

Object Oriented Programming with C++

Introduction and Course details

Lecturer : Astha Sharma

CHAPTER 3

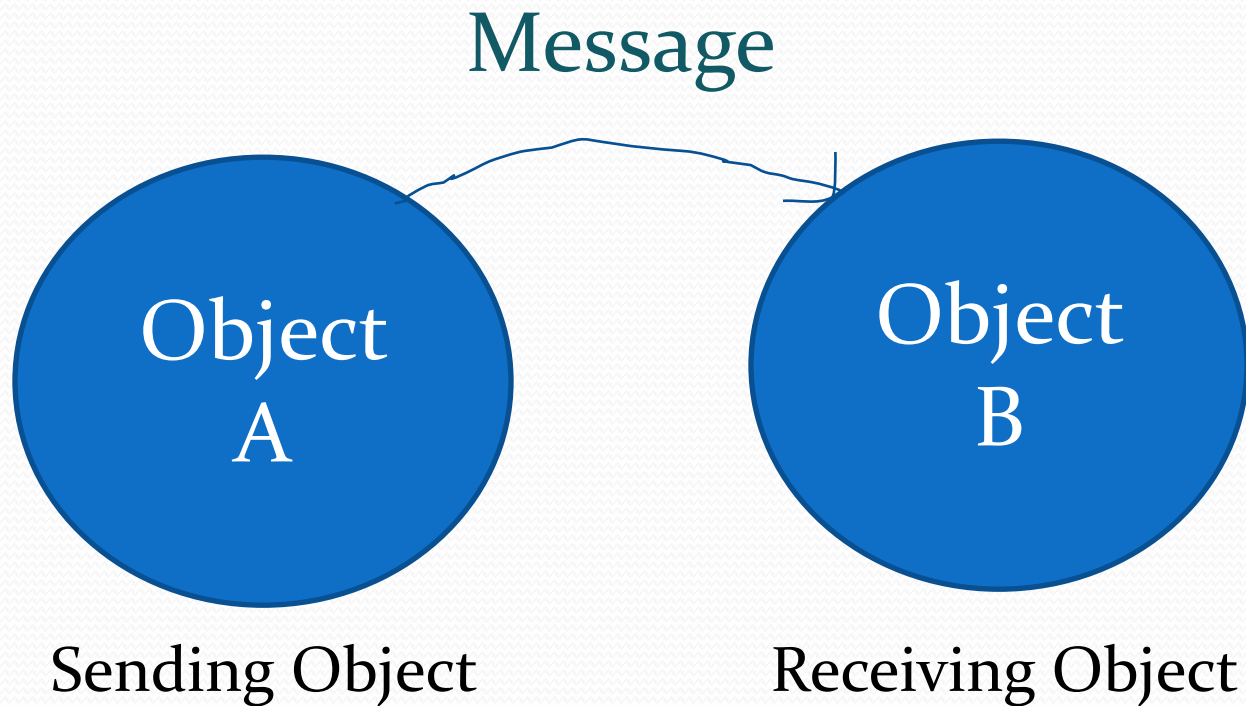
Message, Instance and Initialization
(6 hrs)

Message Passing

Message

- A request for an object to perform one of its operations is called message.
- Operation means functions or methods.
- All communication between objects is done via message called as message passing.
- Objects and Classes interact using messages. Message passing refers to sending and receiving of messages by the objects of class just like we people exchange information in real life.

What is Message Passing?



Message Passing

Message passing syntax in C++

Format of Message Passing

Message passing requires :

- **Recipient object** : Object that undergoes change
 - **Dot operator** : Enables message passing features
 - **Method** : Contains the message to be passed
 - **Parameters** : Values needed for the method
 - **Sender object** : Object that contains the code of method invocation done in the recipient object.
-
- Message passing format:
 recipient_object.method(parameters);

Example of Message

- Shape rectangle;
- rectangle.findarea(width..90, height..80);
- Rectangle is the object of class called “Shape”.
- findarea() is a message or function of class Shape.
- width and height are information passed in message, function

Message Passing





Ex:

```
Student1 . getdata("ram" , 10);
```

```
Student2 . getdata("suju", 20);
```

```
Circle.calc_area(12);
```

```
Rectangle. Calc_area(12,50);
```

Constructor in C++

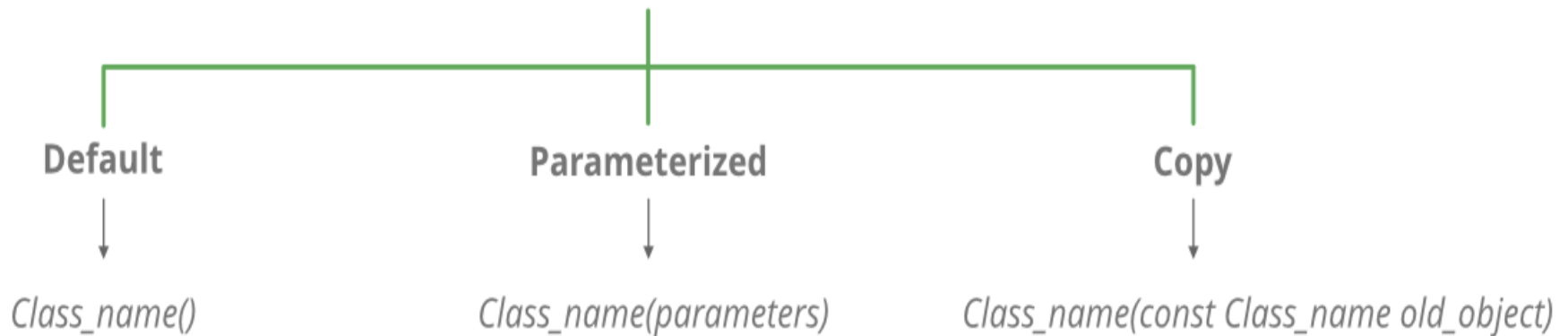
What is a constructor?

- A constructor is a member function of a class which initializes values of objects of a class.
- In C++, constructor is automatically called when object(instance of class) is created.
- It is special member function of the class.

Things to remember while creating a constructor:

- Constructor has the same name as the class itself
- It is called automatically when object of that class is invoked.
- It is always declared in public part of class
- It does not return any type of value(not even void type)
- It may/may not have arguments(participate in overloading)
- It does not participate in inheritance
- It is executed only once when objects are created and further calling is not possible
- It is never invoked with object name.

Constructor in C++



Types of Constructors



Default Constructors:

- Default constructor is the constructor which doesn't take any argument.
- It has no parameters.
- In C++, the standard describes the default constructor for a class as a constructor that can be called with no arguments (this includes a constructor whose parameters all have default arguments).

```
#include <iostream>
#include <conio.h>

using namespace std;

class student
{
public:
    student()
    {
        cout<<"default constructor called";
    }
};

int main()
{
    student s1;

    return 0;
}
```

Output:

default constructor called


```
#include <iostream>
#include <conio.h>

using namespace std;

class counter
{
private :
    int count;
public:
    counter()
    {
        count = 0;
        cout << "count : " << count;
    }
};

int main()
{
    counter c1;
    return 0;
}
```

Output:

count : 0



Remember:

- Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

Parameterized Constructors:

- It is possible to pass arguments to constructors.
- Typically, these arguments help to initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- When you define the constructor's body, use the parameters to initialize the object.

```
#include <iostream>
#include <conio.h>

using namespace std;

class counter
{
private :
    int count;
public:
    counter(int num)
    {
        count = num;
        cout << "count : " << count;
    }
};


int main()
{
    counter c1(90);
    return 0;
}
```


Copy Constructor:

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

The copy constructor is used to –

1. Initialize one object from another of the same type.
2. Copy an object to pass it as an argument to a function.
3. Copy an object to return it from a function.

- 
- There are two types of copy constructors:
 1. **default copy constructor**
 2. **user_defined copy constructor**
 - The compiler generated copy constructor is called default copy constructor.
 - The copy constructor which is copied from the parameterized constructor previously defined is called user_defined copy constructor.

- 
- There are two types of copy constructors:
 1. **default copy constructor**
 2. **user_defined copy constructor**
 - The compiler generated copy constructor is called default copy constructor.
 - The copy constructor which is copied from the parameterized constructor previously defined is called user_defined copy constructor.

```
#include <iostream>
#include <conio.h>
```

```
using namespace std;
```

```
class counter
{
```

```
private :
    int counts;
```

```
public:
```

```
    counter()
    {
        counts = 0;
    }
```

```
    counter(int num)
    {
        counts = num;
    }
```

```
    counter (counter &x)
    {
        counts = x.counts;
    }
```



```
int display_counts()
{
    return counts;
}
```


```
};
```


```
int main()
{
    counter c1; //c1 obj + default constructor called
    counter c2(3); // c2 obj + parameterized constructor called
    counter c3(c2); // c3 obj + copy constructor called
    counter c4=c1; // c4 obj + copy constructor called

    cout << "the counts in object c1 is :"<< c1.display_counts()<<endl;
    cout << "the counts in object c2 is :"<< c2.display_counts()<<endl;
    cout << "the counts in object c3 is :"<< c3.display_counts()<<endl;
    cout << "the counts in object c4 is :"<< c4.display_counts()<<endl;

    return 0;
}
```

Memory map in C++

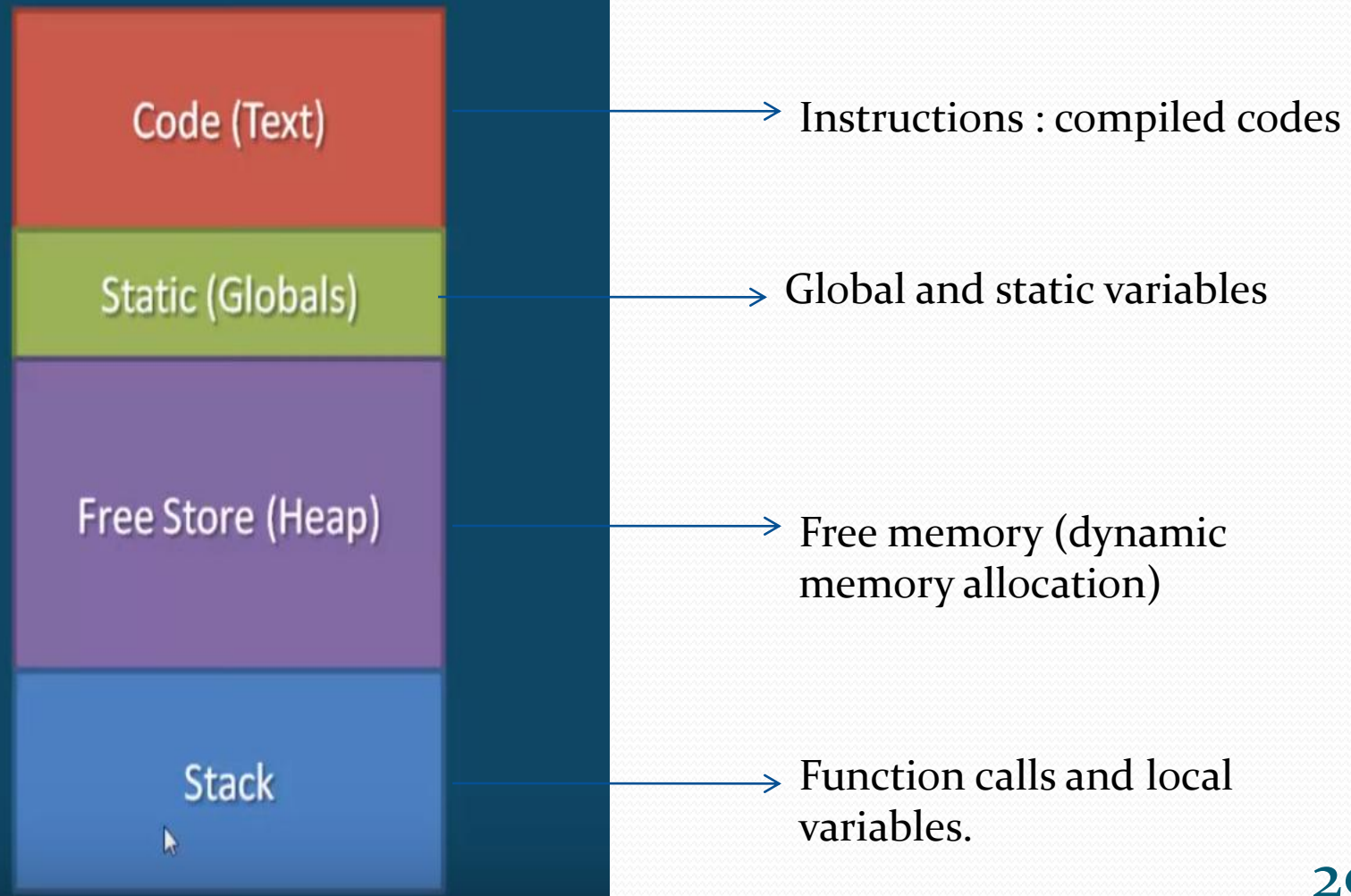
- 
- Memory is one important and crucial resource on our machine.
 - It is always good to know the architecture of memory, the way operating system manages memory and the way memory is accessible to us , the programmers.
 - So, let's start with memory mapping in C++.




•In C++, although the actual memory management is quite complex, yet memory can be divided into 4 sub-parts :

1. Code segment
2. Global/static segment
3. Free store segment
4. Stack segment

Memory Layout



- 
1. **Program/Code** : This region holds the compiled code of the program. Here, all the functions of the program start at a specific address.
 2. **Global variables**: In this region, global variables are stored during the entire lifetime of the program.

3. Stack:

- This region is responsible for storing the return addresses at function calls, arguments passed to a function as well as local variables.
- It is important to note that local variables remain in the computer memory as long as the function continues.
- Soon as the function ceases, memory is erased from the stack.
- In addition to all this, the stack is also responsible for storing the current state of the CPU.

4. Heap:

- In this region, there is an abundance of free memory chunks that are allocated with the help of DMA.

```

#include<stdio.h>
int total;
int Square(int x)
{
    return x*x; //  $x^2$ 
}
int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return z; //  $(x+y)^2$ 
}
int main()
{
    int a = 4, b = 8;
    total = SquareOfSum(a,b);
    printf("output = %d",total);
}

```


Stack



Global

total

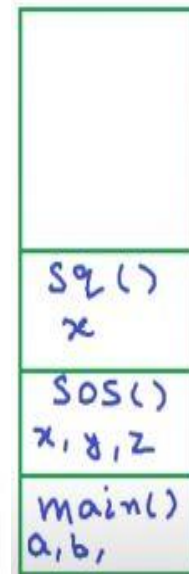
Stack



Global

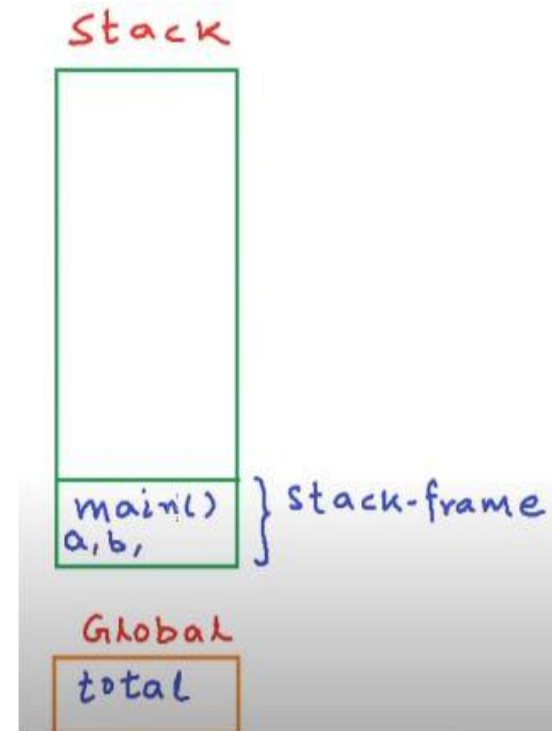
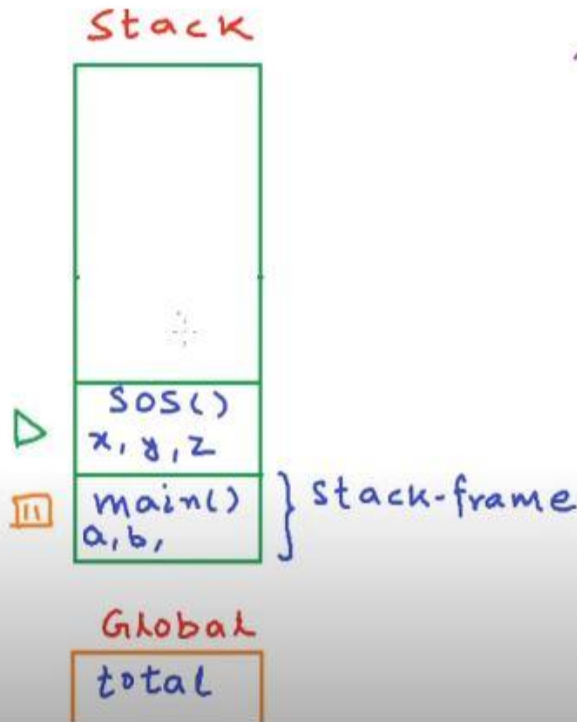
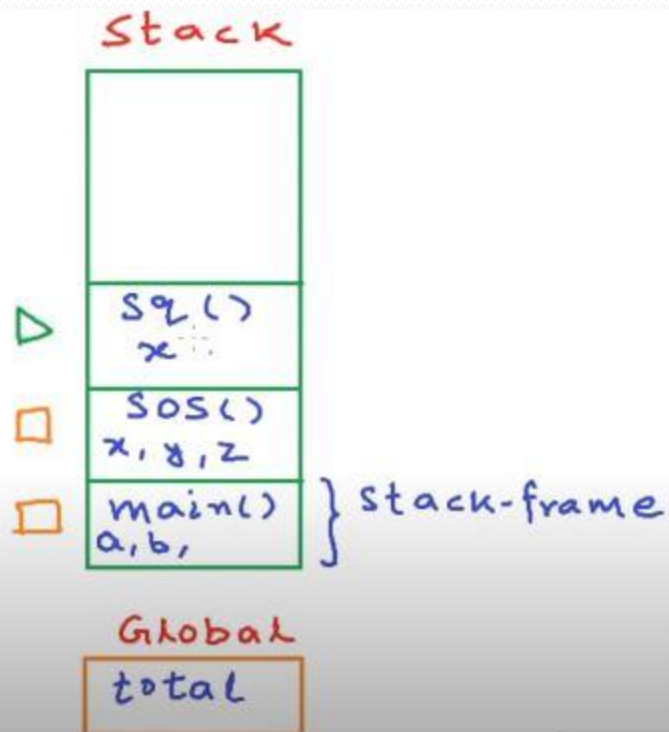
total

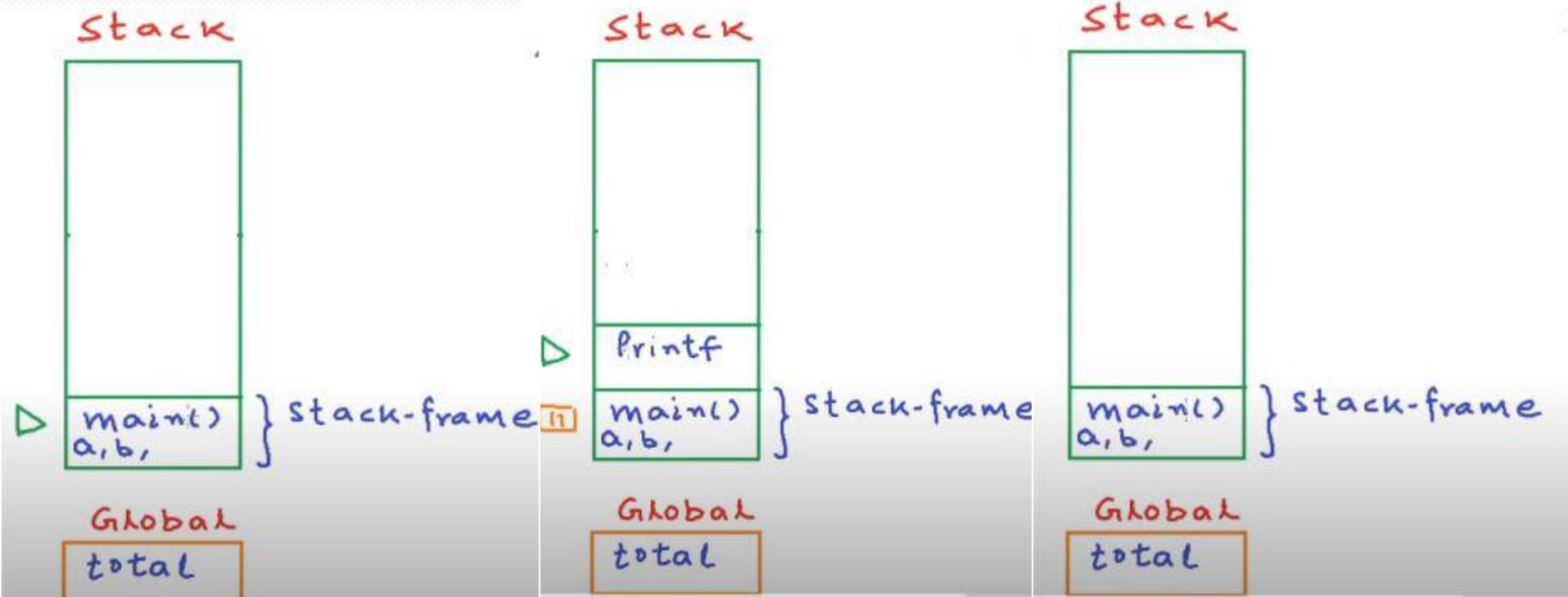
Stack



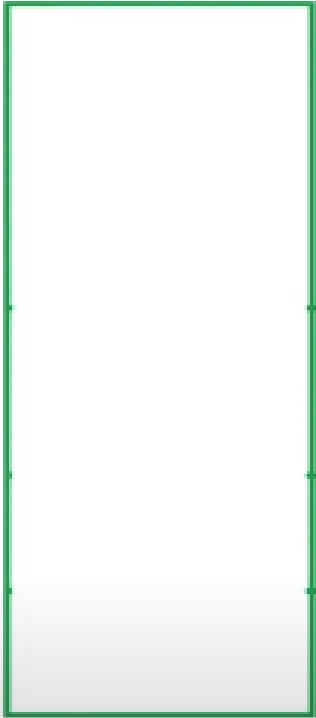
Global

total





Stack

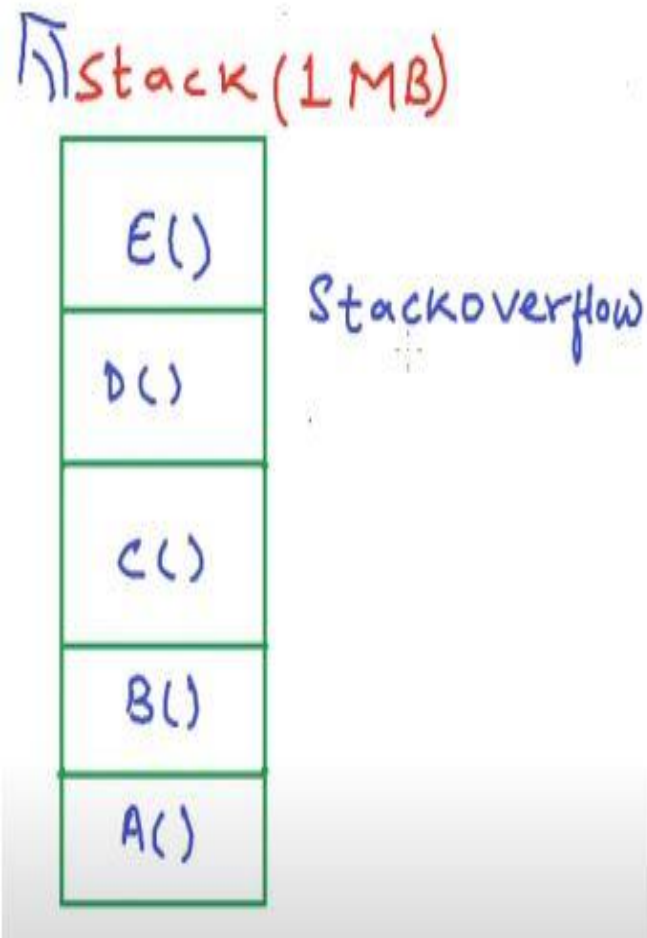


Global

At last,

- Stack and global memory segments also get emptied as all the instructions or compiled codes get executed.
- In this way, memory is managed for such programs which do not contain the DMA.

Drawback of *stack*/ Need of *heap*



- When program starts O.S. reserves some reserved memory for stack (say 1 MB)
- But actual allocation of space of stack (local variables and function calls) happens during runtime.
- And if our actual stack goes beyond the reserved memory of stack (more than 1 MB) then this is called stack overflow and our program stops.
- Example of stack overflow, is bad recursion.
- That's why heap is needed.

```
#include <stdio.h>
#include <conio.h>
```

```
int main()
{
```

```
int a; // goes on stack
```

```
int *p;
```

```
p= new int;
```

```
*p = 10;
```

```
free(p);
```

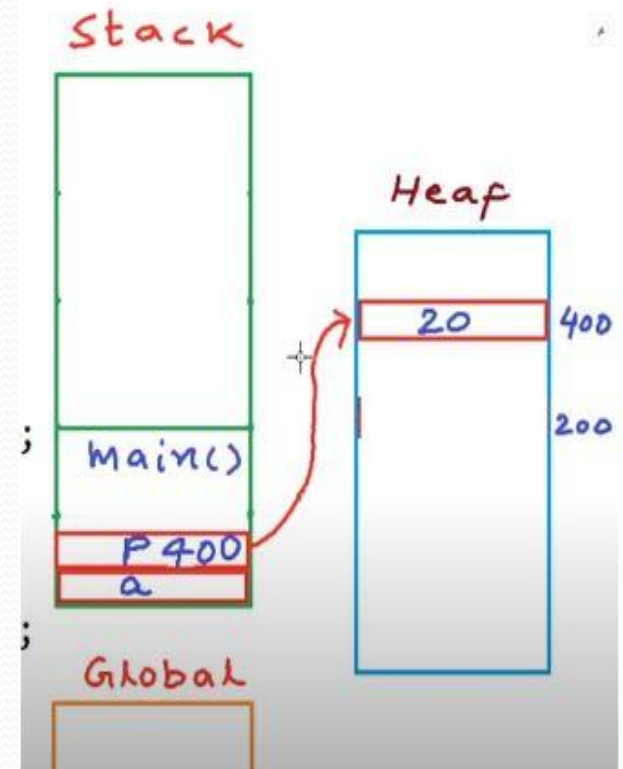
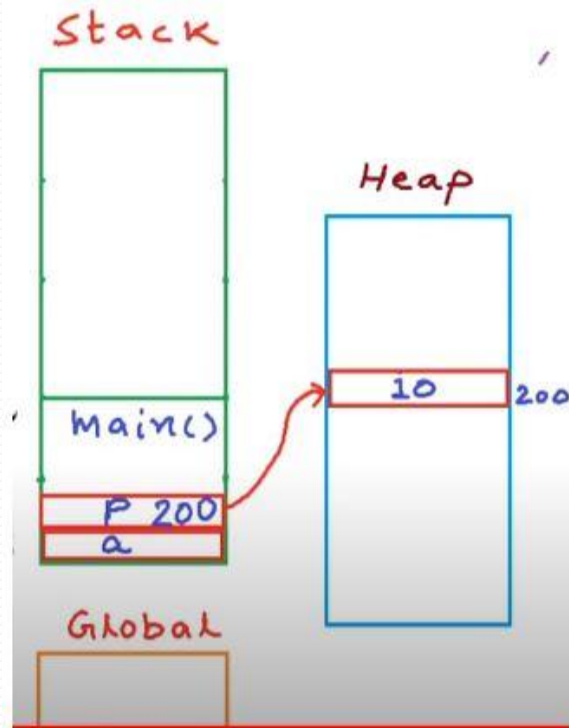
```
p=new int;
```

```
*p = 20;
```

```
free(p);
```

```
return 0;
```

```
}
```



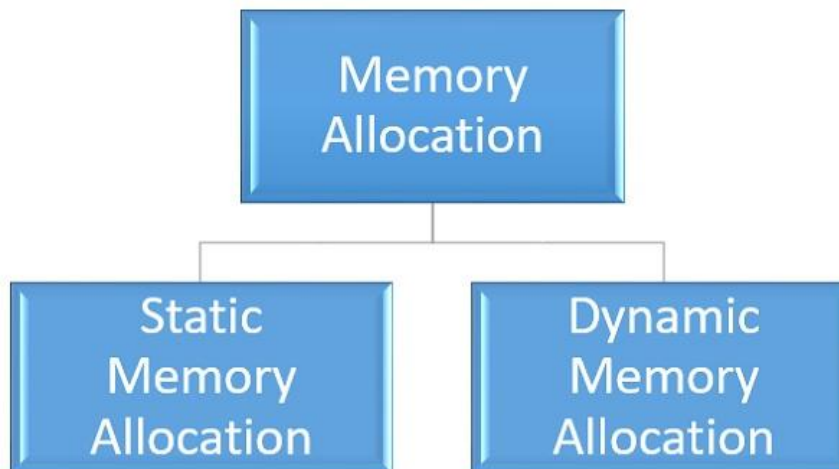
Benefit of heap memory

- Heap is a free space(pool) of memory.
- It is not reserved at compile time.
- During dynamic allocation of memory, the space of heap is needed.
- There is no question of heap overflow as no space is reserved prior to DMA.
- Also, heap memory is not deallocated itself after the program ends, but have to free the space by the programmer manually
- Otherwise, the reserved memory of heap cannot be able to use for other purpose.

Memory allocation in C++

Dynamic and Static Allocation of Memory in C++:

- The golden rule of computers states that anything and everything (data or instruction) that needs to be processed must be loaded into internal memory before its processing takes place.
- Therefore, every data and instruction that is being executed must be allocated some area in the main(internal) memory.
- The main(internal) memory is allocated in these two ways :





1. Static Memory Allocation:

- Static Memory is allocated for declared variables by the compiler.
- The address can be found using the address of operator and can be assigned to a pointer.
- The memory is allocated during compile time.

2. Dynamic Memory Allocation:

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer.
- The memory is allocated during run time.
- New and Delete operator help in DMA.

Static Memory Allocation in C++

- When the amount of memory to be allocated is known beforehand and the memory is allocated during compilation itself, it is referred to as static memory allocation.

- For example, when you declare a variable as follows :

```
short var ;
```

then, in such a case the computer knows that what the length of a short variable is. Generally, it is 2 bytes. So, a chunk of 2 bytes internal memory will be allocated to variable var during compilation itself. Thus, it is an example of static memory allocation.

DATA TYPE	SIZE (IN BYTE)
char	1
short int	2
int	2
long int	4
float	4
double	8
long double	10
void	MEANING LESS
enum	2

Dynamic Memory Allocation in C++

- When the amount of memory to be allocated is not known beforehand rather it is required to allocate (main) memory as and when required during runtime (when the program is actually executing) itself, then, the allocation of memory at run time is referred to as dynamic memory allocation.
- C++ offers these two operators for dynamic memory allocations : New and Delete
- The operator **new** allocates the memory dynamically and returns a pointer storing the memory address of the allocated memory.
- The operator **delete** deallocates the memory (the reverse of new) pointer by the given pointer.

new Operator:

- The new operator offers dynamic memory allocation similar to the standard library function `malloc()` and `calloc()`.
- The new operator obtains memory at runtime from the memory heap from the operating system and returns the address of the obtained memory.
- The new operator is used as follows:
`data_type *data_type_ptr;`
`data_type_ptr = new data_type; //allocates single variable`
`data_type_ptr = new data_type[size]; //allocates an array`

For example:

```
int *iptr;  
iptr = new int ; //allocates space for single variable  
iptr = new int[n]; //allocates space for an array of integer with n  
elements
```

```
float *fptr;  
fptr = new float ; //allocates space for single variable  
fptr = new float[n]; //allocates space for an array of floats with n  
elements
```

In C, DMA is done as:

```
int *ptr;  
ptr = (int *) malloc (sizeof(int)*n);
```

C++ equivalent is:

```
int *ptr;  
ptr = new int [n];  
where n is input from keyboard.
```


Similarly, to allocate 'n' floating point numbers in C, we write:


```
float *fptr;  
fptr = (float*) malloc (sizeof(float)*n);
```

C++ equivalent is :

```
float *fptr;  
fptr = new float[n];
```

Also we can write:

```
float *fptr = new float[n];
```



The new operator can also be used to initialize the allocated memory location as:

```
int *iptr = new int(14);  
//creates an integer and initializes to 14
```

Or

```
float *fptr = new float(3.79);  
//creates a float and initializes to 3.79
```

Memory deallocation in C++

delete/free Operator

- When we allocate memory using `new` at runtime the memory is reserved even after it is not needed or after the scope of the variable ends.
- After sometime we may need to allocate more memory according to our need.
- Leaving the memory unused is called **memory leak**.
- When the dynamically allocated memory is no longer needed it has to be destroyed to prevent memory leakage.
- The `delete` operator is used to release the memory allocated by the `new` operator back to the operating system memory heap.
- Memory released by the use of `delete` or `free` will be reused by other parts of the program.

The delete operator is used as follows:

```
delete data_type_ptr; // releases a single dynamic variable  
delete []data_type_ptr; //releases dynamically created array
```

For example:

```
int *iptr;  
iptr = new int(4);  
.....  
delete iptr; //or free(iptr);
```

If allocated memory is for multiple elements(arrays):

```
int *iptr;  
iptr = new int[5];  
...  
delete [] iptr;
```

The equivalent to delete statement is:

```
free(data_type_ptr);
```

```

#include <iostream>
#include <conio.h>

using namespace std;

int main()
{
    int *a = new int;
    int *b = new int;
    int *sum = new int;
    cout << "\n enter two numbers:";
    cin>>*a>>*b;
    *sum = *a + *b;

```

```

    cout << "\n the value of sum is: " << *sum<<endl;
    cout <<"before deleting the a,b,sum ptrs:"<<endl;
    cout <<"a:"<<*a<<endl;
    cout <<"b:"<<*b<<endl;
    cout <<"sum:"<<*sum<<endl;
    delete a;
    delete b;
    delete sum;
    cout <<"after deleting the a,b,sum ptrs:"<<endl;
    cout <<"a:"<<*a<<endl;
    cout <<"b:"<<*b<<endl;
    cout <<"sum:"<<*sum<<endl;
    return 0;
}

```

```

enter two numbers:100 200

the value of sum is: 300
before deleting the a,b,sum ptrs:
a:100
b:200
sum:300
after deleting the a,b,sum ptrs:
a:4629344
b:200
sum:5662384

Process returned 0 (0x0)    execution time : 4.805 s
Press any key to continue.

```