope

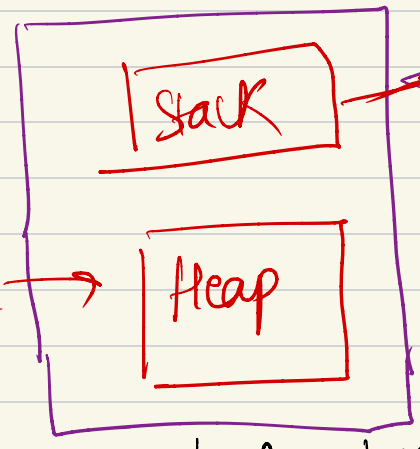
Simple closing / opening brackets inside a function

```
int main() {
    {
        int c, d;
        cin>>c>>d;
    }
    cout<<"Hello";
    cout<<c+d<<'\n';   // ERROR: c & d don't exist here
    return 0;
}
```

Local Scope, i.e, c, d can't be accessed outside of these brackets

---

→ Primarily there are 2 regions of RAM



Stack whatever local variables are defined are assigned memory here in the stack.

variable value defined dynamically during Runtime is stored in Heap - We have to clean heap ourselves.

→ So far, all of our memory allocation is happening during compile time.

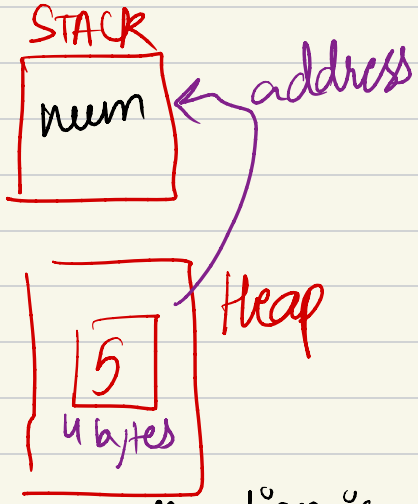
→ Compile Time memory allocation always happen in the STACK.

→ Runtime memory allocation happens in heap.

★ How to allocate memory while the runtime?

```
int main()
{
    int*num= new int(5);
    cout << * num ;
}
```

allocation of memory dynamically



once allocation is done in heap. The address is returned to the var 'num' in stack

When we define an array like int a[n], Here as soon as this is defined, memory is allocated in stack (fixed). But we only get the value of n during runtime resulting in less or more memory allocation during compilation time in the stack. This can lead to errors or wastage of memory.

We have to release the memory once we are done with execution so,

```
int main()
{
  int * num = new int(5);
   cout << * num;
   delete num;
}
```

← release of memory in heap