Stack Mechanism Discussion with Time Complexity

1. *Creation of Stack*

```
typedef struct Stack
{
    int top;
    int size;
    int *s;

} Stack;

void create(Stack *st)
{
    cout << "Enter the size of the stack" << endl;
    cin >> st->size;
    st->top = -1;
    st->s = (int *)malloc(st->size * sizeof(int));
}
….
        case 1:

            create(&stck);
            cout << "Stack Created" << endl;
            break;
```

Here struct Stack has three member variable which are Integers:

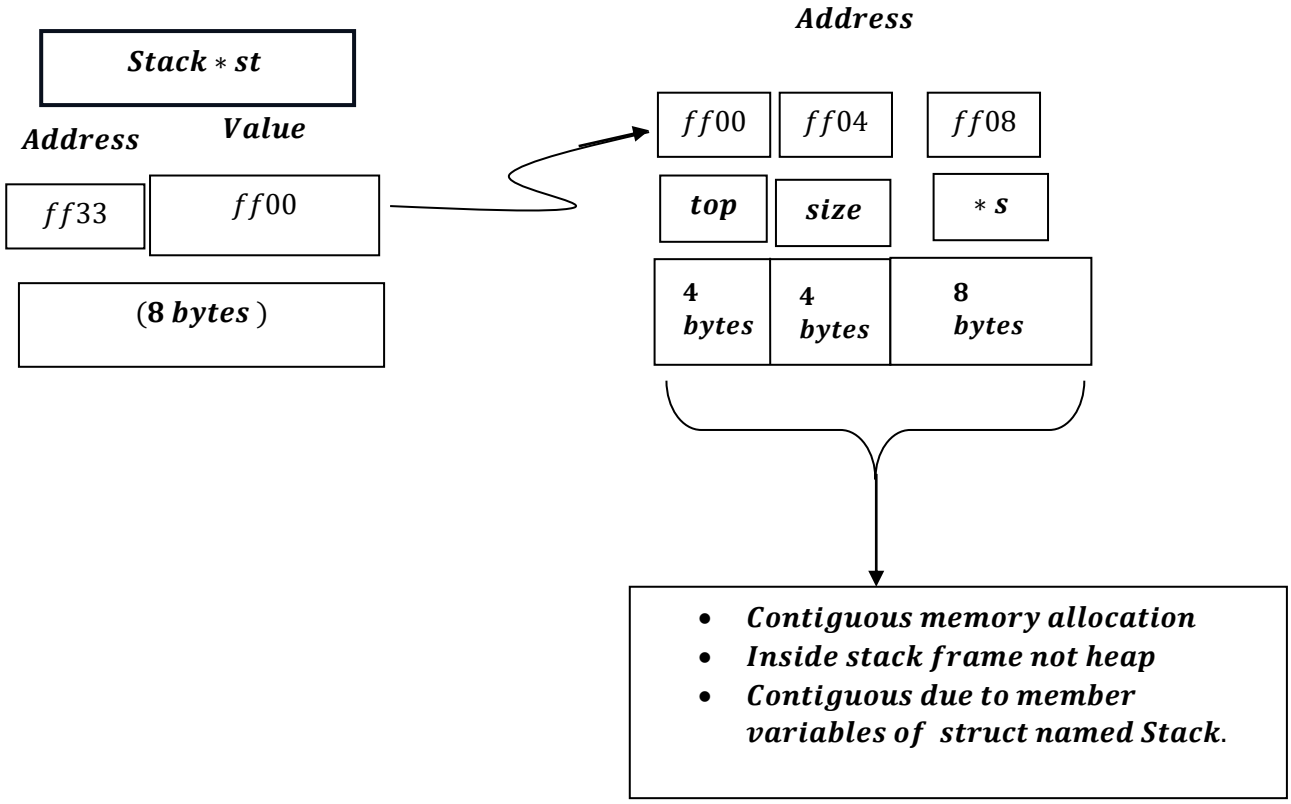
1. *Top*
2. *Size*
3. *Integer Pointer s*

Create function call will create a call stack frame of create function, where it will take input from user `size` and assign `top` of stack to -1 *in initial phase and integer pointer s will obtain an address of a memory location inside heap i.e. s will help to allocate memory inside heap, where size* $*$ *sizeof(int) will get allocated.*

Say size $= 5$ *, then* 5×4 *bytes(size of int* $) = 20$ *bytes of memory will be allocated inside the heap memory.*

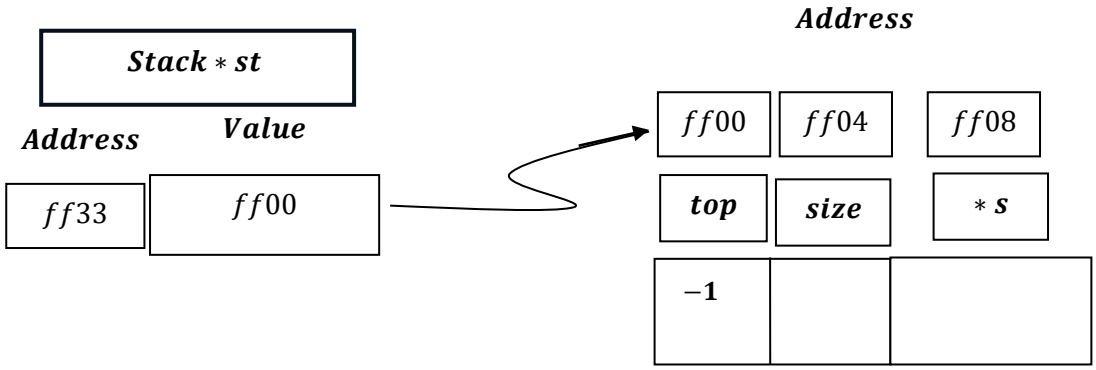
Stack Data structure basically shows how call stack frame works i.e. in LIFO method , but in Stack Data Structure we manually do push() and pop() , where in call stack frame, it creates stack frame for entire function containing local variables , parameters , return address and saved registers and adds or push another stack frame if called again and pops whole stack frame automatically thus different from Stack datastructure , yet conceptually similar.

Actually in Memory:



Now assigning of values:

$st \rightarrow top = -1$



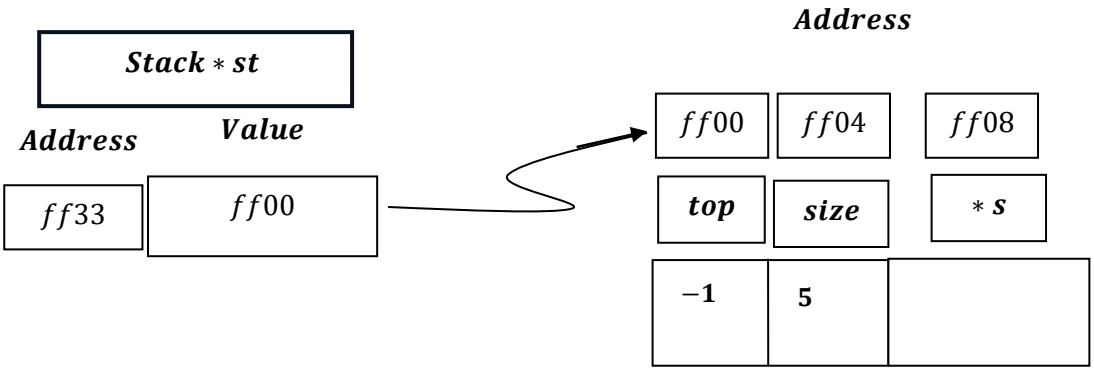
$\boldsymbol{How\ Allocated - 1?}$

$\boldsymbol{\rightarrow Calculation\ is : Base\ Address + offset}$

$\boldsymbol{\rightarrow ff00 + (0 \times 4\ bytes[size\ of\ int]) = ff00 + 0 = -1.}$

$\boldsymbol{Similarly,}$

$\boldsymbol{st \rightarrow size = 5}$



$\boldsymbol{\rightarrow Calculation\ is : Base\ Address + offset}$

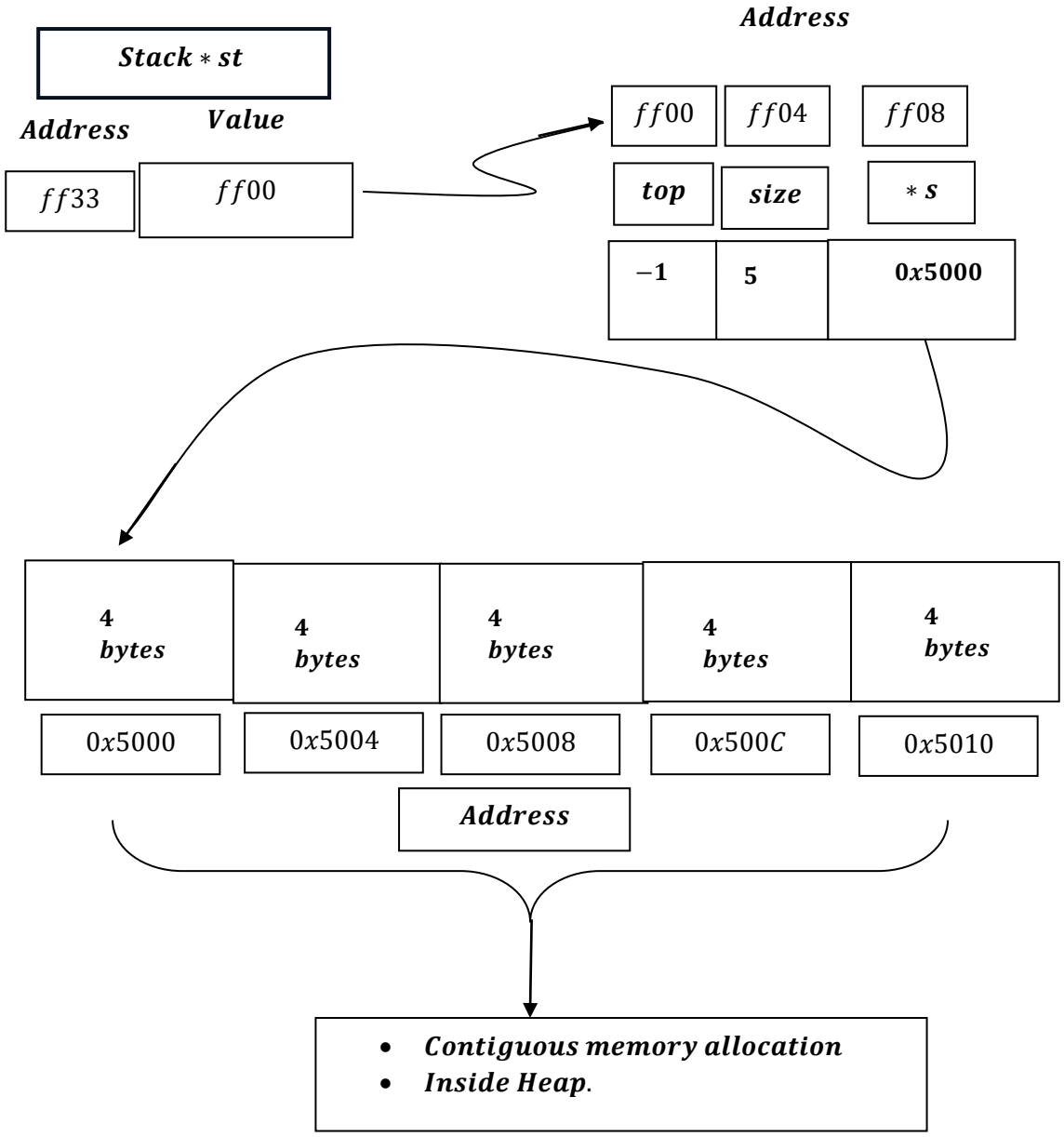
$\boldsymbol{\rightarrow ff00 + (1 \times 4\ bytes) = ff00 + 4 = 5.}$

$\boldsymbol{Now\ most\ important\ part:}$

```
    st->s = (int *)malloc(st->size * sizeof(int));
```

$\boldsymbol{i.e.\ size = 5\ and\ sizeof(int) = 4\ bytes = 5 \times 4\ bytes = 20\ bytes.}$

$\boldsymbol{\therefore\ s\ will\ hold\ a\ base\ memory\ address\ of\ 4\ bytes\ in\ heap\ and\ there\ will\ be\ 4\ more\ contiguous\ memory\ addresses\ in\ heap\ contributing\ 20\ bytes\ of\ space.}$



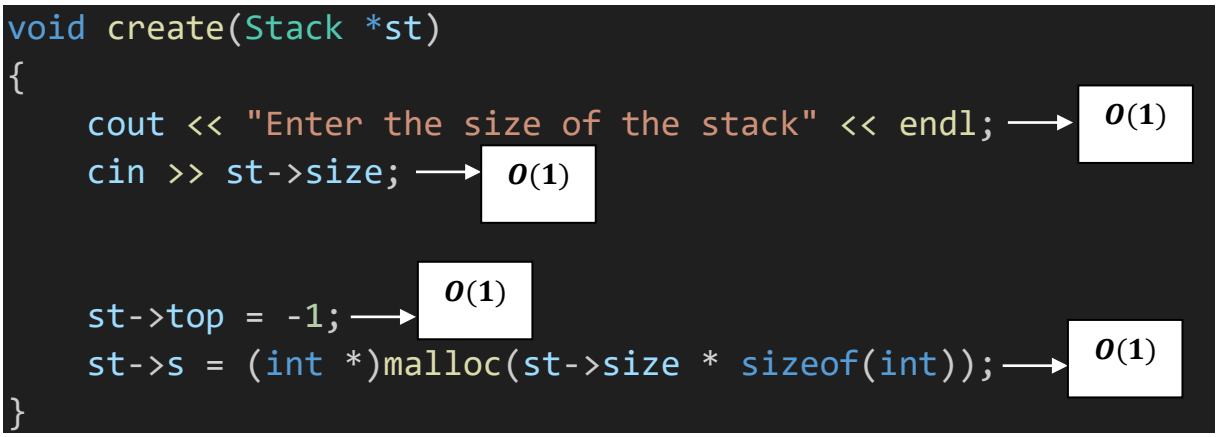
$$ff00\begin{bmatrix}\textbf{Base}\\\textbf{Address}\end{bmatrix} + 2(\textbf{index}) \times 4\ \textbf{bytes} = ff00 + 8 = ff08 = 0x5000$$

During Static allocation of memory say a[5], the memory is allocated at compile time, memory is reserved before program runs, Cannot change size later[Physical size]though logically we can change. Such during deletion in program but that doesnot change the physical size of array which always remain 5.

During Dynamic memory allocation , memory is allocated at run − time, user enters size at run − time, memory allocated based on user − input, hence compiler didnot know size before hand and as size was decided during program execution run − time, (using heap)it is dynamic, but dynamic doesnot mean memory keep changing automatically.

We can free the old memory allocated to heap logically also applies physically and again resize it physically only happens in dynamic memory allocation.

Time Complexity of creation of Stack



1. *Print statement will take `c` amount of time i.e.* $O(1)$.
2. *User input size will also take `c` unit of time i.e.* $O(1)$.
3. *Assigining top also will take `c` unit of time i.e.* $O(1)$.
4. *Memory allocation will also take `c` unit of time i.e.* $O(1)$.

$c \rightarrow$ *stands for constant.*

Hence total time time complexity $\rightarrow O(1) + O(1) + O(1) + O(1) = O(1)$.

Note: $O(1)$ ***is not counted for n time but `c` constant amount of time.***