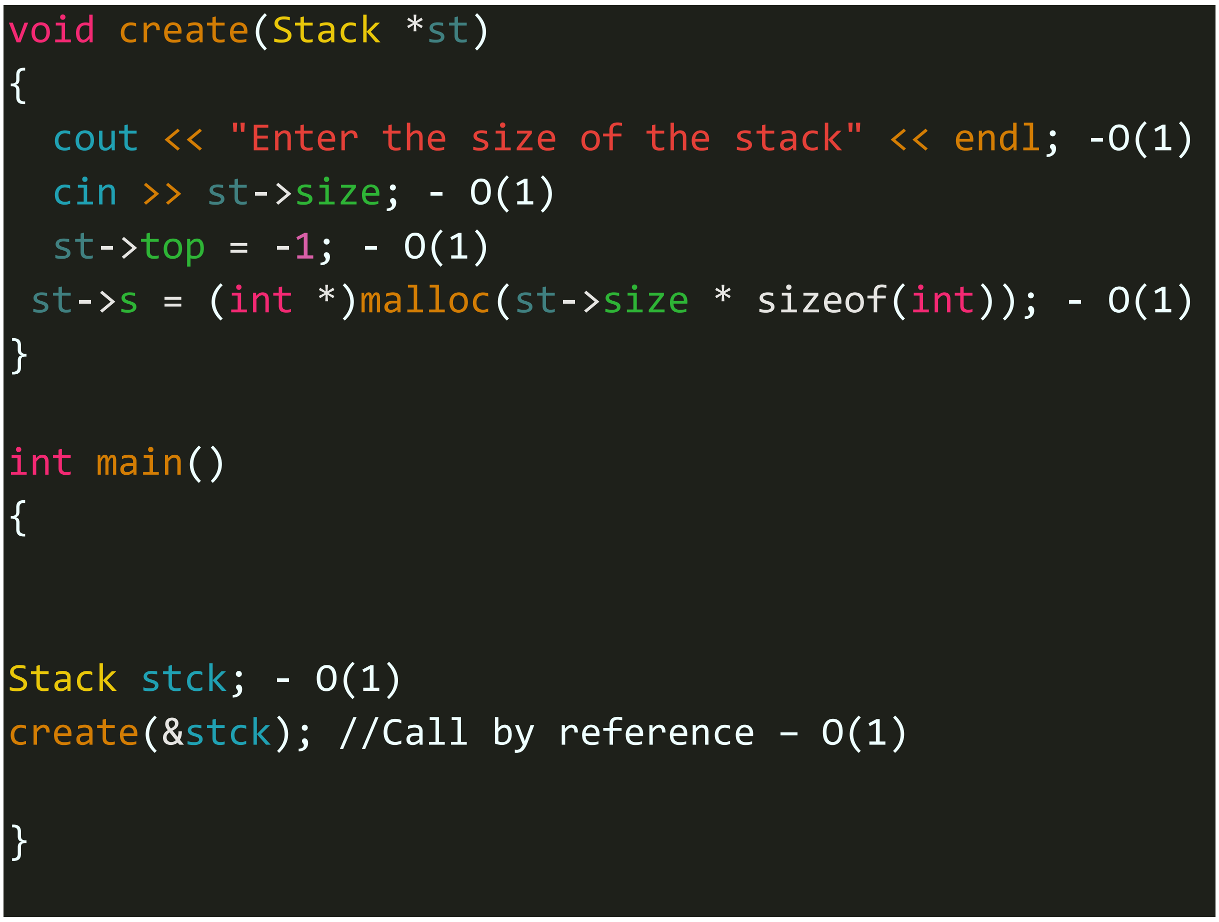
Calculation of Time Complexity of Stack

1. *Creation*



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

int main()
{


Stack stck; - O(1)
create(&stck); //Call by reference – O(1)

}
```

Hence Time Complexity of Creation of Stack =

$$\boldsymbol{O}(\mathbf{1}) + \boldsymbol{O}(\mathbf{1}) + \boldsymbol{O}(\mathbf{1}) + \boldsymbol{O}(\mathbf{1}) + \boldsymbol{O}(\mathbf{1}) + \boldsymbol{O}(\mathbf{1})$$

$$= \boldsymbol{O}(\mathbf{1})$$

2. ***Push***

```
void push(Stack *st, int x)
{
    if (st->top == st->size - 1) → O(1)
    {
        cout << "Stack Overflow" << endl;
    }
    else → O(1)
    {
        st->top++;
        st->s[st->top] = x;
    }
}

int main()
{

   int x; → O(1)
   cin >> x; → O(1)
   push(&stck, x); //Call by reference → O(1)

}
```

Hence Time Complexity of Insertion =

$$\boldsymbol{O(1) + O(1) + O(1) + O(1) + O(1) = O(1)}$$

Now, $O(1)$ *is for single element*

what will be for `n` times operation as we know

n times $1 = n$ *, hence it must generate* $O(n)$,

but here we have analyze according to Stack

Operation .

<u>*Stack Push Operation [Time Complexity]*</u>

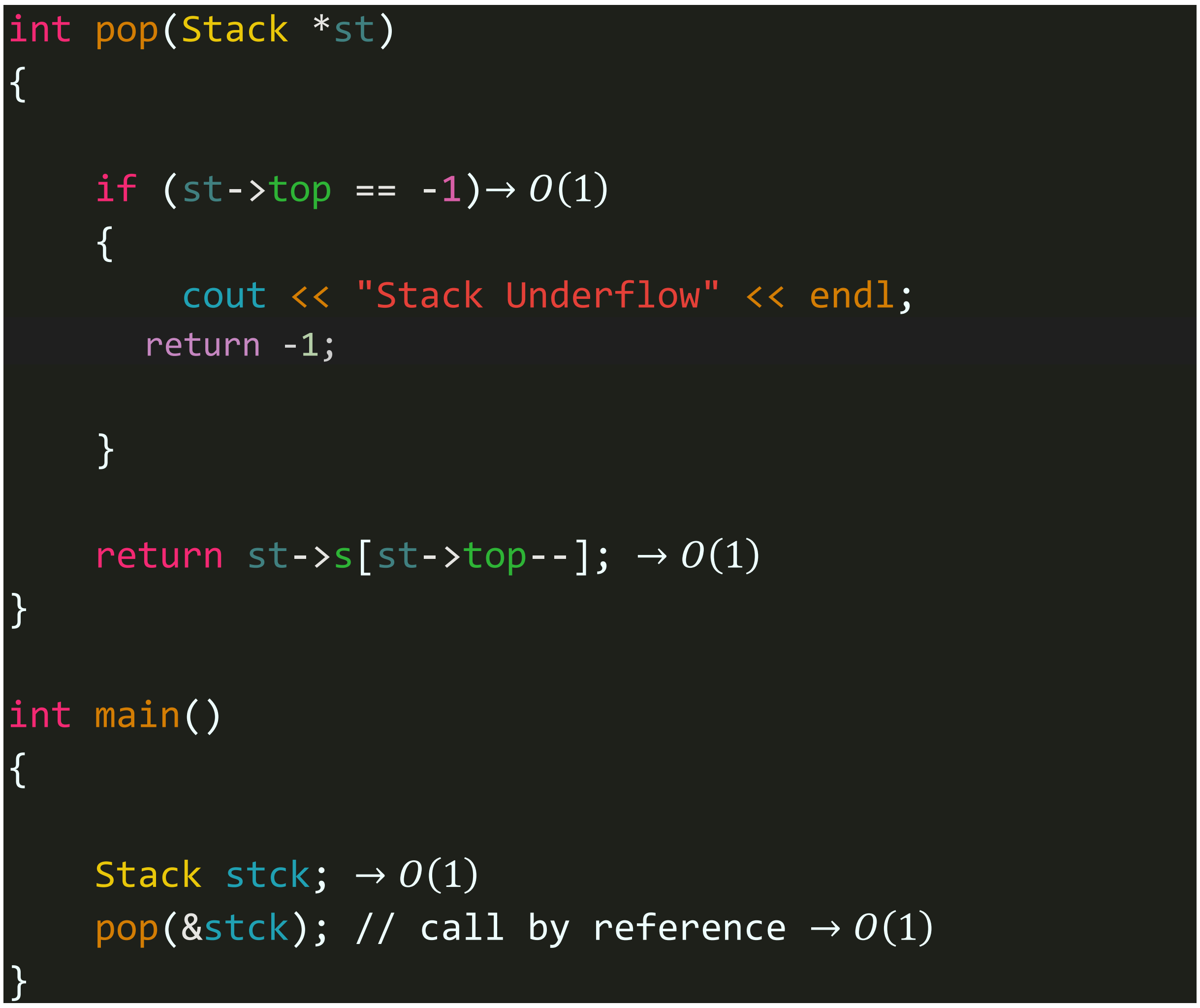
[*Single Element Pushed*]	$\rightarrow O(1)$
[*Single Element Pushed*]	$\rightarrow O(1)$
[*Single Element Pushed*]	$\rightarrow O(1)$
[*Single Element Pushed*]	$\rightarrow O(1)$
[*Single Element Pushed*]	$\rightarrow O(1)$
[*Single Element Pushed*]	$\rightarrow O(1)$
………	
[*Single Element Pushed*]	$\rightarrow O(1)$

Hence for an element Push will be performed in the top of particular stack which generates $O(1)$ *time compexity*.

Hence, $O(1) + O(1) + O(1) + \cdots + O(1)$ *for each Push Operation* $= O(1)$

Also when stack overflow i.e. Top = STACK SIZE − 1, it prints ``Stack Overflow`` and along with if condition check whether Top = STACK SIZE − 1 , this whole operation takes constant time: O(1) apart from PUSH operation.

3. <u>*Pop Operation*</u>



```
int pop(Stack *st)
{

    if (st->top == -1)→ O(1)
    {
        cout << "Stack Underflow" << endl;
      return -1;

    }

    return st->s[st->top--]; → O(1)
}

int main()
{

    Stack stck; → O(1)
    pop(&stck); // call by reference → O(1)
}
```

<u>Hence Time Complexity of Pop =</u>

$$O(1) + O(1) + O(1) + O(1) = O(1)$$

Now, $O(1)$ ***is for single element***

what will be for `n` times operation as we know

n times $1 = n$ ***, hence it must generate*** $O(n)$,

but here we have analyze according to Stack

Operation .

<u>Stack Pop Operation [Time Complexity]</u>

[*Single Element Popped*]	$\rightarrow O(1)$
[*Single Element Popped*]	$\rightarrow O(1)$
[*Single Element Popped*]	$\rightarrow O(1)$
[*Single Element Popped*]	$\rightarrow O(1)$
[*Single Element Popped*]	$\rightarrow O(1)$
[*Single Element Popped*]	$\rightarrow O(1)$
………	
[*Single Element Popped*]	$\rightarrow O(1)$

Hence for an element (First Element)gets Popped, hence Pop function will gets executed in a particular stack which generates $\boldsymbol{O(1)}$ time compexity.

Hence, $\boldsymbol{O(1) + O(1) + O(1) + \cdots + O(1)}$ ***for each Pop Operation*** $\boldsymbol{= O(1)}$.

Also when stack underflow i.e. no element present, it prints ``Stack Underflow`` and return $\boldsymbol{-1}$ ***, along with if condition check whether*** $\boldsymbol{Top = -1}$ ***, this whole operation takes constant time:*** $\boldsymbol{O(1)}$ ***apart from poping.***

4. *Is Empty Operation*

```
int isEmpty(Stack st)
{
    if (st.top == -1)→ O(1)
    {
        return 1;
    }
    return 0;→ O(1)
}
int main()
{

    Stack stck;→ O(1)
     isEmpty(stck); → O(1)

}
```

$$\boldsymbol{Time\ Complexity = O(1) + O(1) + O(1) + O(1)}$$

$$\boldsymbol{= O(1)}$$

5. *Is Full Operation*

```
int isFull(Stack st)
{
    return st.top == st.size - 1;→ O(1)
}

int main()
{

    Stack stck;→ O(1)
   isFull(stck);→ O(1)
}
```

$$\boldsymbol{Time\ Complexity = O(1) + O(1) + O(1) = O(1)}$$

6. *Traversal Of Stack*

```
void Display(Stack st)
{

    for (int i = st.top; i >= 0; i--)→ O(n)
    {
        cout << st.s[i] << " ";
    }
    cout << endl;
}
```

$$\boldsymbol{Time\ Complexity = O(n)}$$

7. *Peek Operation*

```
int peek(Stack st)
{
    if (!isEmpty(st)) → O(1)
    {
        return st.s[st.top];
    }
    return -1;→ O(1)
}
int main()
{

 Stack stck;→ O(1)

 peek(stck); → O(1)

}
```

$$\boldsymbol{Time\ Complexity = O(1) + O(1) + O(1) + O(1)}$$

$$\boldsymbol{= O(1)}$$

$8.$ *Deletion of Stack*

```
free(stck.s); → O(1)
 stck.s = NULL; → O(1)
```

$$\boldsymbol{Time\ Complexity = O(1) + O(1) = O(1)}$$

There fore we can sum up Time Complexity Operation as:

Operation	*Time Complexity*
Creation of Stack	$O(1)$
Push()	$O(1)$
Pop()	$O(1)$
Peek()	$O(1)$
Traversal	$O(n)$
IsEmpty()	$O(1)$
IsFull()	$O(1)$
Deletion of Stack	$O(1)$

-----------------*****************---------------------------