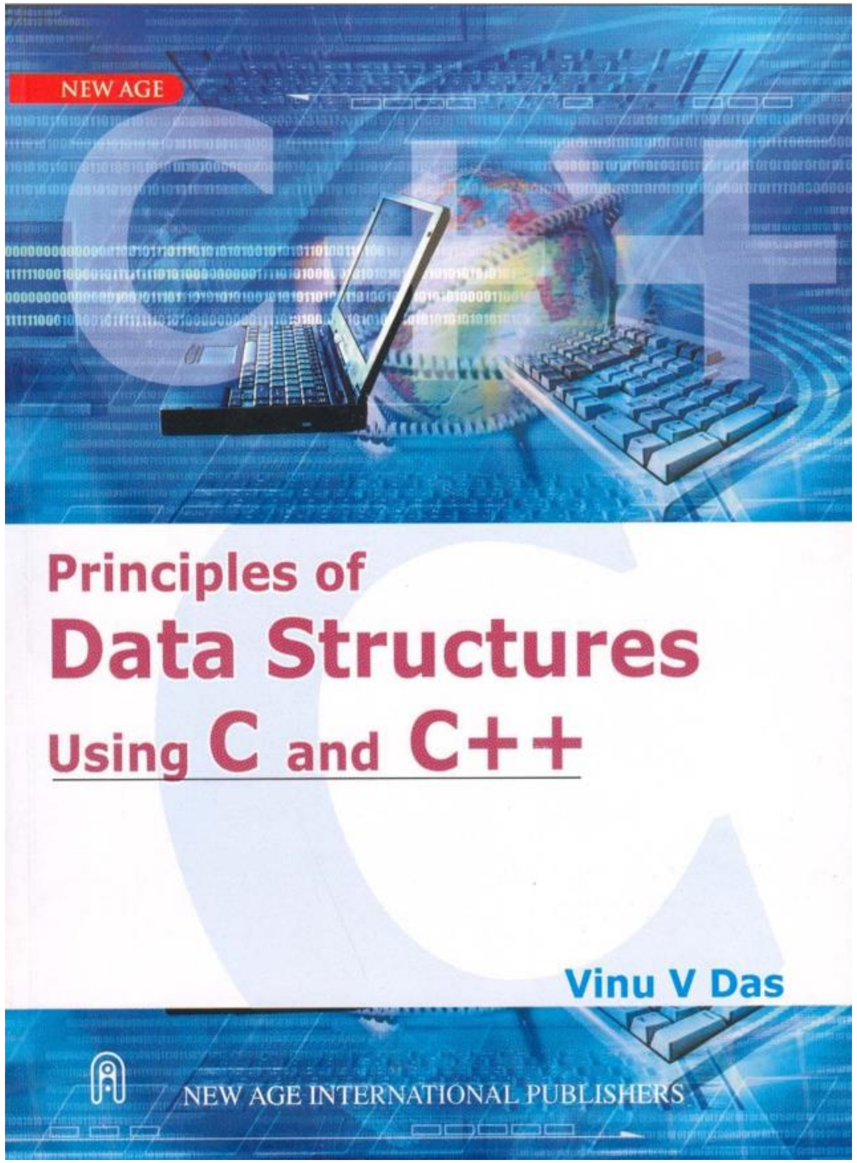
Data Structure

Lecture 2

Dr. Ahmed Fathalla

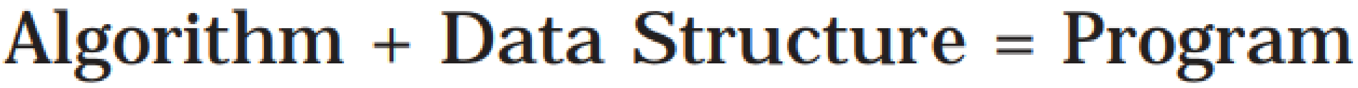
Resources

Book: Principles of Data Structures Using C and C++



Data structure

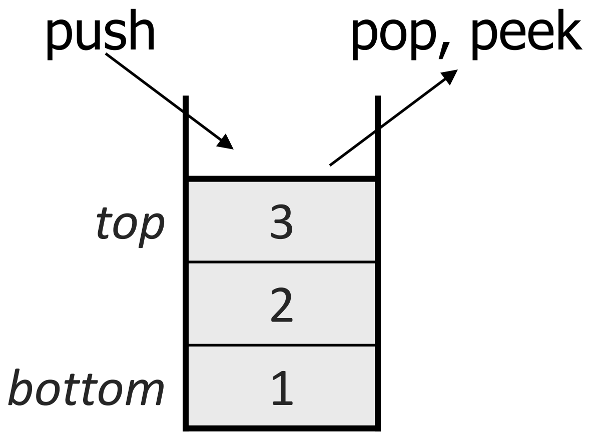
- Data structure is the ***structural representation of logical relationships between elements of data***.
  In other words a data structure is ***a way of organizing data items by considering its relationship to each other (Section 1.1)***.
- Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity.
- Data structure affects the design of both the structural and functional aspects of a program.



Algorithm + Data Structure = Program

Stack

- **Stack**: It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the top of the stack. (Chapter 3)
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (the "top").



push

pop, peek

top

3

2

bottom

1

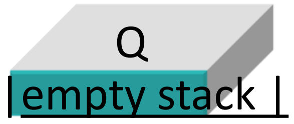
stack

Motivation: What and Why `Stacks`?

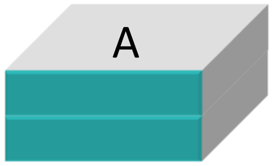
basic stack operations:

- **push**: Add an element to the top.
- **pop**: Remove the top element.
- **peek**: Examine the top element.

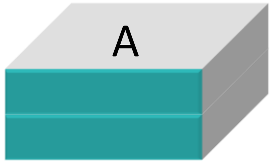
Push box Q onto empty stack:



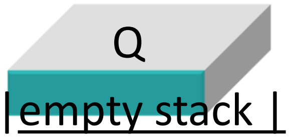
Push box A onto stack:



Pop a box from stack:



Pop a box from stack:

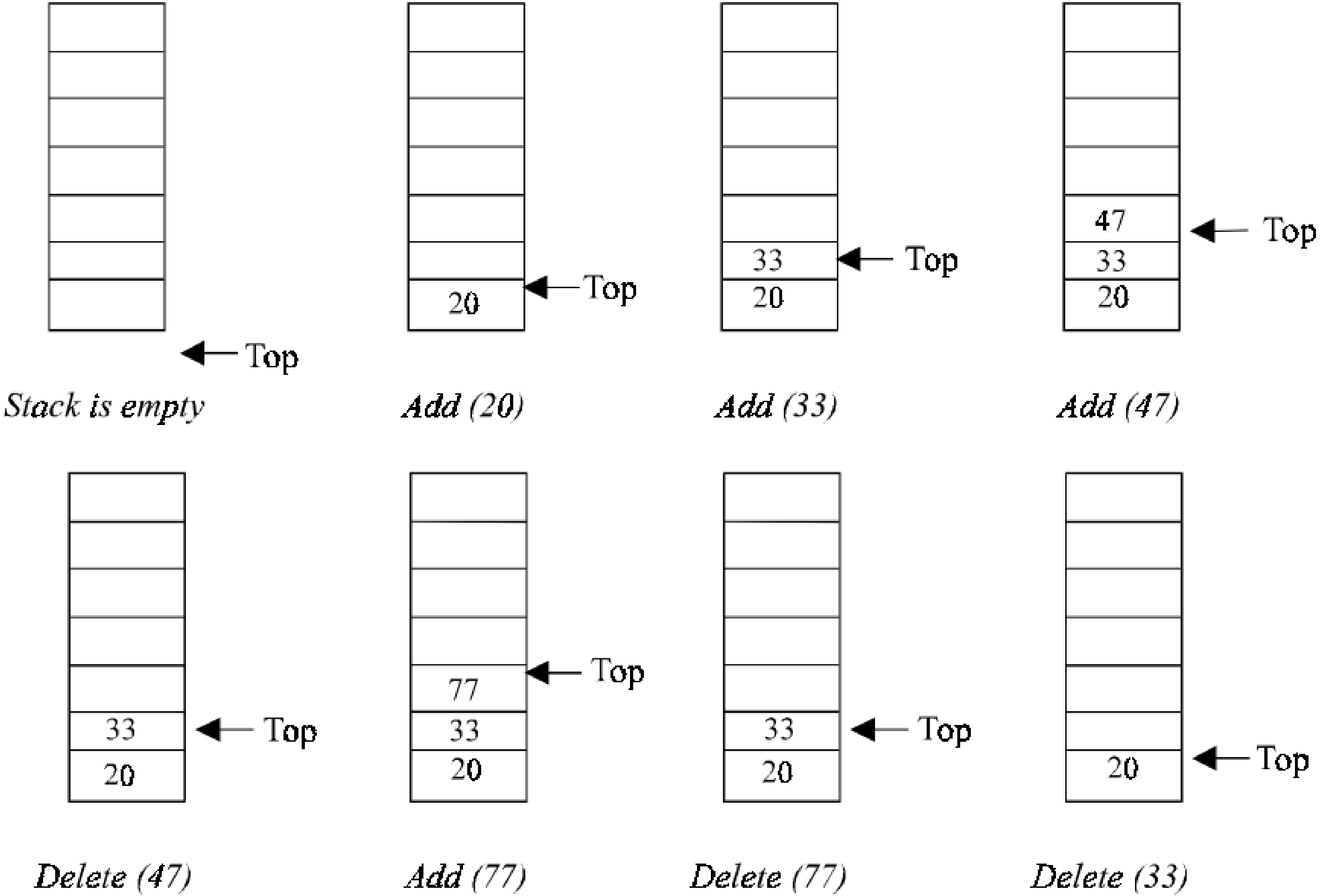


Stack implementation

Stack can be implemented in two ways:

- Static implementation (Array-based implementation).
- Dynamic implementation (Linked-based implementation).

Static implementation (using arrays)



Stack is empty *Add (20)* *Add (33)* *Add (47)*

Delete (47) *Add (77)* *Delete (77)* *Delete (33)*

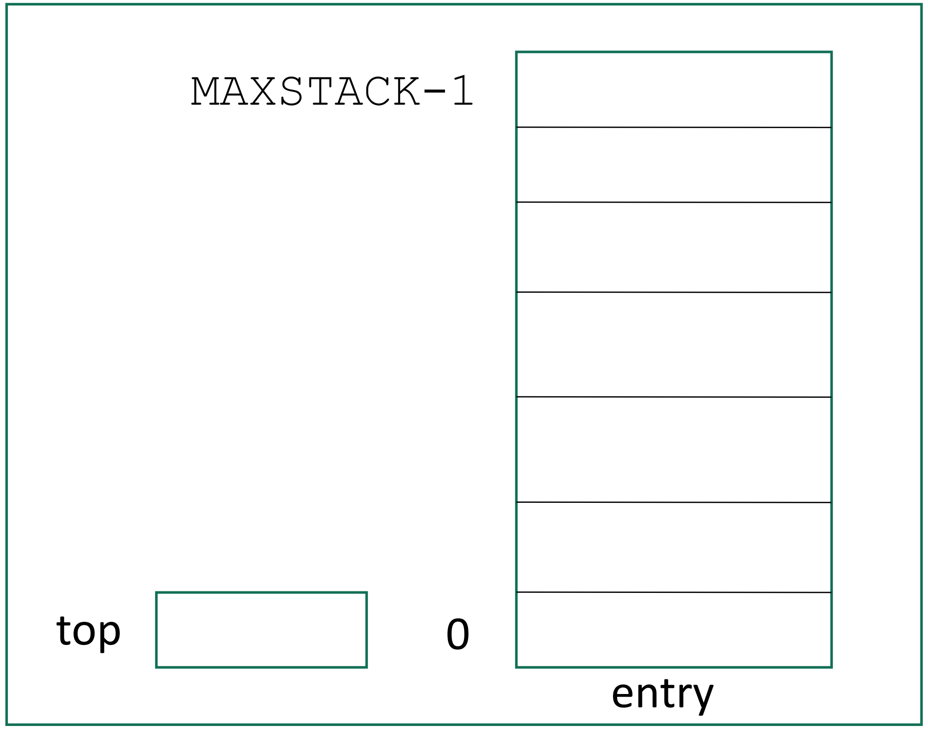
Definition: ***Abstract Data Type (ADT)*** is a data type that is accessed only through an **interface** (or **Accessing mechanism**). We refer to a program that uses an ADT as a **client** (or **user level**) and a program that specifies the data type as an **implementation level.**

Definition: ***Stack*** of elements of type *T is a finite sequence of elements of T together* with the following operations:

1. **Create** the stack, leaving it empty.
2. Determine whether the stack is **empty or not**.
3. Determine whether the stack is **full or not**.
4. **Find the size** of the stack.
5. **Push** a new entry onto the top of the stack, provided the stack is not full.
6. **Pop** the entry off the top of the stack, provided the stack is not empty.
7. **Retrieve** the Top entry off the stack, provided the stack is not empty.
8. **Traverse** the stack, visiting each entry.
9. **Clear** the stack to make it empty.

Static implementation (using arrays)

```
struct Stack{
   int top;
   StackEntry entry[MAXSTACK];
};
```



MAXSTACK-1

top

0

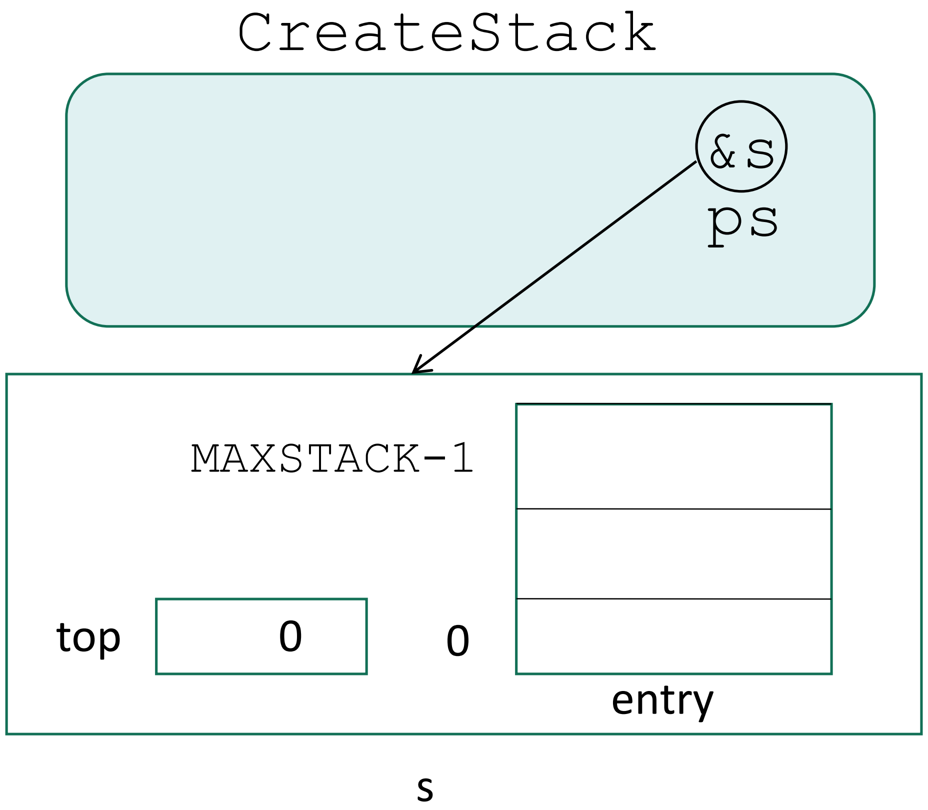
entry

struct stack ⍰

`StackEntry` and `MAXSTACK` should be defined in the User Level.

Implementation level (what really happens)

```
void CreateStack(Stack *ps){
   ps->top=0;
}
```



User Level (interface)

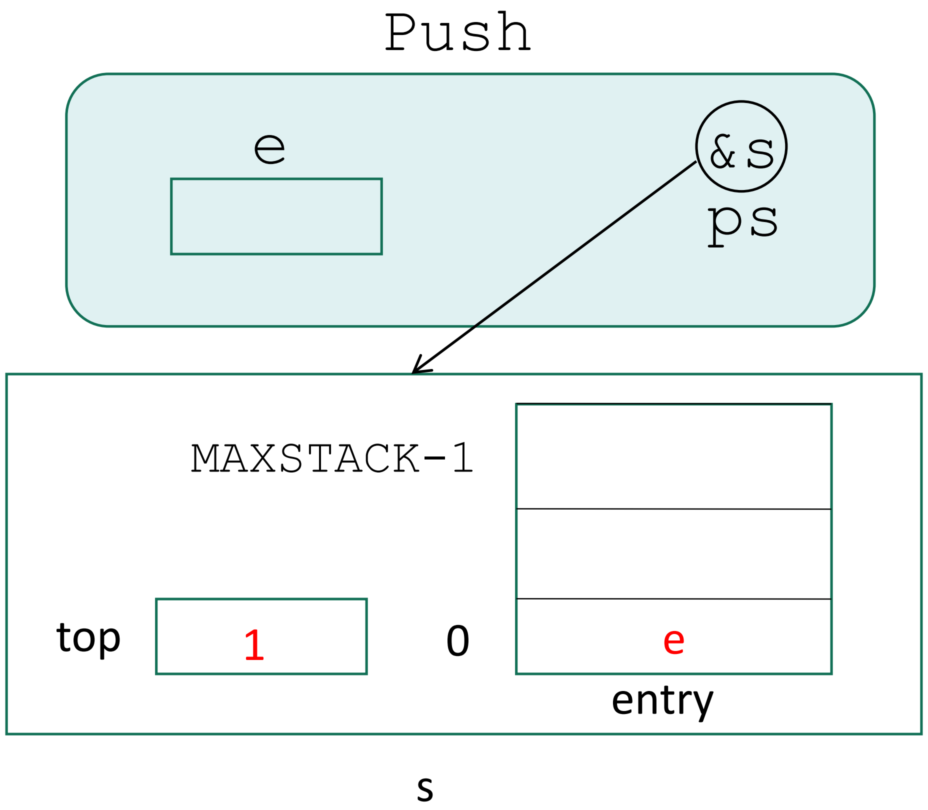
```
void main(){

Stack s;

CreateStack(&s);



}
```

`top` is the index of the first available place.

Implementation level (what really happens)

User Level (interface)

```
void Push(StackEntry e, Stack *ps){
    ps->entry[ps->top]=e;  ⎫
    ps->top++;             ⎭ ps->entry[ps->top++]=e;
}
```



```
void main(){
StackEntry e;
Stack s;
   :
CreateStack(&s);
       :
Push(e, &s);

}
```

```
void Push(StackEntry e, Stack *ps){
   ps->entry[ps->top++]=e;
}
```

The user has to check before calling `Push`

Other ways (no precondition) are:

```
if (ps->top==MAXSTACK)
   printf(“Stack is full”);
else ps->entry[ps->top++]=e;
//but this is not professional
```

```
int Push(…){
   if (ps->top==MAXSTACK)
    return 0;
   else {
    ps->entry[ps->top++]=e;
    return 1;
   }//This is fine
```

```
void main(){
StackEntry e;
Stack s;
   :
CreateStack(&s);
     :
if (!StackFull(&s))
   Push(e, &s);
}
```

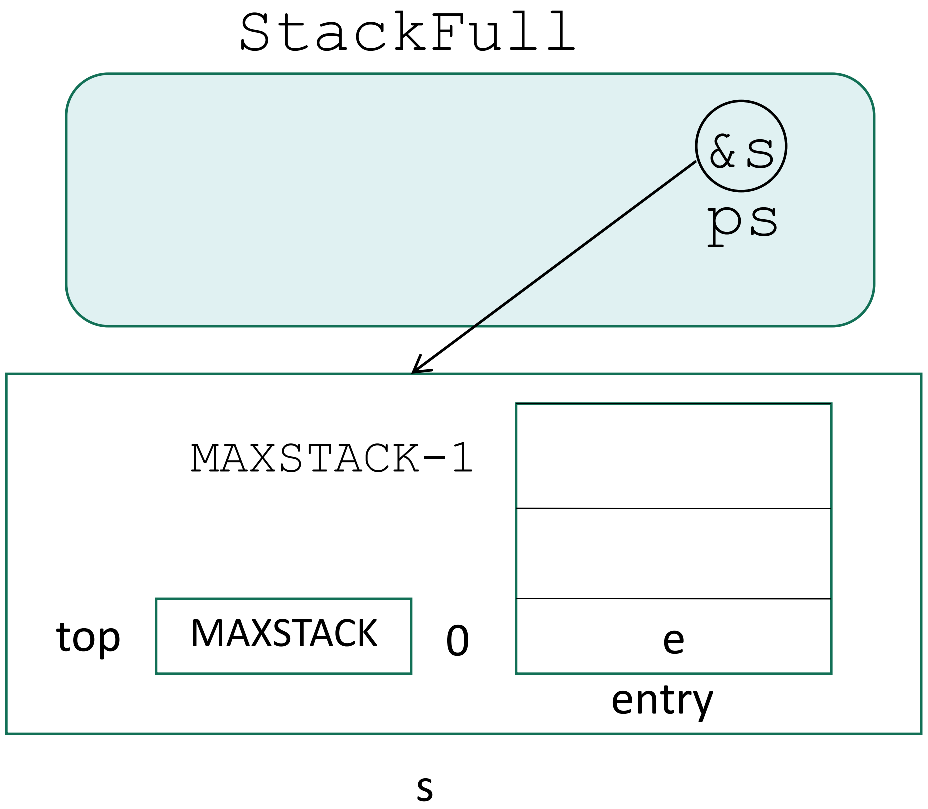
```
if (!Push(e, &s))
   …
```

Implementation level (what really happens)

```c
int StackFull(Stack *ps){
    if (ps->top==MAXSTACK)
        return 1;
    else
        return 0;
}
```

User Level (interface)

```c
return ps->top >= MAXSTACK;
```



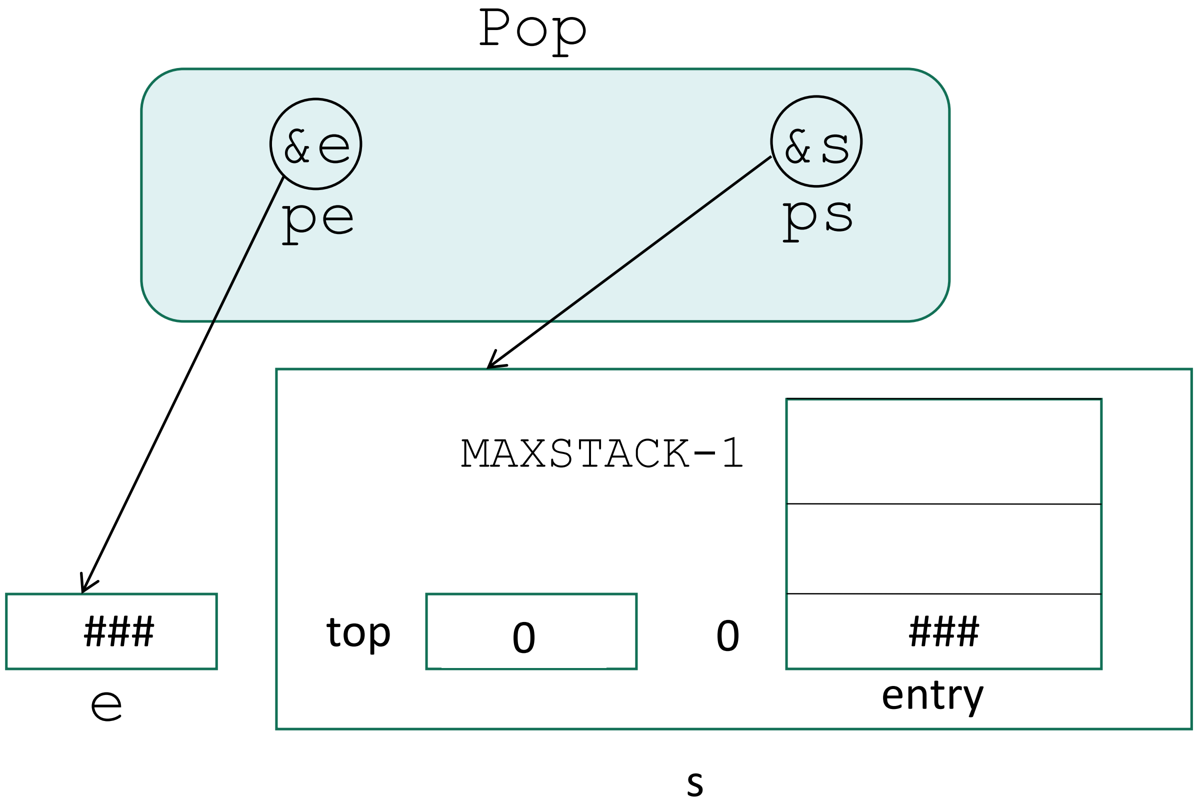
```c
void main(){
StackEntry e;
Stack s;
    :
CreateStack(&s);
        :
if (!StackFull(&s))
    Push(e, &s);
}
```

It could be: `StackFull(s)` but this wastes memory and time of copying.

Implementation level
(what really happens)

User Level
(interface)

```
void Pop(StackEntry *pe, Stack *ps){
  ps->top--;                    } *pe=ps->entry[--ps->top];
  *pe=ps->entry[ps->top];       }
}
```



```
void main(){
StackEntry e;
Stack s;
       :
CreateStack(&s);
       :
if (!StackFull(&s))
   Push(e, &s);
       :
Pop(&e, &s);

}
```

```
void Pop(StackEntry *pe, Stack *ps){
   *pe=ps->entry[--ps->top];
}
```

The user has to check before calling `Pop`

Other ways (no precondition) are:

```
if (ps->top==0)
   printf(“Stack is Empty”);
else *pe=ps->entry[--ps->top];
//but this is not professional
```

```
int Pop(…){
   if (ps->top==0)
    return 0;
   else {
    *pe=ps->entry[--ps->top];
    return 1;
   }//This is fine
```

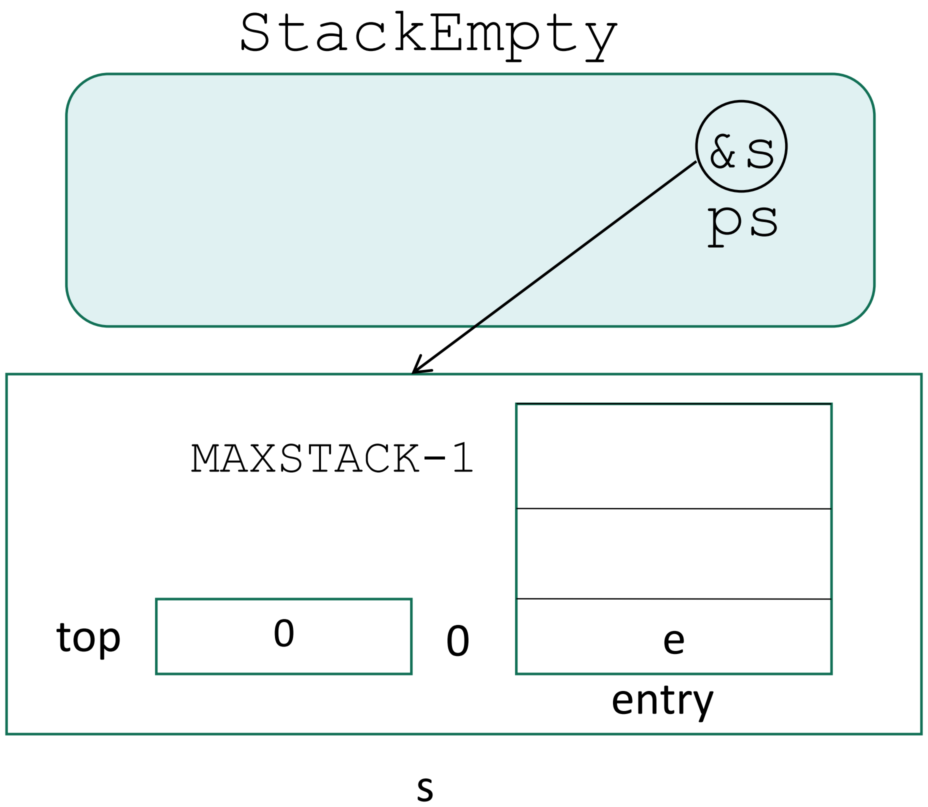
```
void main(){
StackEntry e;
Stack s;
   :
CreateStack(&s);
      :
if (!StackEmpty(&s))
   Pop(&e, &s);
}
```

```
if (!Pop(&e, &s))
   …
```

Implementation level (what really happens)

```
int StackEmpty(Stack *ps){
    if (ps->top==0)
        return 1;
    else
        return 0;
}
```

```
return !ps->top;
```



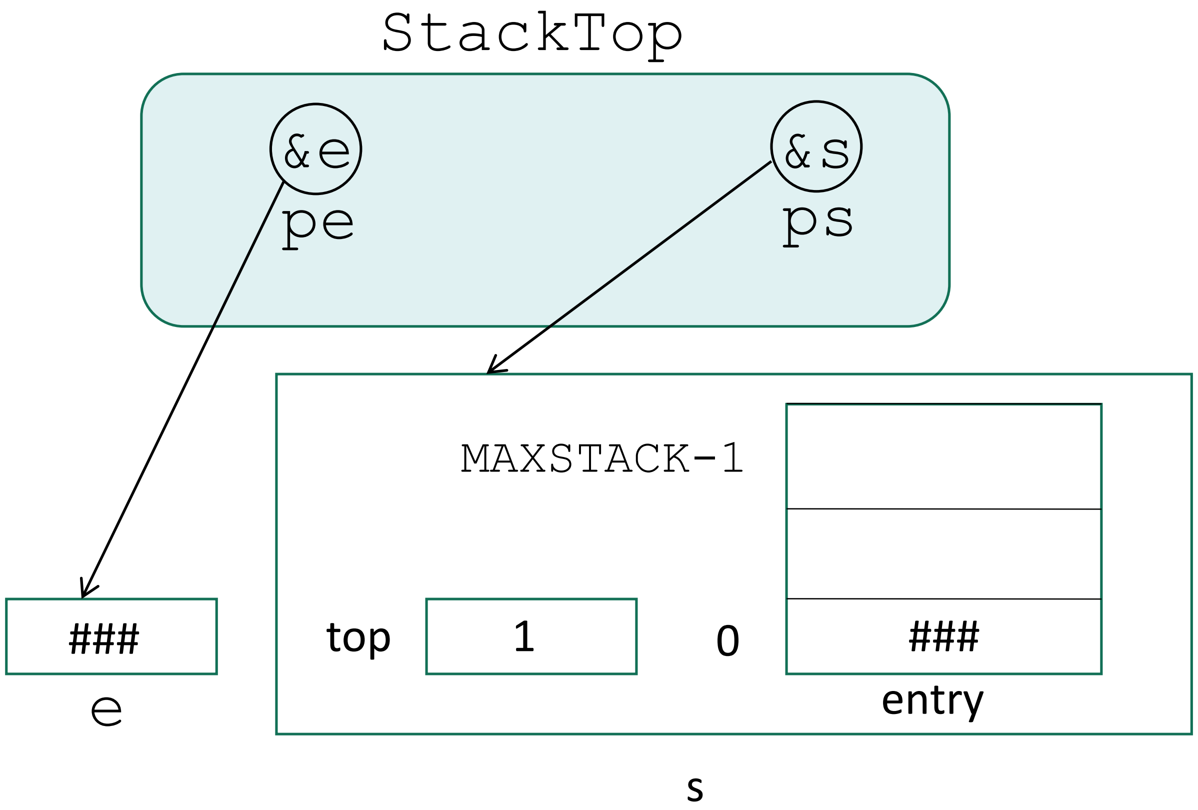
User Level (interface)

```
void main(){
StackEntry e;
Stack s;
    :
CreateStack(&s);
        :
if (!StackEmpty(&s))
    Pop(&e, &s);
}
```

It could be: `StackEmpty(s)` but this wastes memory and time of copying.

Implementation level (what really happens)

```
//Same preconditions of Pop.
void StackTop(StackEntry *pe, Stack *ps){
   *pe=ps->entry[ps->top-1];
}
```



User Level (interface)

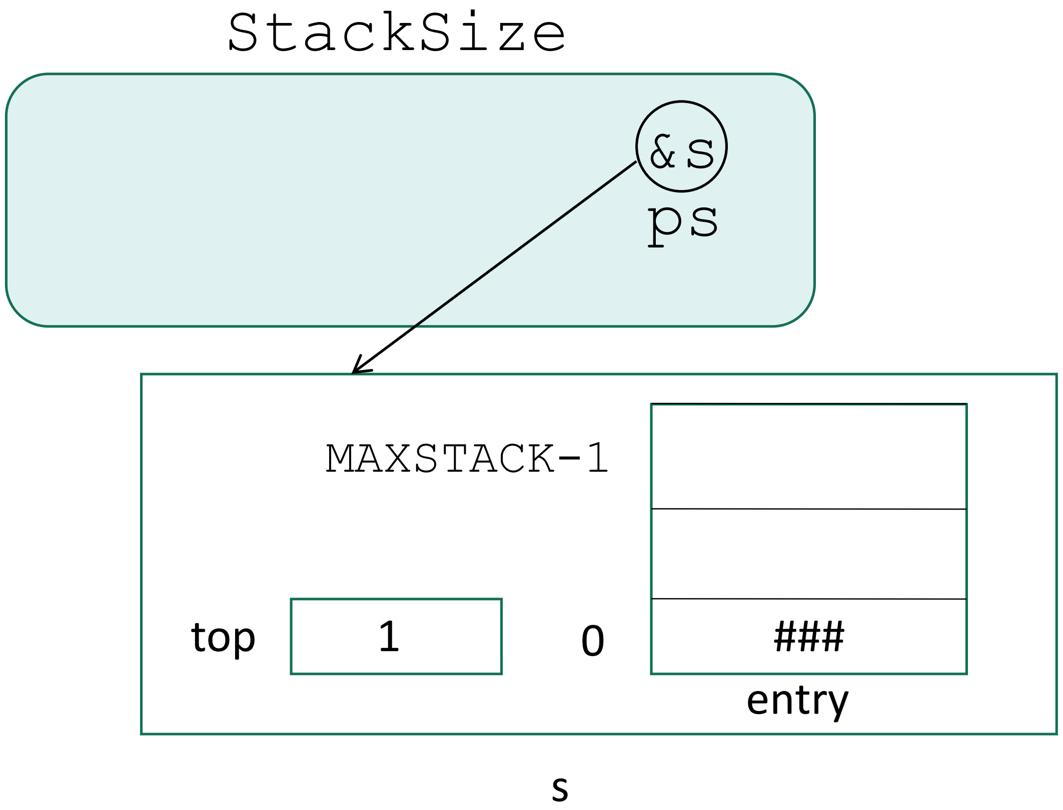
```
void main(){
StackEntry e;
Stack s;
   :
CreateStack(&s);
      :
StackTop(&e, &s);
}
```

It could be: `StackTop(&e, s)` but this wastes memory and time of copying

Implementation level (what really happens)

User Level (interface)

```
/*Pre: Stack is initialized.
Post: returns how many elements exist.
int StackSize(Stack *ps){
   return ps->top;
}
```



```
void main(){
StackEntry e;
Stack s;
int x;
    :
CreateStack(&s);
       :
x=StackSize(&s);
}
```

It could be:

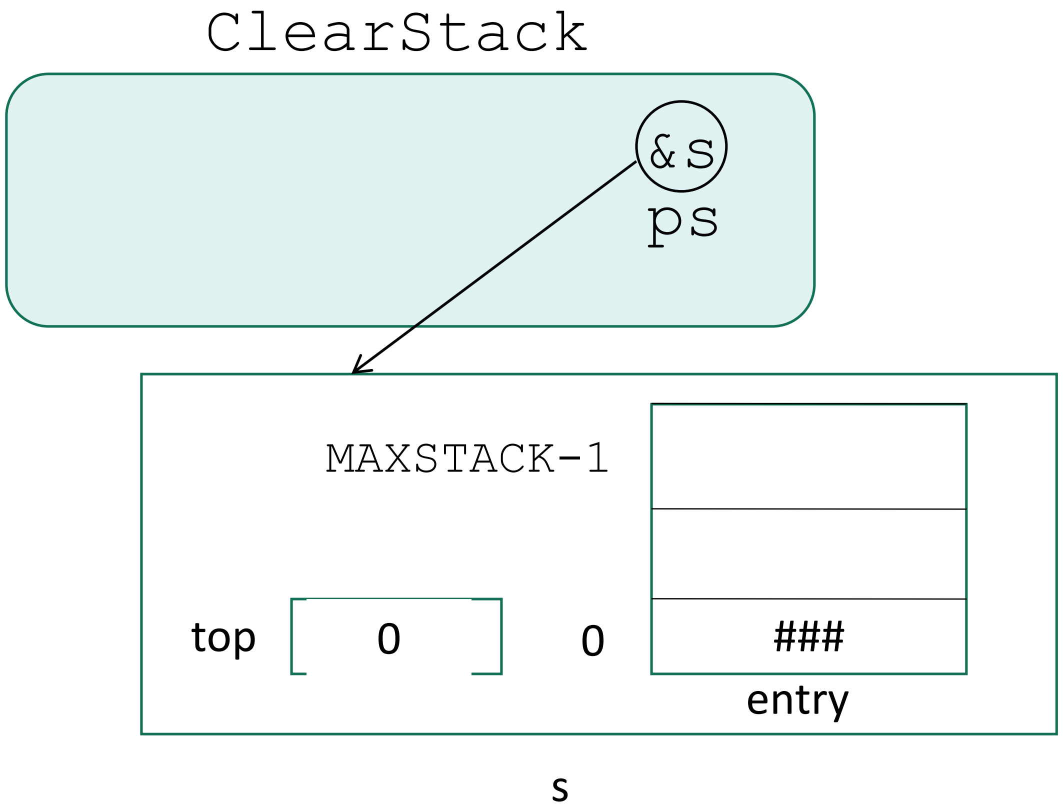
`StackSize(s)`

but this wastes memory and time of copying

Implementation level (what really happens)

User Level (interface)

```
/*Pre: Stack is initialized.
Post: destroy all elements; stack looks initialized.
void ClearStack(Stack *ps){
   ps->top=0;
}
```



```
void main(){
StackEntry e;
Stack s;
   :
CreateStack(&s);
      :
ClearStack(&s);
}
```

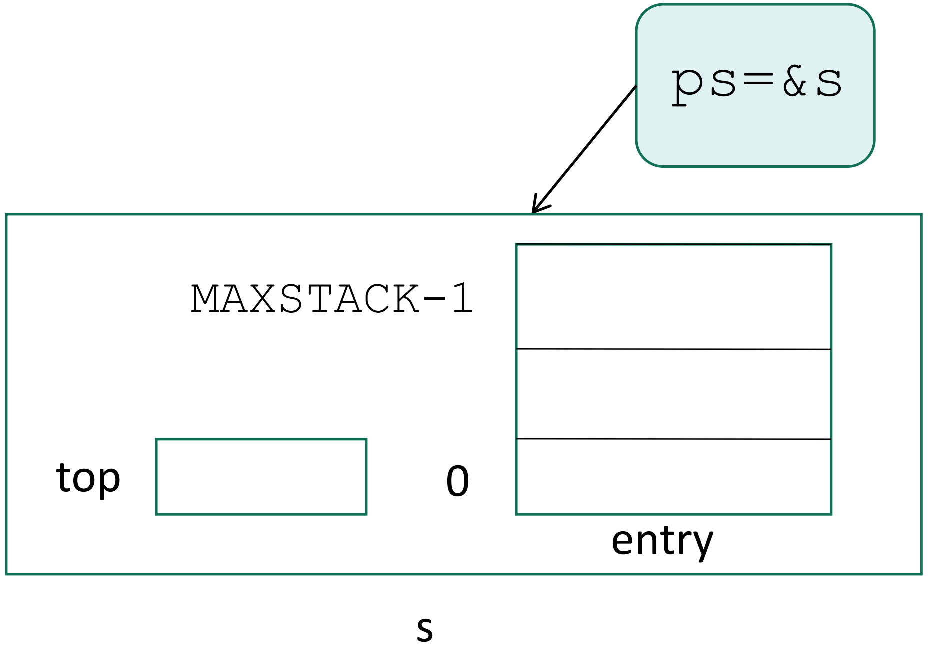
Same code as `CreateStack`; why new function then?

1- conceptually

2- will see later

Implementation level

```
//Precondition: The stack is Initialized
void TraverseStack(Stack *ps){
   for(int i=ps->top; i>0; i--)
       cout<<ps->entry[i-1]<<" ";
}
```

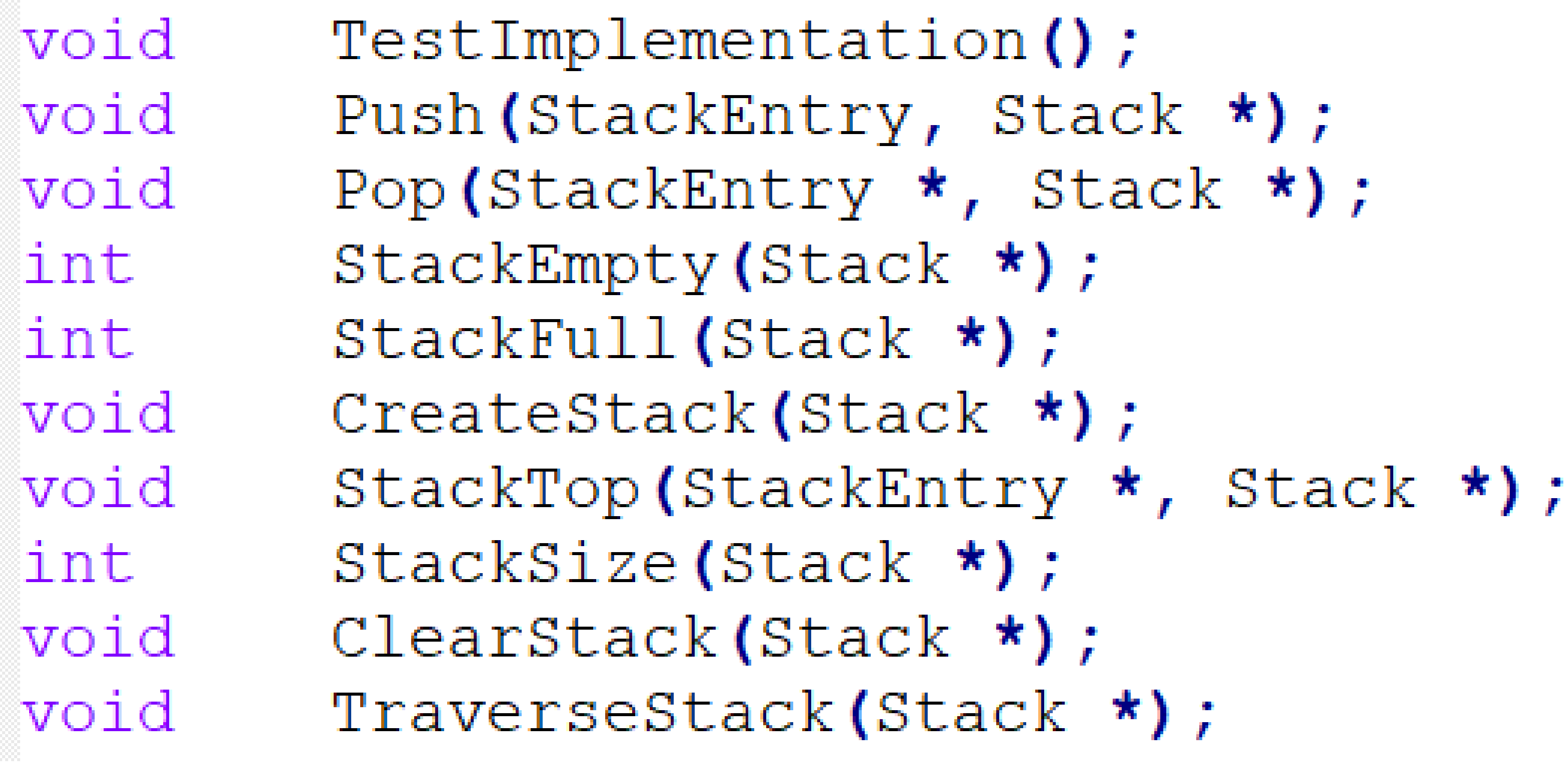


User Level:

how to process each element with a user-defined function

```
void main(){

Stack s;

CreateStack(&s);
   .
   .
TraverseStack(&s);
}
//&s only for efficiency as
said before.
```

Stack Accessing mechanism



```
void    TestImplementation();
void    Push(StackEntry, Stack *);
void    Pop(StackEntry *, Stack *);
int     StackEmpty(Stack *);
int     StackFull(Stack *);
void    CreateStack(Stack *);
void    StackTop(StackEntry *, Stack *);
int     StackSize(Stack *);
void    ClearStack(Stack *);
void    TraverseStack(Stack *);
```

Exercise: How to write the function `StackTop` in the user level? (e.g., if you do not have the source code of the implementation)

User Level:

```
void StackTop(StackEntry *pe, Stack *ps){
  Pop(pe, ps);
  Push(*pe, ps);
}

void main(){
StackEntry e;
Stack s;
   :
CreateStack(&s);
      :
StackTop(&e, &s);
}
```