Pointer,Dynamic memory and Linked list

instructor

Dr Farzana Rahman

Pointer type, pointer arithmetic

int* -> int

char* -> char

Why strong types?

Why not some generic type?

Because-

Dereference

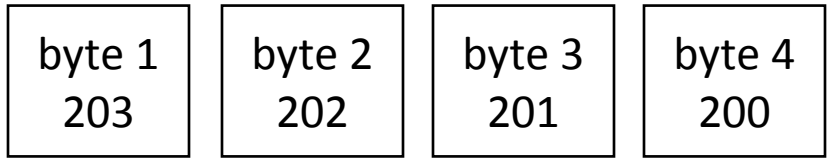
Access/modify value

int -> 4 bytes

char-> 1 byte

float-> 4 bytes

int a=1250



int a=1250

int* p

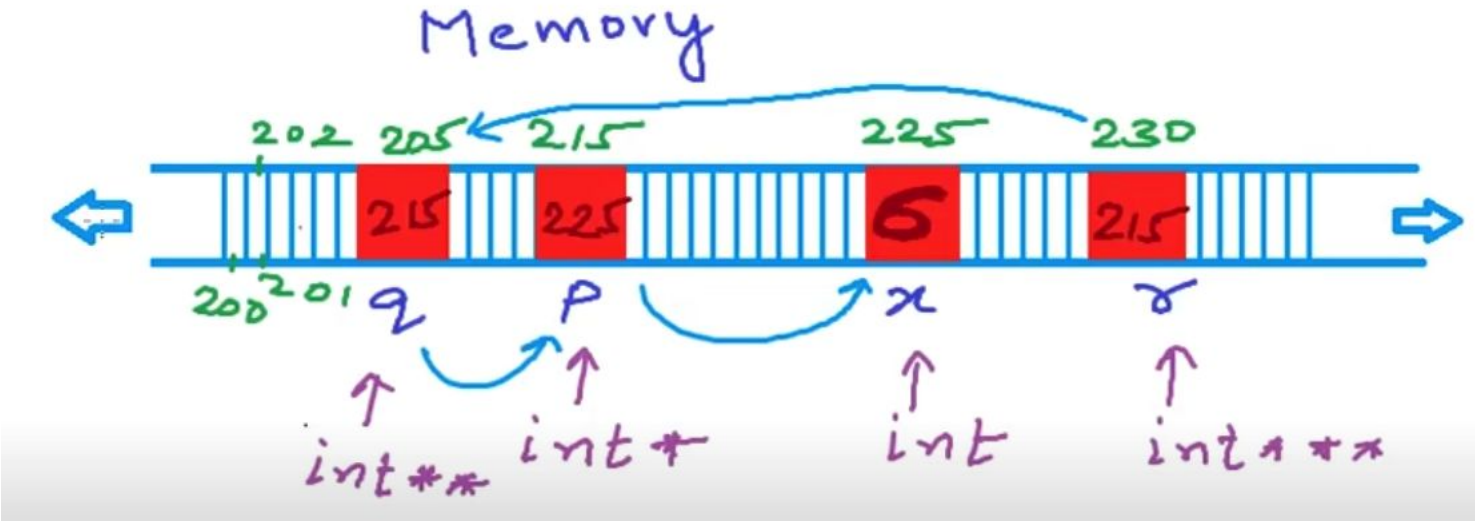
p=&a

print P // 200

print *p // look at 4 bytes starting at 200 to get the value 1250

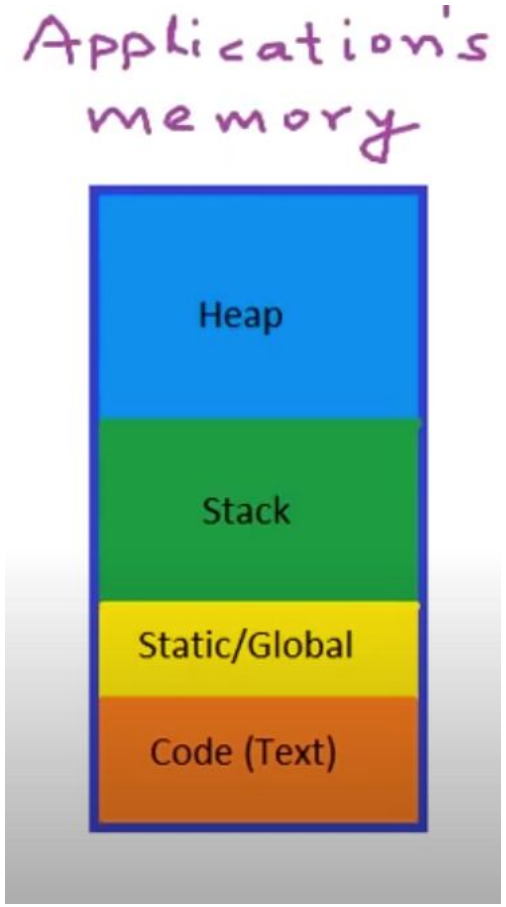
Pointer to Pointer

```
int x=6;
int* p;
p=&x;
int** q;
q=&p
int*** r;
r=&q

cout<<***r // 225
```



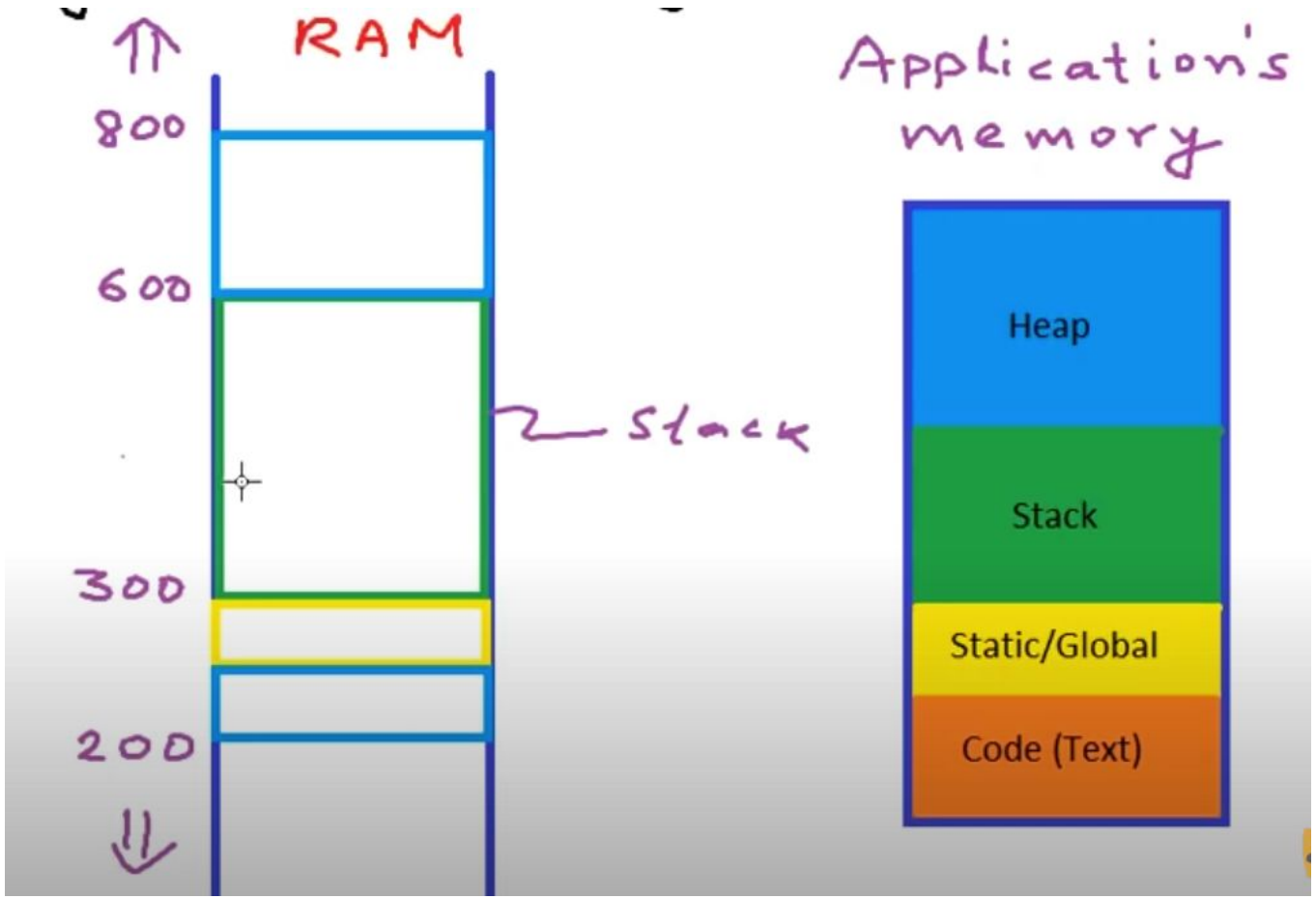
Pointers as function arguments/ Call by reference

```
#include<iostream>
using namespace std;
void increment(int a){
a=a+1;

}
int main()
{
int a;
a=10;
increment(a);
cout<<a;
return 0;}
}
```



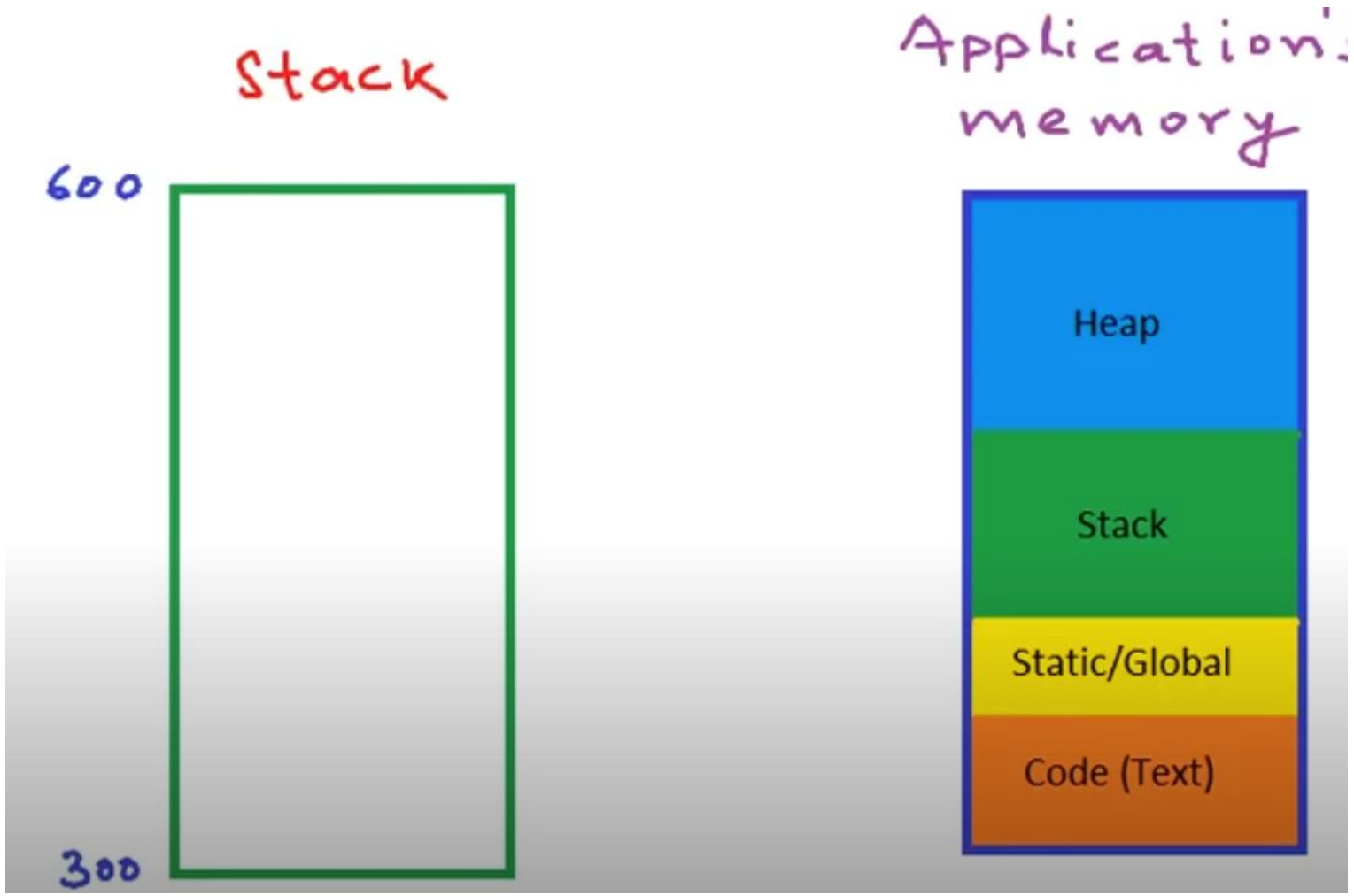
Pointers as function arguments/ Call by reference

```
#include<iostream>
using namespace std;
void increment(int a){
a=a+1;

}
int main()
{
int a;
a=10;
increment(a);
cout<<a;
return 0;}
}
```



Pointers as function arguments/ Call by reference

```
#include<iostream>
using namespace std;
void increment(int* p){
*p=(*p)+1;

}
int main()
{
int a;
a=10;
increment(&a);
cout<<a;
return 0;}
}
```



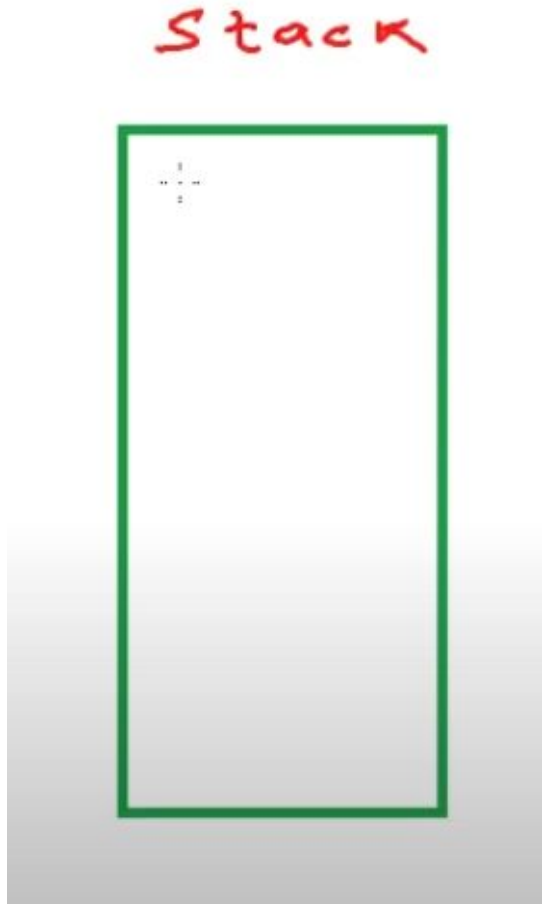
size of A=8
size of A[0]=4
sum of elements= 3
size of A=24
size of A[0]=4

Array as function argument

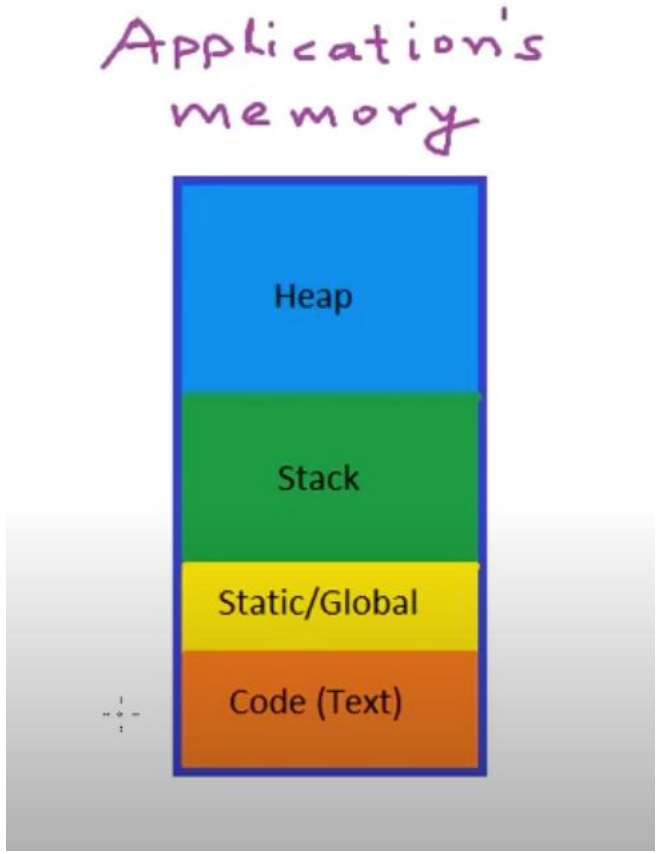
```
int SumOfElements(int A[])
{
int i, sum=0;
int size=sizeof(A)/sizeof(A[0]);
cout<<"size of A="<<sizeof(A)<<"\n";
cout<<"size of A[0]="<<sizeof(A[0])<<"\n";

for(i=0;i<size;i++)
{
sum+=A[i];
}
return sum;
```

```
int main()
{
int A[]={1,2,3,4,5,6};
int total=SumOfElements(A);
cout<<"sum of elements= "<<total;

cout<<"\n size of A="<<sizeof(A)<<"\n";
cout<<"\nsize of A[0]="<<sizeof(A[0])<<"\n";
return 0;
}
```

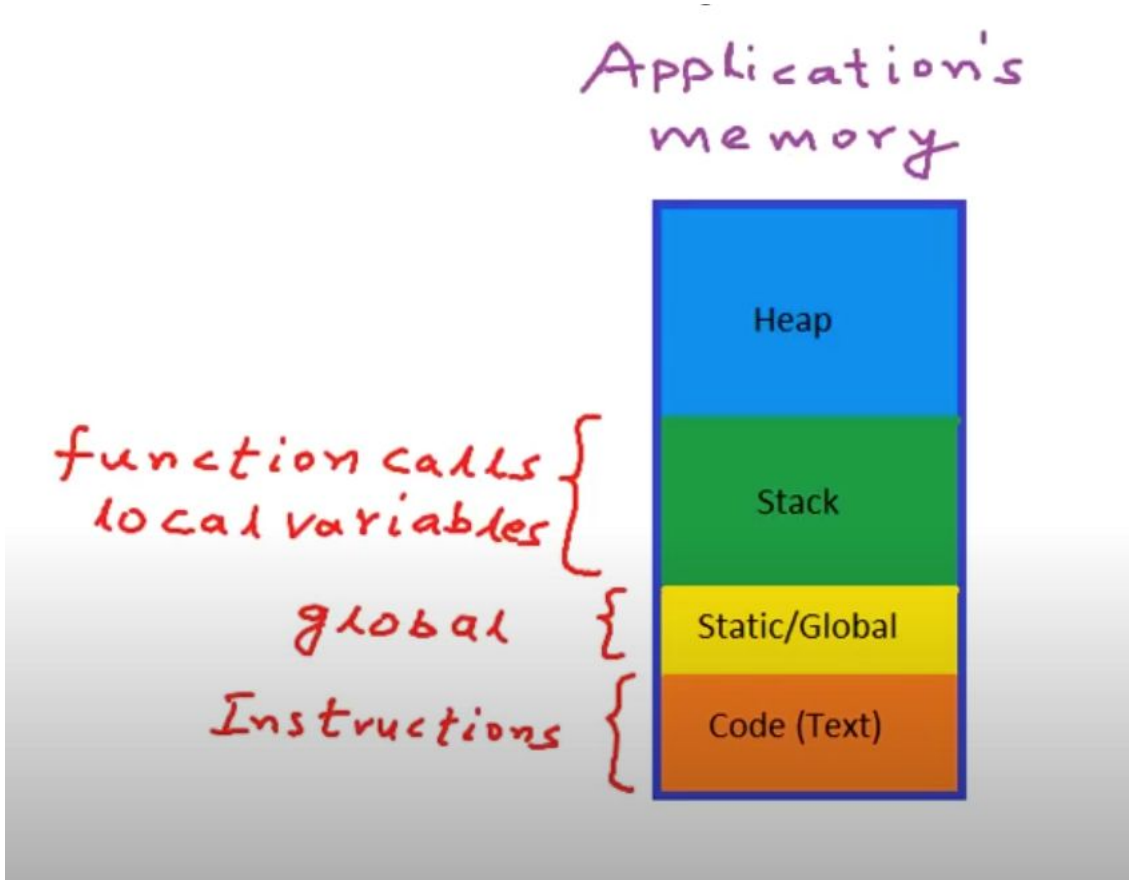
test_arrasfarg.cpp



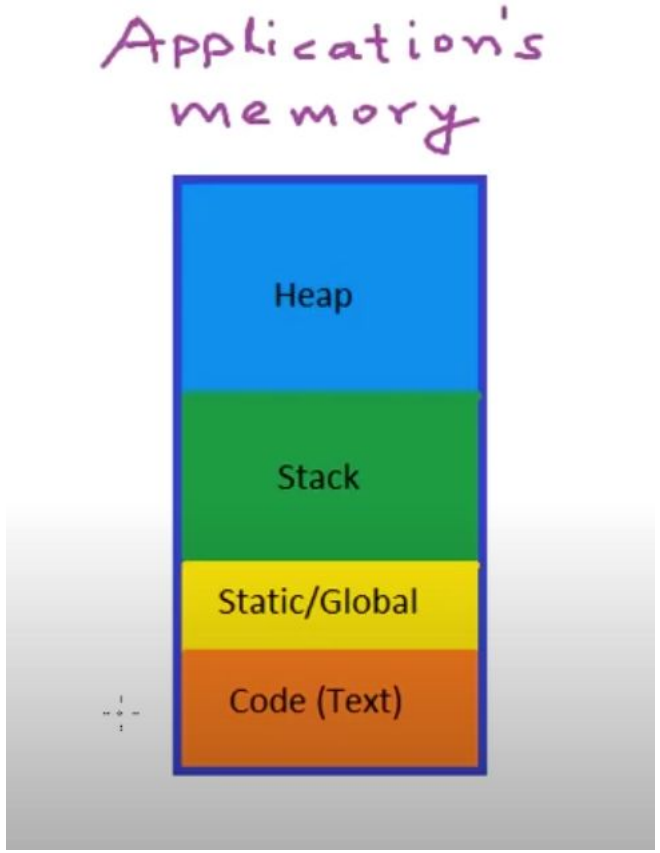
Pointers and dynamic memory



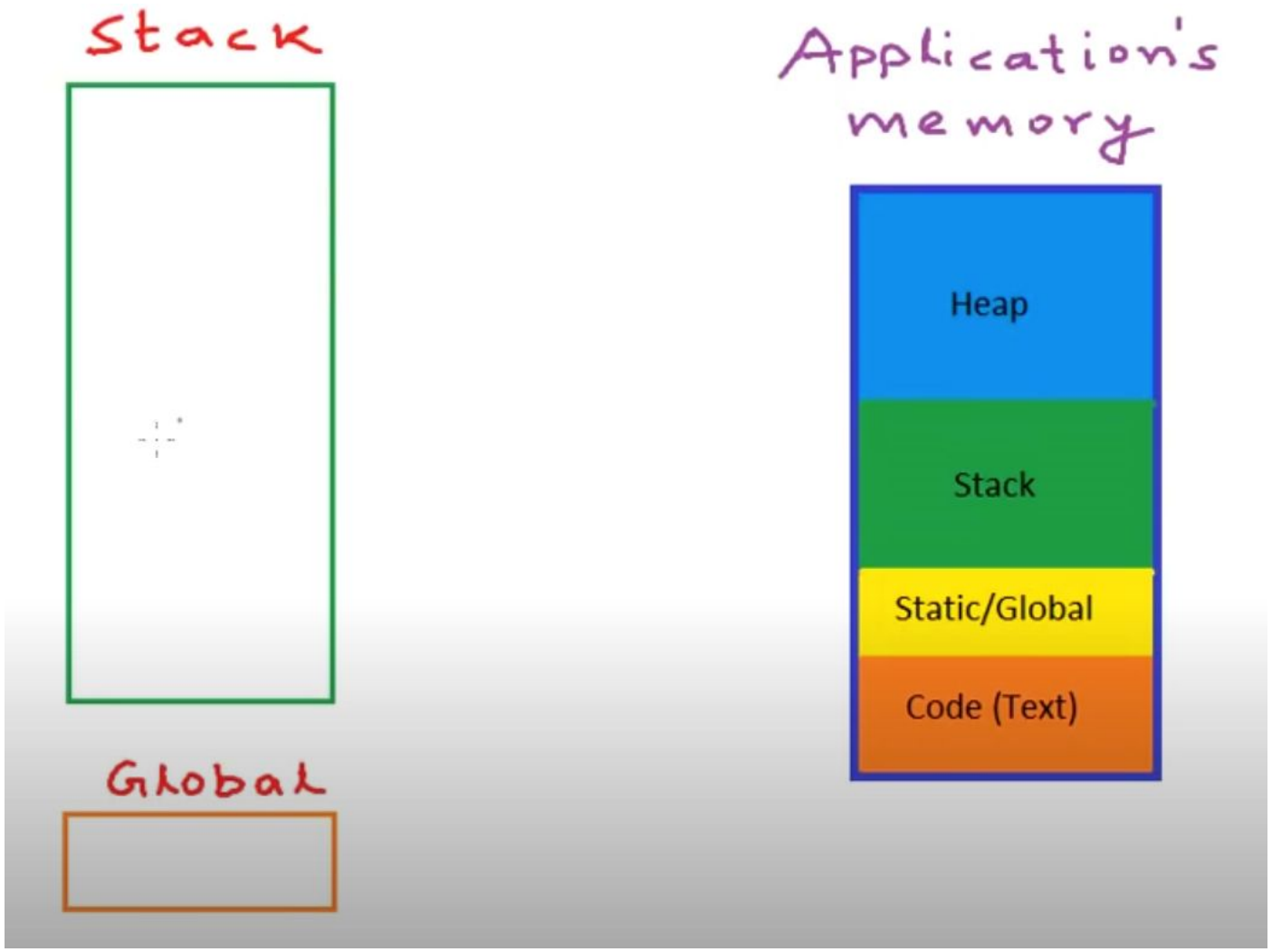
Pointers and dynamic memory



```
#include<iostream>
using namespace std;
int total;
int Square (int x)
 {
	return x*x;
}

int SquareOfSum(int x, int y)
 {
	int z=Square(x+y);
	return z;
}

int main()
{	int a=4, b=8;
	total=SquareOfSum(a,b);
	cout<<total;
}
```

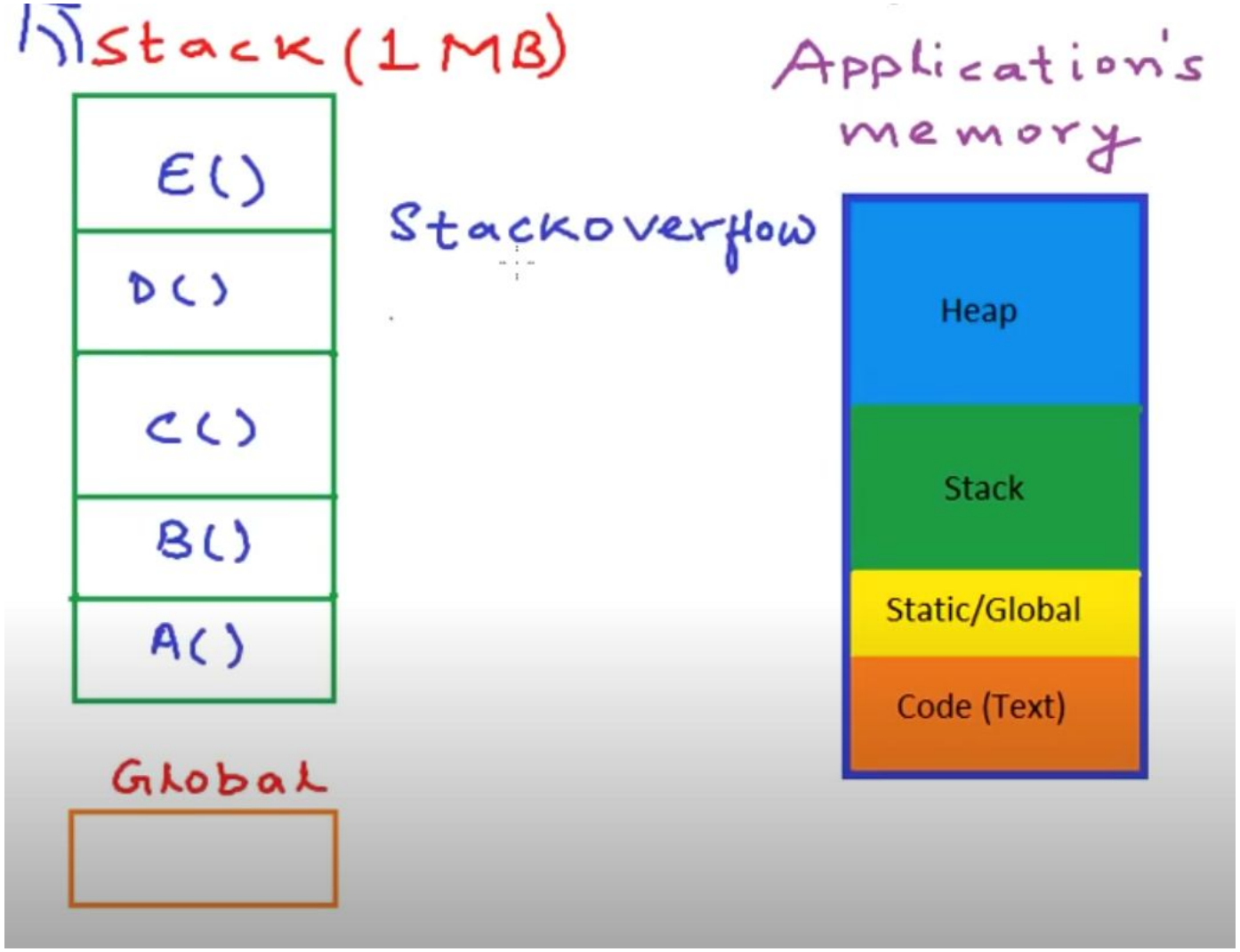


```
#include<iostream>
using namespace std;
int total;
int Square (int x)
 {
       return x*x;
}

int SquareOfSum(int x, int y)
 {
       int z=Square(x+y);
       return z;
}

int main()
{      int a=4, b=8;
       total=SquareOfSum(a,b);
       cout<<total;
}
```



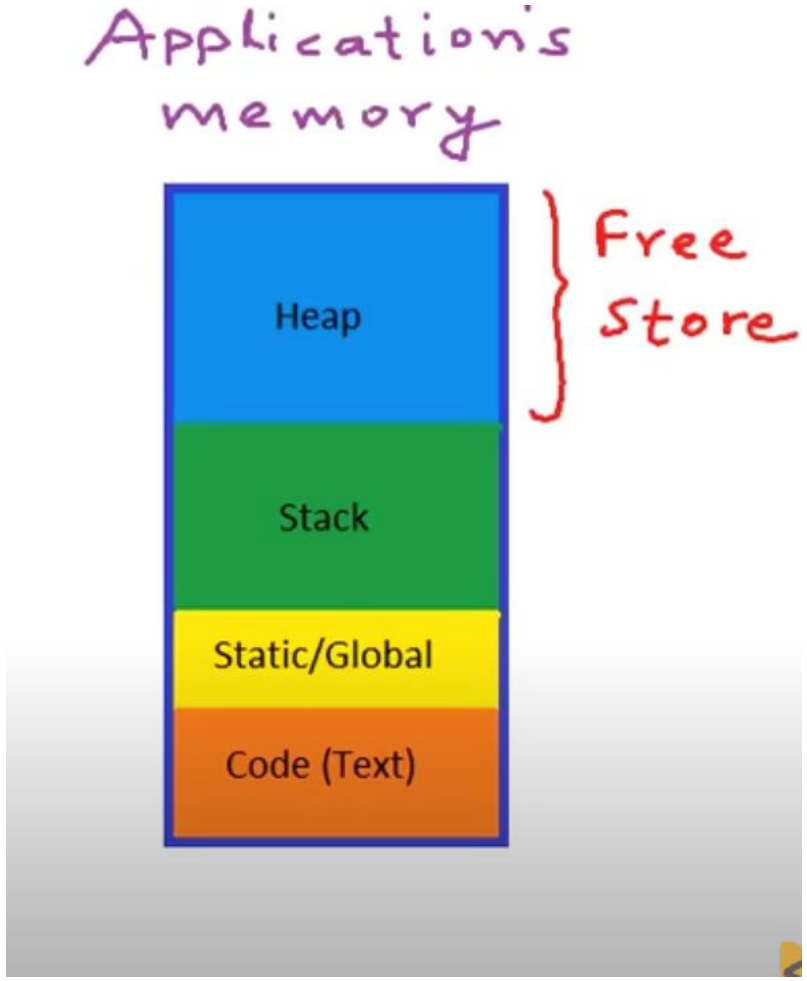
stack overflow

```
#include<iostream>
using namespace std;
int total;
int Square (int x)
 {
       return x*x;
}

int SquareOfSum(int x, int y)
 {
       int z=Square(x+y);
       return z;
}

int main()
{      int a=4, b=8;
       total=SquareOfSum(a,b);
       cout<<total;
}
```

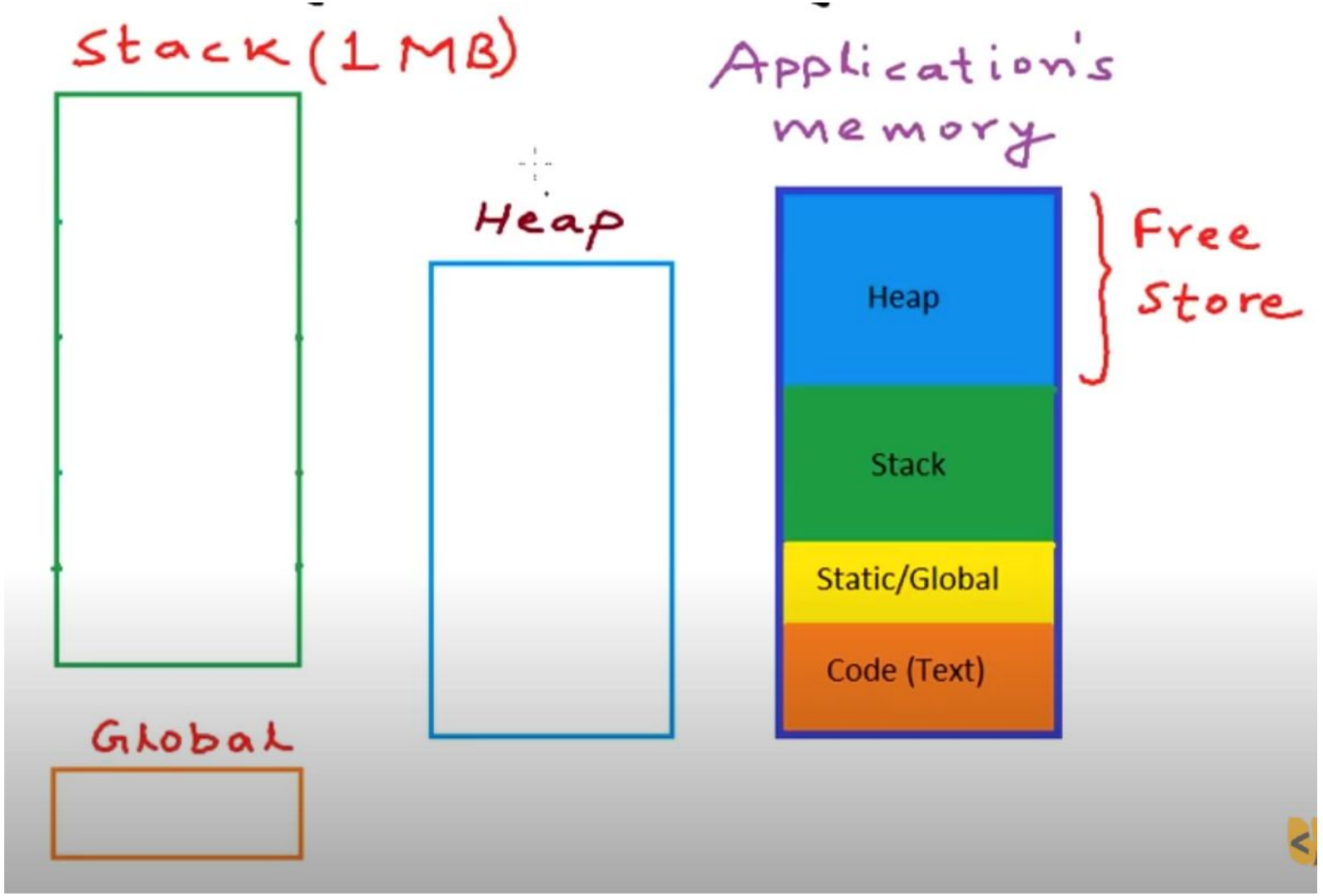


Dynamic memory

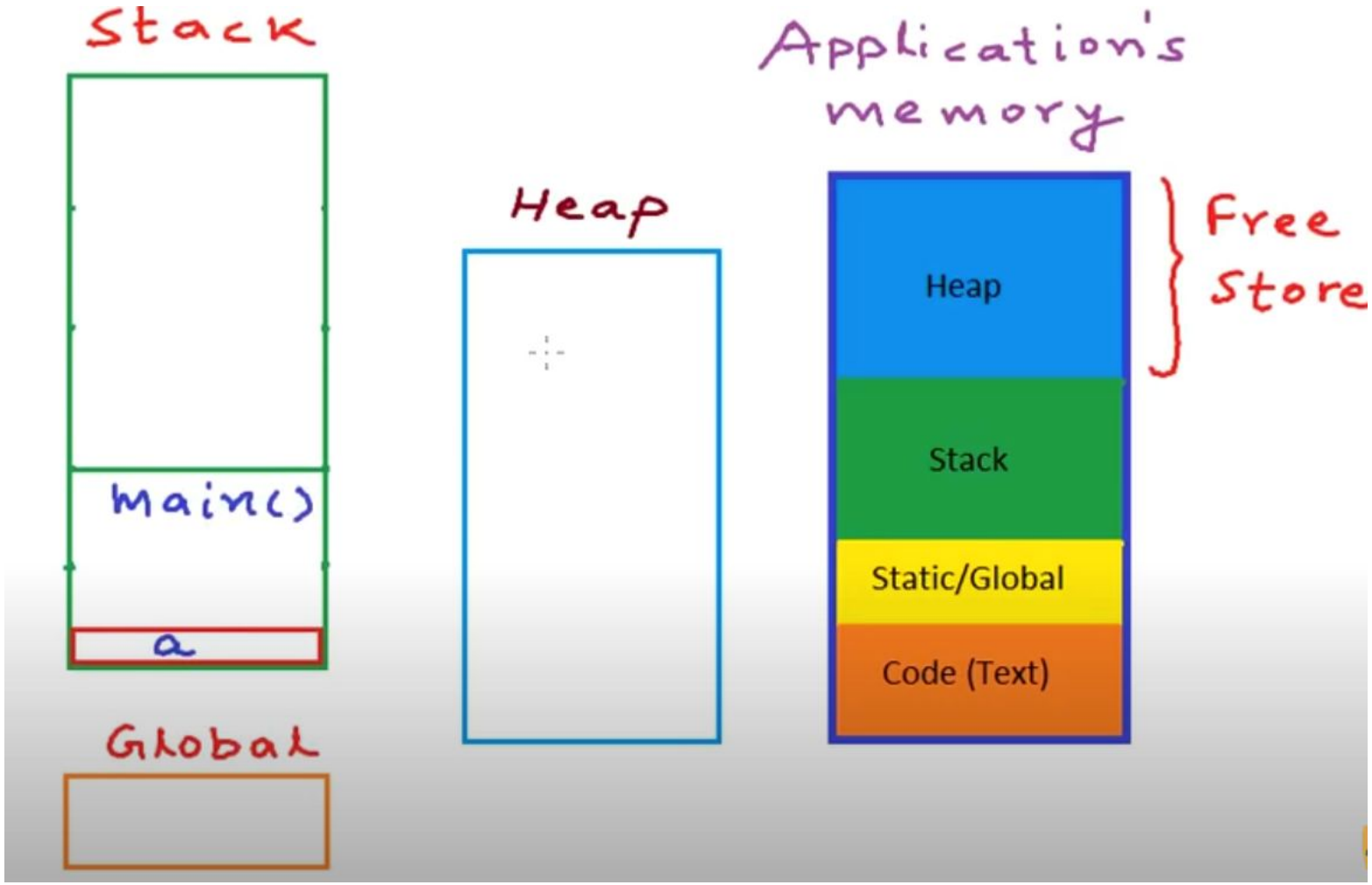
- Heap is called dynamic memory.
- using the heap refers to as dynamic memory allocation.
- free pool of memory.
- we can allocate memory chunck flexibly during program runtime.
- we need to deallocate memory to save us from memory likage.



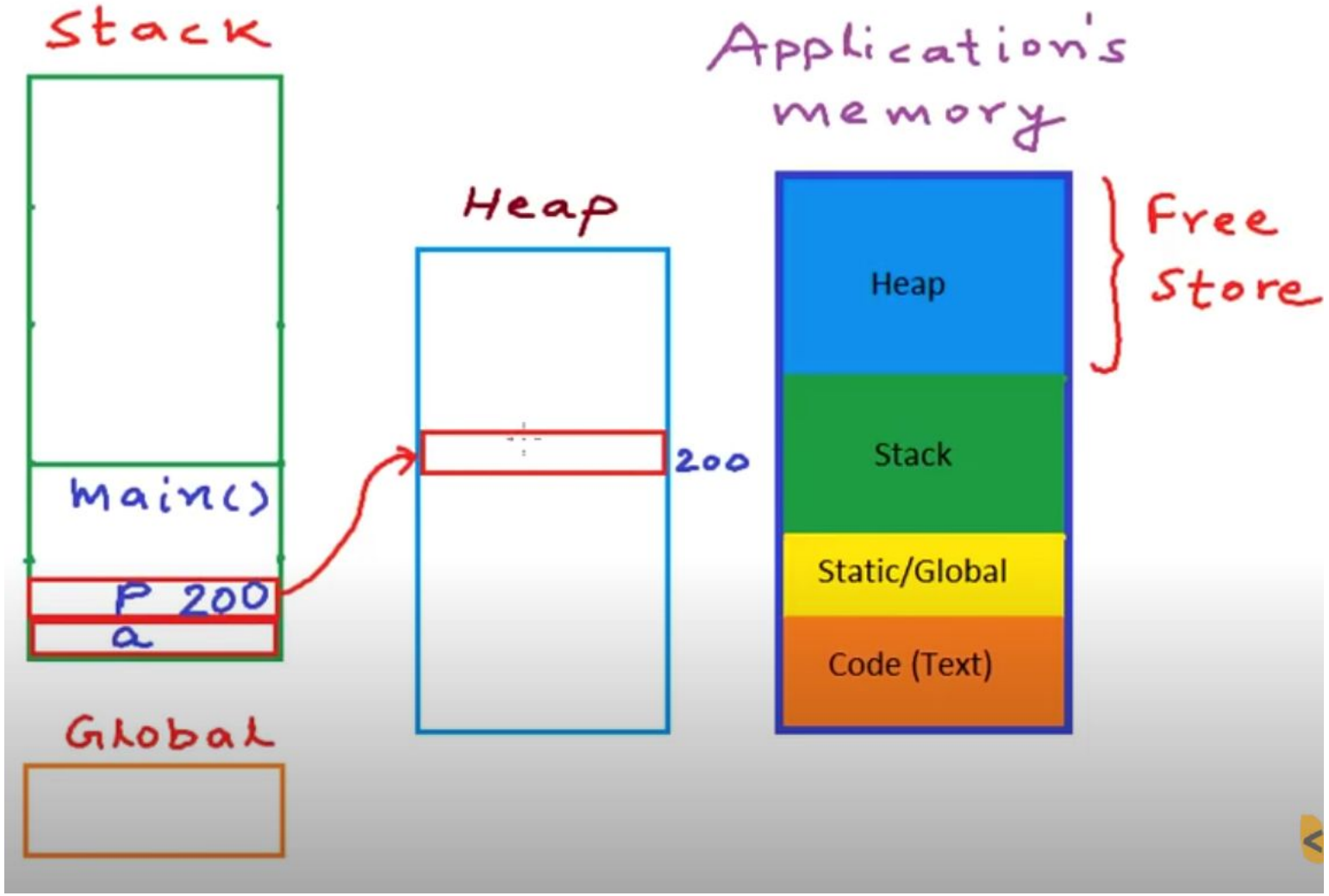
C++
new
delete


Stack (1 MB)
Heap
Global
Application's memory
Heap
Stack
Static/Global
Code (Text)
Free Store

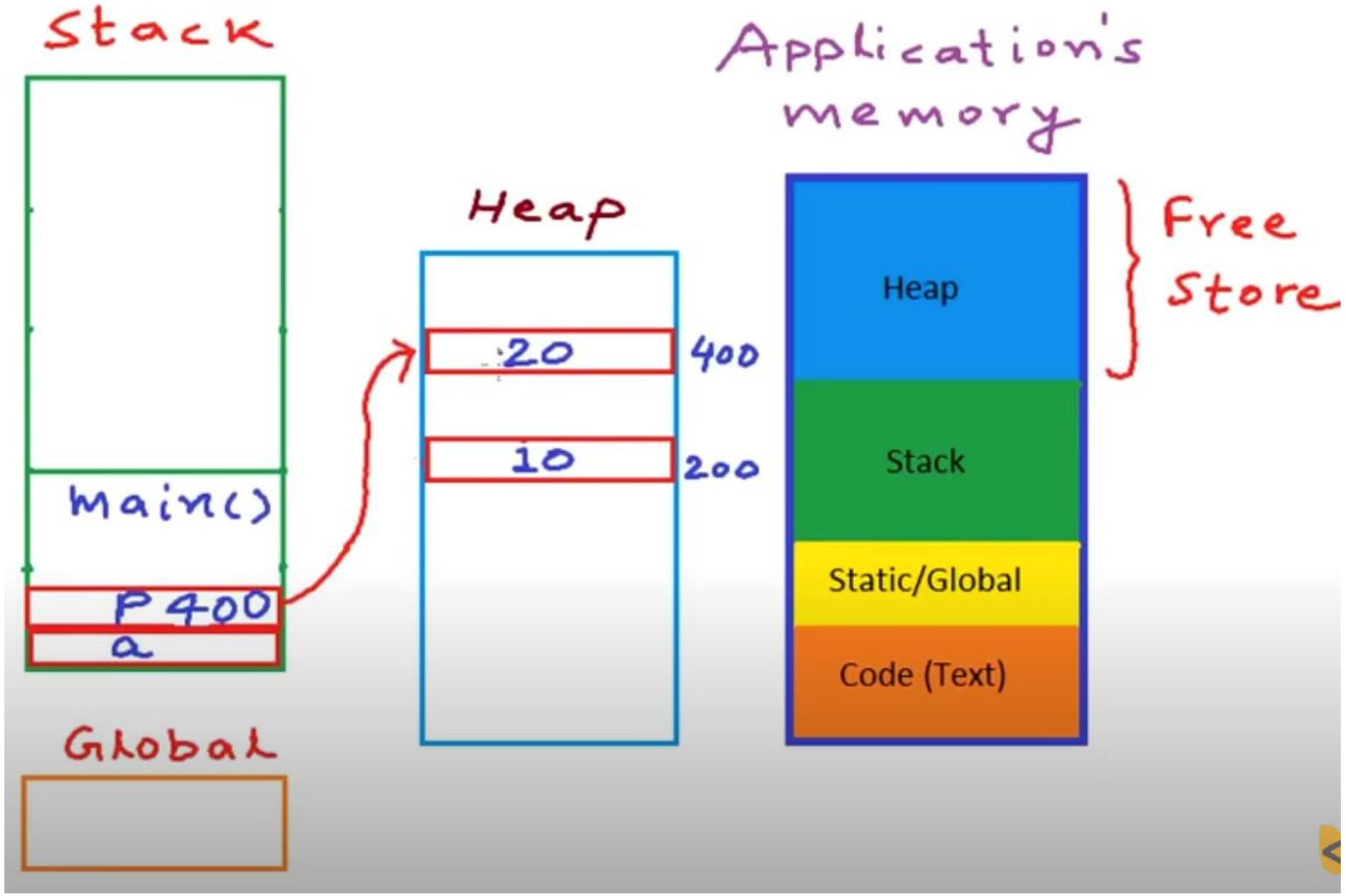
```
#include<iostream>
using namespace std;

int main()
{
        int a; //goes on stack



}
```



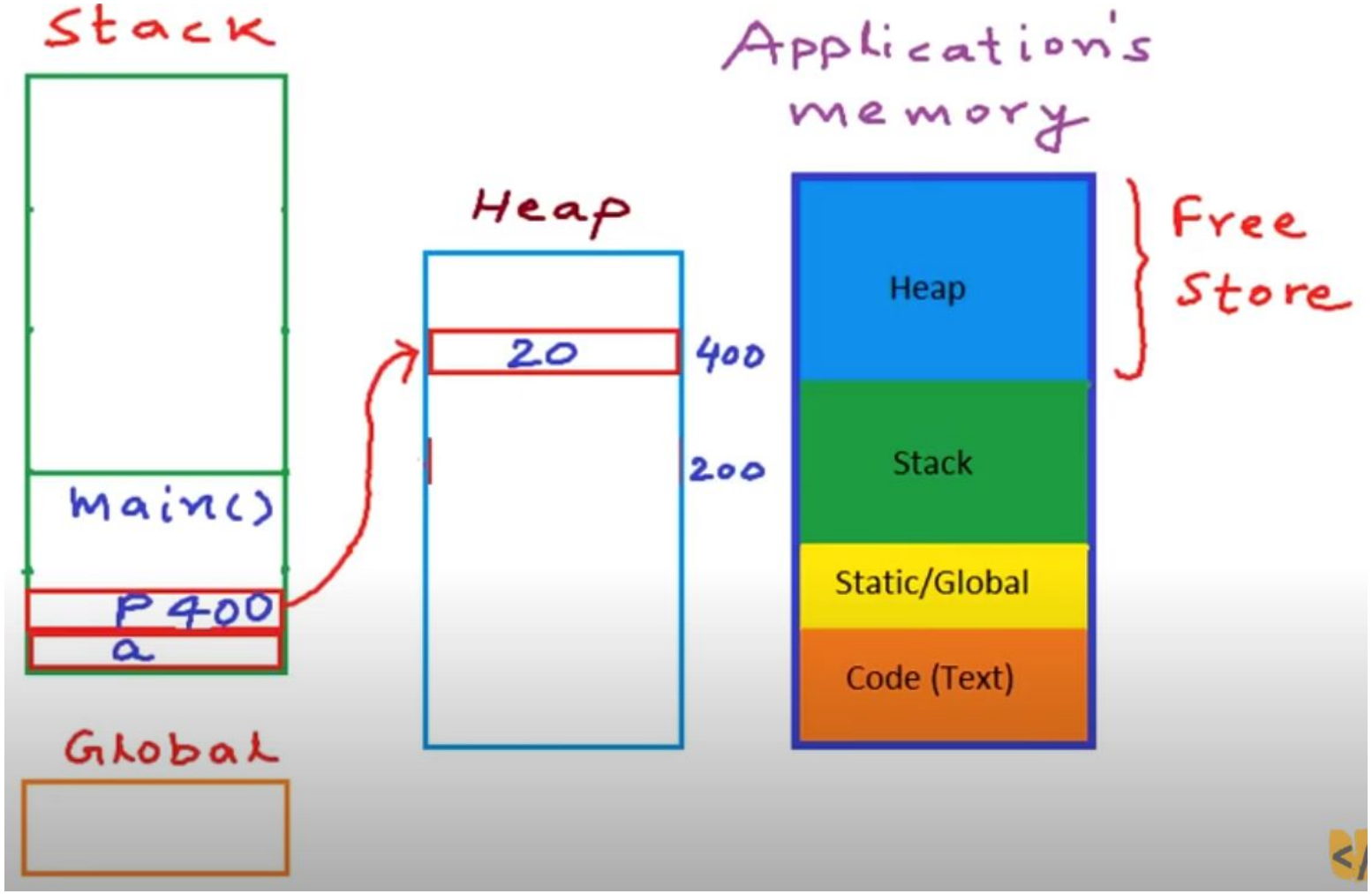
```
#include<iostream>
using namespace std;

int main()
{
	int a; //goes on stack
	int* p;
	p=new int; // allocate
from heap
	*p=10;

}
```



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

int main()
{
        int a; //goes on stack
        int* p;
        p=new int; // allocate
from heap
         *p=10;
        p=new int;
        *p=20;

}
```

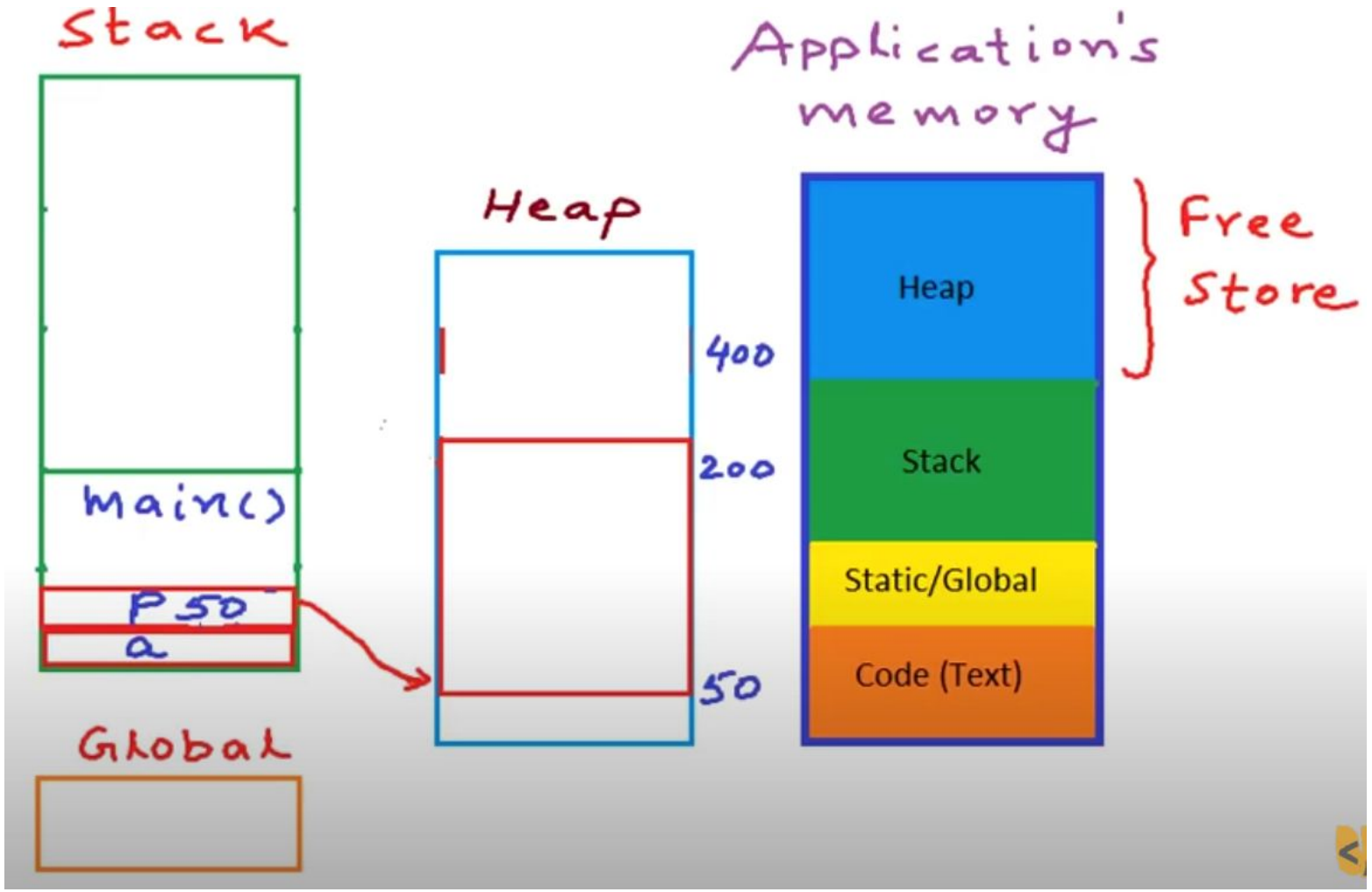


```
#include<iostream>
using namespace std;

int main()
{
        int a; //goes on stack
        int* p;
        p=new int; // allocate
from heap
         *p=10;
        delete p;
        p=new int;
        *p=20;


}
```



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

int main()
{
	int a; //goes on stack
	int* p;
	int* a=new int; // allocate
	from heap
	delete p;
	int* a=new int[10];

	delete[] a;
}
```

p[0]= *p
p[1]=*(p+1)



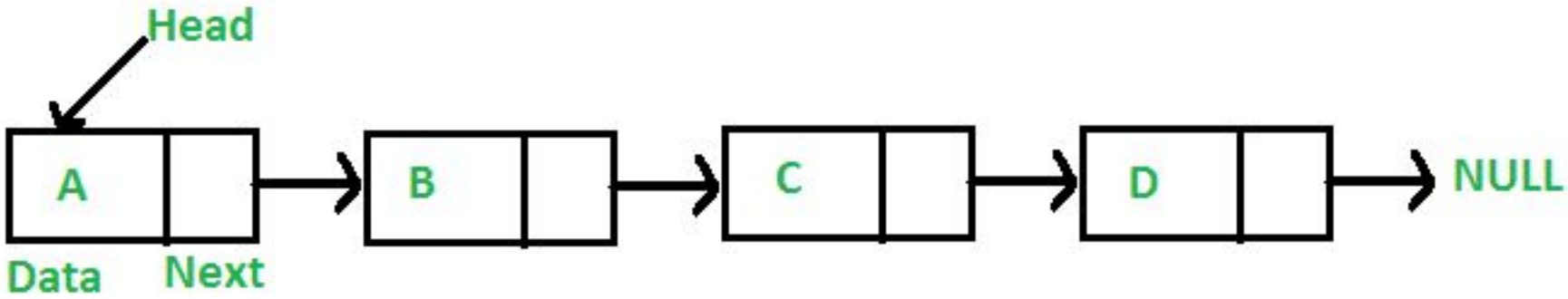
- Lets practice an dynamic_array

  dyn_array.cpp

Linked List

- Singly linked list
- Circular linked list
- Doubly linked list

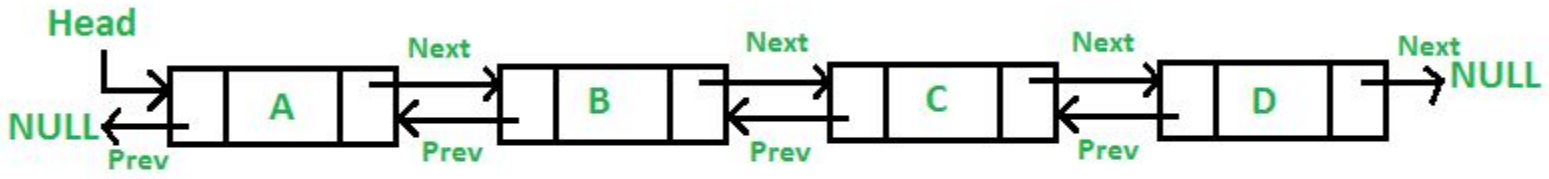
Singly Linked List



•A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

•In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Doubly linked List



A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.

Circular linked list

Circular linked list *is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*

Link list implementation

- C++

```
class Node {
public:
    int data;
    Node* next;
};
```

- C &C++

```
struct Node {
    int data;
    struct Node* next;
};
```

Recursion

- ***approach(1) – Simply adding one by one***

f(n) = 1 + 2 + 3 +……..+ n

- ***approach(2) – Recursive adding***

f(n) = 1 n=1

f(n) = n + f(n-1) n>1

Factorial (Recursive function)

```
int fact(int n)
{
if (n < = 1)       // base case
return 1;
else
return n*fact(n-1);
}
```

Stack Overflow

```
int fact(int n) // wrong base case (it may cause // stack overflow).
{

if (n == 100)
return 1;
else
return n*fact(n-1);

 }
```

```
void recurse()
{ ... .. ... recurse();
 ... .. ... }

int main()
{ ... .. ... recurse();
 ... .. ... }
```