Chapter 6: Stacks

Data Structures and Algorithms with C

LIFO | Stack ADT | push / pop | Overflow | Linked stack | Call stack | Multiple stacks

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_06_Stacks.html

Learning Outcomes

Define LIFO

Model a stack as a list $S = \{a_0, \ldots, a_{n-1}\}$ and identify its bottom and top.

Implement with an array

Write push/pop around a `top` index and check for overflow and underflow.

Build a linked stack

Allocate nodes with `malloc`, link them, and `free` each node you remove.

Understand the call stack

Explain why function calls behave like a stack and why the call stack can overflow.

Make design decisions

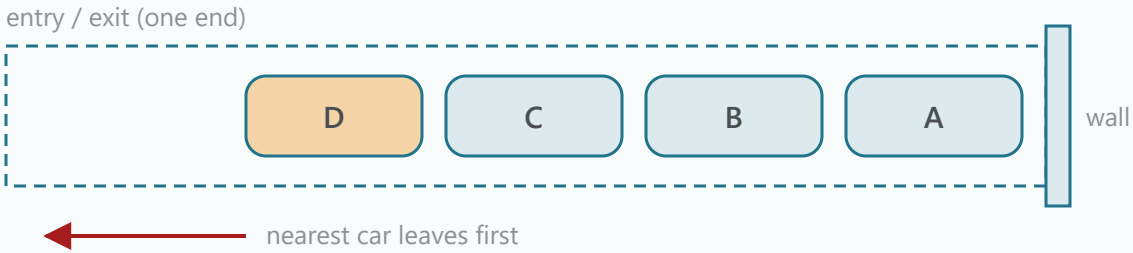
Choose between a fixed array, a growable array, and a linked list by cost.

Manage multiple stacks

Share one memory array among n stacks using `top[i]` and `boundary[i]`.

The Stack: Last In, First Out

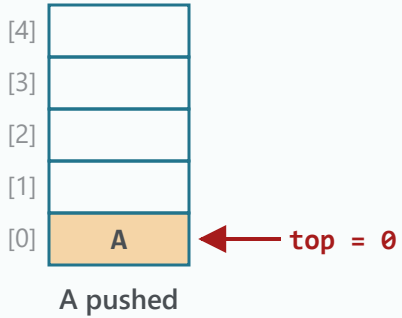
A stack is an ordered list in which all insertions and deletions happen **at one end only**. For $S = \{a_0, a_1, \ldots, a_{n-1}\}$, a_0 is the **bottom** and a_{n-1} is the **top**. The last item in is the first one out: **LIFO** (last-in-first-out).



Cars parked in a one-car-wide dead end: the last one in (D) has to leave first.

Don't confuse the stack ADT with the **heap** data structure (Chapter 13) or with the heap memory that `malloc` manages.

How a Stack Grows and Shrinks

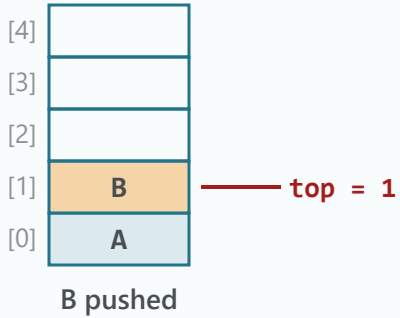


A pushed

A pushed → A is now on top; top = 0.

- Push order: A, B, C, D, E.
- Pop order (if we removed them all): E, D, C, B, A, **exactly reversed**.
- To reach an item in the middle (say B), everything above it must be popped first.
- This restriction is not a flaw but a contract: “the last job opened is the first one closed.”

How a Stack Grows and Shrinks

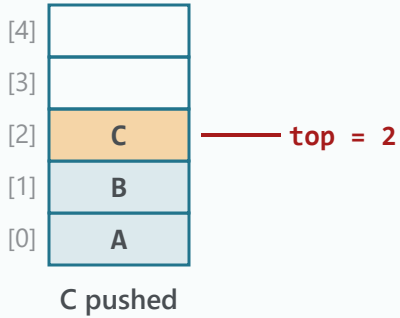


B pushed

B pushed → B is now on top; top = 1.

- Push order: A, B, C, D, E.
- Pop order (if we removed them all): E, D, C, B, A, **exactly reversed**.
- To reach an item in the middle (say B), everything above it must be popped first.
- This restriction is not a flaw but a contract: “the last job opened is the first one closed.”

How a Stack Grows and Shrinks

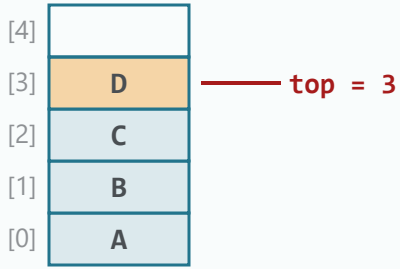


C pushed

C pushed → C is now on top; top = 2.

- Push order: A, B, C, D, E.
- Pop order (if we removed them all): E, D, C, B, A, **exactly reversed**.
- To reach an item in the middle (say B), everything above it must be popped first.
- This restriction is not a flaw but a contract: “the last job opened is the first one closed.”

How a Stack Grows and Shrinks

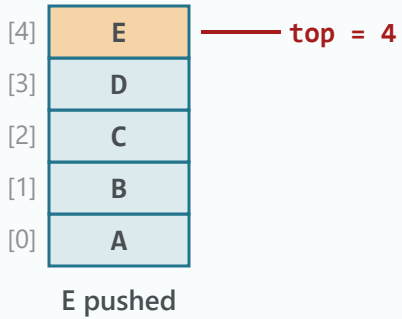


D pushed

D pushed → D is now on top; top = 3.

- Push order: A, B, C, D, E.
- Pop order (if we removed them all): E, D, C, B, A, **exactly reversed**.
- To reach an item in the middle (say B), everything above it must be popped first.
- This restriction is not a flaw but a contract: “the last job opened is the first one closed.”

How a Stack Grows and Shrinks

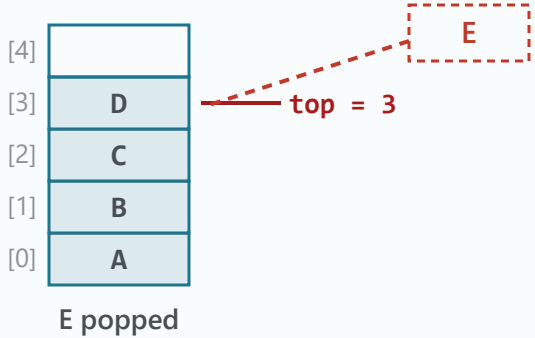


E pushed

E pushed → E is now on top; top = 4.

- Push order: A, B, C, D, E.
- Pop order (if we removed them all): E, D, C, B, A, **exactly reversed**.
- To reach an item in the middle (say B), everything above it must be popped first.
- This restriction is not a flaw but a contract: “the last job opened is the first one closed.”

How a Stack Grows and Shrinks



Deletion happens **only at the top**: the first item removed is E, the last one added. D is on top again; top = 3.

- Push order: A, B, C, D, E.
- Pop order (if we removed them all): E, D, C, B, A, **exactly reversed**.
- To reach an item in the middle (say B), everything above it must be popped first.
- This restriction is not a flaw but a contract: “the last job opened is the first one closed.”

Abstract Data Type: Stack

Operation	Meaning	Precondition / error
`stack_create(max)`	Empty stack holding at most *max* items	—
`stack_full(S)`	True if it holds max items	—
`push(S, x)`	Puts x on top	**Overflow** if full
`stack_empty(S)`	True if it holds no items	—
`pop(S)`	Removes and returns the top item	**Underflow** if empty
`stack_top(S)` · peek	Reads the top item without removing it	Error if empty

An ADT says **what** the operations do, not **how**. The same interface can be implemented with a fixed array, a growable array, or a linked list (Horowitz–Sahni, ADT 3.1).

Where Stacks Show Up

Call stack
Every function call pushes a frame holding its return address and local variables.

Expression evaluation
Compilers use a stack to convert infix to postfix and to evaluate postfix.

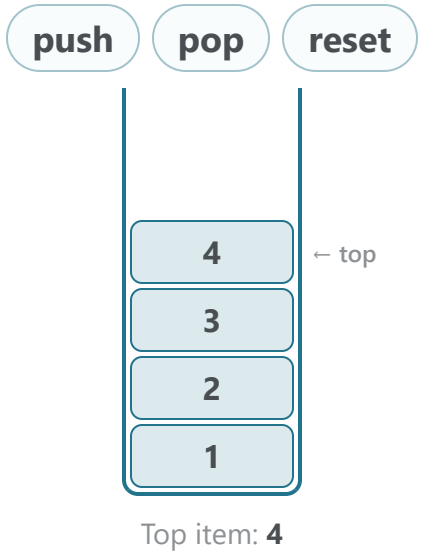
Backtracking
At a dead end in a maze, go back to the most recent fork.

Bracket matching
Every closing bracket must match the most recently opened one.

Undo
An editor pushes every action; Ctrl+Z undoes the latest one.

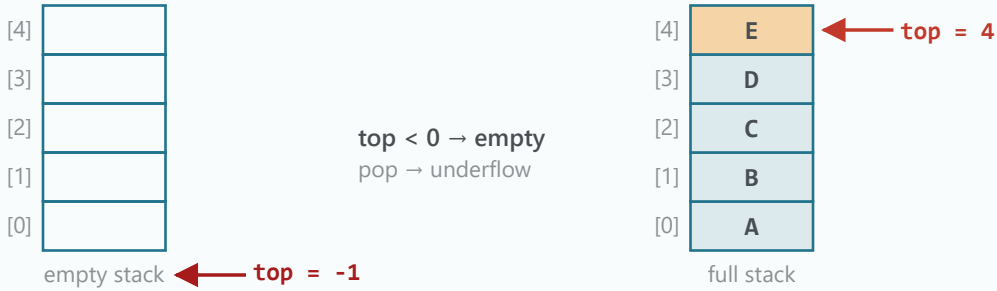
Depth-first search
DFS keeps the vertices still to visit on a stack.

Push and Pop Only at the Top



Push four times, then pop: items come out in reverse order. Popping an empty stack is **underflow**.

Array-Based Stack: The `top` Index



Empty: top < 0

An empty stack has **top = −1**. Popping from it is an **underflow** error.

Full: top ≥ MAX_STACK_SIZE − 1

Pushing onto it is an **overflow** error.

Item count = top + 1

Mixing the two up is the classic off-by-one bug.

What goes wrong?

C does not check array bounds: an unchecked push overwrites neighboring memory.

Array-Based Stack: C Definitions

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 101

typedef struct {
    int key;
    /* other fields */
} element;

element stack[MAX_STACK_SIZE];

void stack_full(void) {
    printf("Stack is full!!!\n"); exit(-1);
}
void stack_empty(void) {
    printf("Stack is empty!!!\n"); exit(-2);
}
```

- `element`: the record stored on the stack; for now its only field is `key`.
- The stack is a **global** array; `top` is passed in **by address** from the caller.
- The error routines print a message and end the program with `exit`.

INSTRUCTOR "Do it this way and you get a zero, maybe even negative points. How many times have we covered this here?"

Push and Pop Routines

```
void push(int *top, element item) {
    if (*top >= MAX_STACK_SIZE - 1)  /* any room? */
        stack_full();          /* warn and exit */
    stack[++*top] = item;      /* bump the index first */
}                              /* then store the item */

element pop(int *top) {
    if (*top < 0)
        stack_empty();         /* warn and exit if empty */
    return stack[(*top)--];    /* return, then decrement */
}
```

`stack[++*top]`

`*top` is incremented first, **then** the item is written at the new index: on an empty stack, −1 → 0.

`stack[(*top)--]`

The item at the old index is read first, **then** `*top` is decremented. The parentheses matter: `*top--` would decrement the pointer itself.

INSTRUCTOR “Normally, a variable that appears as ‘*param’ in a routine's formal parameter list must also be used as ‘*param’ inside the routine.”

A Precedence Trap in C: `*` with `++` / `--`

Expression	How C parses it	Result
`++*top`	`++(*top)`: increment the int it points to	Correct (push)
`(*top)--`	use the int it points to, then decrement it	Correct (pop)
`*top--`	`*(top--)`: decrement the **pointer**, read the old address	**Wrong**: top is unchanged
`*top++`	`*(top++)`: advance the pointer	**Wrong**

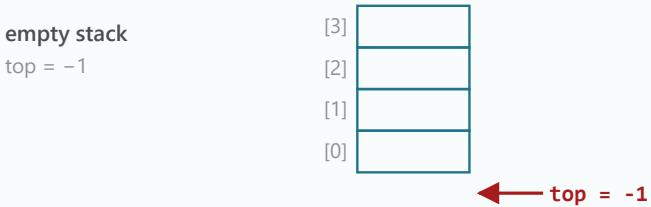
Rule

Postfix `++` / `--` binds **tighter** than unary `*`; prefix `++` and `*` share a level and associate right to left.

Why pass the address?

`top` belongs to the caller; we pass `&top` so the routine can change it (C passes arguments by value).

Code Trace: Array-Based Stack



In `main`, **top = −1** (empty). Push A, B, C; pop; push D, E, F. MAX_STACK_SIZE = 4.

Step 1 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

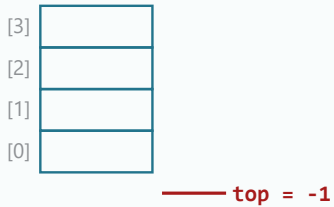
`top(main) = -1` `stack = []`

CALL STACK

`main()`

OUTPUT

Code Trace: Array-Based Stack



-1 ≥ 3? No, there is room.

Step 2 / 15

```
1  void push(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

`*top = -1` `item = A` `stack = []`

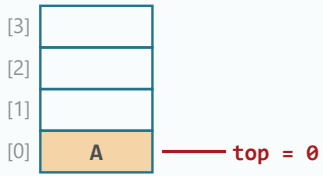
CALL STACK

`push(&top, A)`

`main()`

OUTPUT

Code Trace: Array-Based Stack



++*top → top = 0, then **stack[0] = A**.

Step 3 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
▶    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

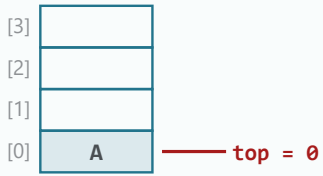
*top = 0 | item = A | stack = [A]

CALL STACK

push(&top, A)
main()

OUTPUT

Code Trace: Array-Based Stack



0 ≥ 3? No, there is room.


```
1  void push(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

*top = 0 | item = B | stack = [A]

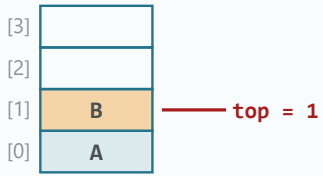
CALL STACK

push(&top, B)

main()

OUTPUT

Code Trace: Array-Based Stack



++*top → top = 1, then **stack[1] = B**.

Step 5 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
▶    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

*top = 1 item = B

stack = [A, B]

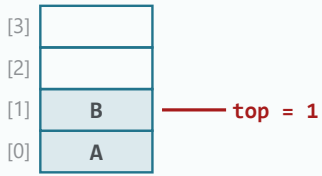
CALL STACK

push(&top, B)

main()

OUTPUT

Code Trace: Array-Based Stack



1 ≥ 3? No, there is room.

Step 6 / 15

```
1  void push(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

`*top = 1` `item = C`

`stack = [A, B]`

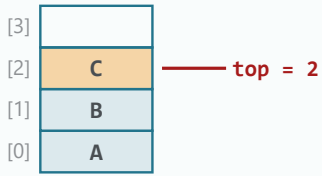
CALL STACK

`push(&top, C)`

`main()`

OUTPUT

Code Trace: Array-Based Stack



++*top → top = 2, then **stack[2] = C**.

Step 7 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
▶    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

*top = 2 | item = C

stack = [A, B, C]

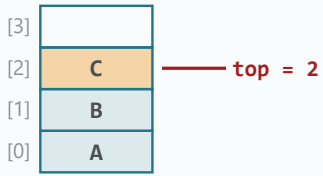
CALL STACK

push(&top, C)

main()

OUTPUT

Code Trace: Array-Based Stack



2 < 0? No, the stack is not empty.

Step 8 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
▶    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

`*top = 2` `stack = [A, B, C]`

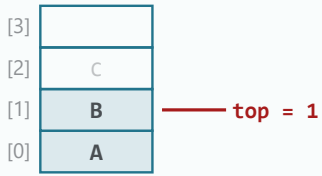
CALL STACK

`pop(&top)`

`main()`

OUTPUT

Code Trace: Array-Based Stack



stack[2] = C is returned, then top = 1. The old value stays in the cell, but it is no longer part of the stack.

Step 9 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
▶    return stack[(*top)--];
10 }
```

VARIABLES

*top = 1 returned = C

stack = [A, B]

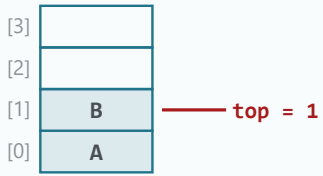
CALL STACK

pop(&top)

main()

OUTPUT

Code Trace: Array-Based Stack



1 ≥ 3? No, there is room.

Step 10 / 15

```
1  void push(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

`*top = 1` `item = D`

`stack = [A, B]`

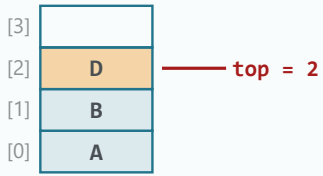
CALL STACK

`push(&top, D)`

`main()`

OUTPUT

Code Trace: Array-Based Stack



++*top → top = 2, then **stack[2] = D**.

Step 11 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
▶    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

`*top = 2` `item = D`

`stack = [A, B, D]`

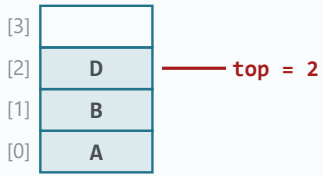
CALL STACK

`push(&top, D)`

`main()`

OUTPUT

Code Trace: Array-Based Stack



2 ≥ 3? No, there is room.

Step 12 / 15

```
1  void push(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

`*top = 2` `item = E`

`stack = [A, B, D]`

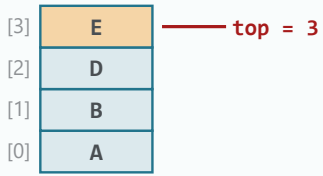
CALL STACK

`push(&top, E)`

`main()`

OUTPUT

Code Trace: Array-Based Stack



++*top → top = 3, then **stack[3] = E**.

Step 13 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
▶    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

*top = 3 | item = E

stack = [A, B, D, E]

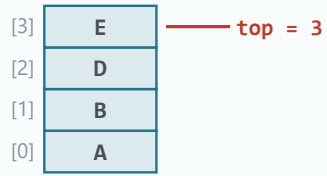
CALL STACK

push(&top, E)

main()

OUTPUT

Code Trace: Array-Based Stack



3 ≥ 3? **Yes**: no room left!

Step 14 / 15

```
1  void push(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

*top = 3 item = F

stack = [A, B, D, E]

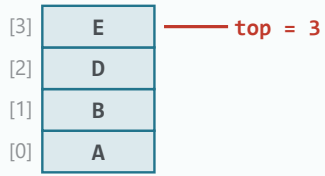
CALL STACK

push(&top, F)

main()

OUTPUT

Code Trace: Array-Based Stack



`stack_full()` prints a message and ends **the whole program** with `exit(-1)`. That is a heavy-handed reaction for library code; a version that returns `bool` leaves the decision to the caller.

Step 15 / 15

```
1  void push(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
▶      stack_full();  /* "Stack is full!!!", exit */
4    stack[++*top] = item;
5  }
6  element pop(int *top) {
7    if (*top < 0)
8      stack_empty(); /* "Stack is empty!!!", exit */
9    return stack[(*top)--];
10 }
```

VARIABLES

`*top = 3` `item = F`

`stack = [A, B, D, E]`

CALL STACK

`push(&top, F)`

`main()`

OUTPUT

```
Stack is full!!!
```

Better Design: The Stack as an Object

```c
#include <stdbool.h>
#define CAP 100

typedef struct {
    int data[CAP];
    int top;                /* -1 when empty */
} Stack;

void stack_init(Stack *s) { s->top = -1; }
bool stack_empty(const Stack *s) { return s->top < 0; }
bool stack_full(const Stack *s)  { return s->top == CAP - 1; }
int  stack_size(const Stack *s)  { return s->top + 1; }
bool stack_top(const Stack *s, int *x) {
    if (stack_empty(s)) return false;
    *x = s->data[s->top];  return true;
}
```

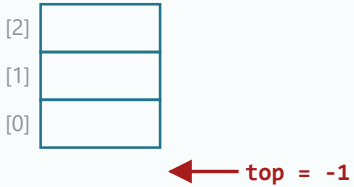
- The array and `top` live **in one struct**, so two stacks can coexist in a program without interfering.
- No globals: every routine takes the stack as a parameter.
- On error the routine returns `false` instead of calling `exit`; the **caller** decides what to do.

INSTRUCTOR "Global variables are forbidden! Inside, use a stack, a queue, local variables, whatever you need. But no globals!"

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 0



After `stack_init(&s)`, **top = −1**. Push 5, 8, 2, 9, then pop four times. CAP = 3.

Step 1 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s.top = −1` `s.data = []`

CALL STACK

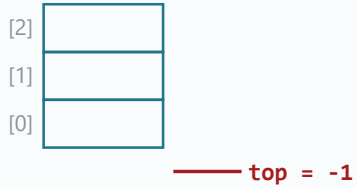
`main()`

OUTPUT

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 0



-1 == 2? No, there is room.

Step 2 / 12

```
1  bool push(Stack *s, int x) {
▶    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = -1` `x = 5` `s→data = []`

CALL STACK

`push(&y, 5)`

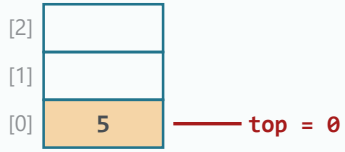
`main()`

OUTPUT

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 1



++s->top → 0; **data[0] = 5**; returns true.

Step 3 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
▶    s->data[++s->top] = x;
▶    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 0` `x = 5` `s→data = [5]`

CALL STACK

`push(&y, 5)`

`main()`

OUTPUT

```
push 5:1
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 1



0 == 2? No, there is room.

Step 4 / 12

```
1  bool push(Stack *s, int x) {
▶    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 0` `x = 8` `s→data = [5]`

CALL STACK

`push(&y, 8)`

`main()`

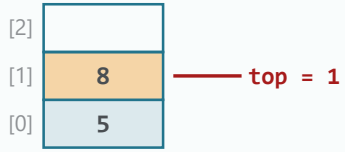
OUTPUT

```
push 5:1
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 2



++s->top → 1; **data[1] = 8**; returns true.

Step 5 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
▶    s->data[++s->top] = x;
▶    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 1` `x = 8` `s→data = [5, 8]`

CALL STACK

`push(&y, 8)`

`main()`

OUTPUT

```
push 5:1 8:1
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 2



1 == 2? No, there is room.

Step 6 / 12

```
1  bool push(Stack *s, int x) {
▶    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 1` `x = 2`

`s→data = [5, 8]`

CALL STACK

`push(&y, 2)`

`main()`

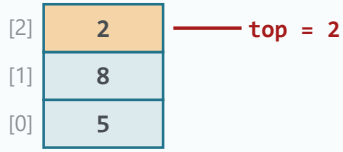
OUTPUT

```
push 5:1 8:1
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 3



++s->top → 2; **data[2] = 2**; returns true.

Step 7 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
▶    s->data[++s->top] = x;
▶    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 2` `x = 2`

`s→data = [5, 8, 2]`

CALL STACK

`push(&y, 2)`

`main()`

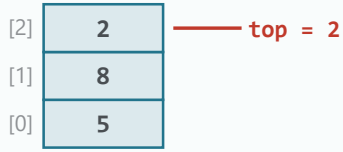
OUTPUT

```
push 5:1 8:1 2:1
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 3



top == CAP − 1 (2 == 2) → **return false**. The program keeps running; the caller learns about the overflow.

Step 8 / 12

```
1  bool push(Stack *s, int x) {
▶    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
7    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 2` `x = 9`

`s→data = [5, 8, 2]`

CALL STACK

`push(&y, 9)`

`main()`

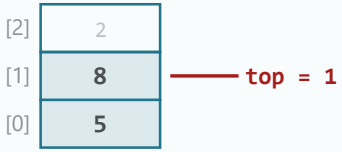
OUTPUT

```
push 5:1 8:1 2:1 9:0
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 2



Not empty; ***x = data[2] = 2**, then top = 1.

Step 9 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
▶    if (s->top < 0) return false;   /* underflow */
▶    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 1` `*x = 2`

`s→data = [5, 8]`

CALL STACK

`pop(&y, &x)`

`main()`

OUTPUT

```
push 5:1 8:1 2:1 9:0
pop 2
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 1



Not empty; ***x = data[1] = 8**, then top = 0.

Step 10 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
▶    if (s->top < 0) return false;   /* underflow */
▶    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = 0` `*x = 8` `s→data = [5]`

CALL STACK

`pop(&y, &x)`

`main()`

OUTPUT

```
push 5:1 8:1 2:1 9:0
pop 2 8
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 0



Not empty; ***x = data[0] = 5**, then top = -1.

Step 11 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
▶    if (s->top < 0) return false;   /* underflow */
▶    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = -1` `*x = 5` `s→data = []`

CALL STACK

`pop(&y, &x)`

`main()`

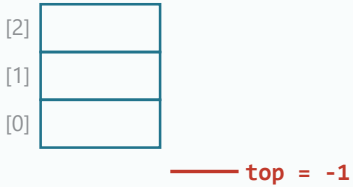
OUTPUT

```
push 5:1 8:1 2:1 9:0
pop 2 8 5
```

Code Trace: Push / Pop Returning `bool`

CAP = 3

size = top + 1 = 0



top < 0 → return false: underflow, and `*x` is left untouched.

Step 12 / 12

```
1  bool push(Stack *s, int x) {
2    if (s->top == CAP - 1) return false; /* overflow */
3    s->data[++s->top] = x;
4    return true;
5  }
6  bool pop(Stack *s, int *x) {
▶    if (s->top < 0) return false;   /* underflow */
8    *x = s->data[s->top--];
9    return true;
10 }
```

VARIABLES

`s→top = -1` `s→data = []`

CALL STACK

`pop(&y, &x)`

`main()`

OUTPUT

```
push 5:1 8:1 2:1 9:0
pop 2 8 5 false
```

Write It as a Component: `stack.h` and `stack.c`

```c
/* stack.h: all the client sees */
#ifndef STACK_H
#define STACK_H
#include <stdbool.h>
typedef struct Stack Stack;  /* opaque */
Stack *stack_create(int capacity);
void stack_destroy(Stack *s);
bool push(Stack *s, int x);
bool pop(Stack *s, int *x);
bool stack_empty(const Stack *s);
#endif
```

Client code cannot see inside the `struct`; it only calls the operations. Even if the implementation switches from an array to a linked list, client code stays the same.

```c
/* stack.c: implementation */
struct Stack { int *data; int top, cap; };

Stack *stack_create(int cap) {
  Stack *s = malloc(sizeof *s);
  if (s == NULL) return NULL;
  s->data = malloc(cap * sizeof *s->data);
  if (s->data == NULL) {
    free(s);  return NULL;
  }
  s->top = -1;  s->cap = cap;
  return s;
}
void stack_destroy(Stack *s) {
  if (s) { free(s->data); free(s); }
}
```

Code Trace: Reversing a String with a Stack



Input **"VISIT"**; an empty stack is set up.

Step 1 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
▶   stack_init(&st);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&st, s[i]);    /* assume CAP suffices */
6   for (i = 0; pop(&st, &c); i++)
7     s[i] = (char)c;
8 }
```

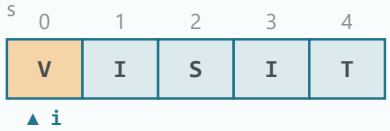
VARIABLES

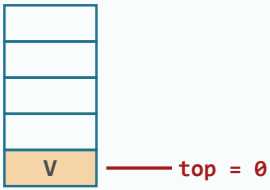
`s = "VISIT"` `i = ?` `st.top = -1`

CALL STACK

- **reverse(s)**
- main()

Code Trace: Reversing a String with a Stack





s[0] = 'V' ≠ '\0' → **push('V')**.

Step 2 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&st, s[i]);    /* assume CAP suffices */
6   for (i = 0; pop(&st, &c); i++)
7     s[i] = (char)c;
8 }
```

VARIABLES

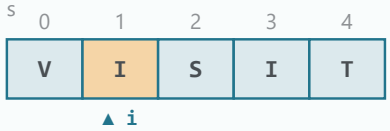
`s = "VISIT"` `i = 0` `st.top = 0`

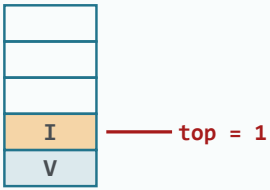
CALL STACK

`reverse(s)`

`main()`

Code Trace: Reversing a String with a Stack





s[1] = 'I' ≠ '\0' → **push('I')**.

Step 3 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&st, s[i]);    /* assume CAP suffices */
6   for (i = 0; pop(&st, &c); i++)
7     s[i] = (char)c;
8 }
```

VARIABLES

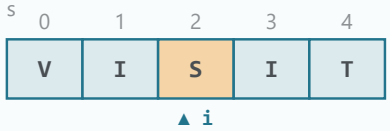
`s = "VISIT"` `i = 1` `st.top = 1`

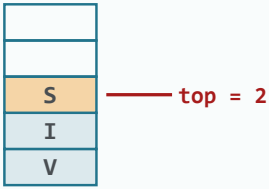
CALL STACK

`reverse(s)`

`main()`

Code Trace: Reversing a String with a Stack





s[2] = 'S' ≠ '\0' → **push('S')**.

Step 4 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&st, s[i]);    /* assume CAP suffices */
6   for (i = 0; pop(&st, &c); i++)
7     s[i] = (char)c;
8 }
```

VARIABLES

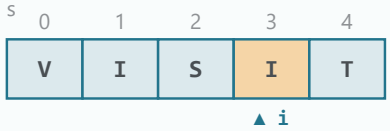
s = "VISIT" | i = 2 | st.top = 2

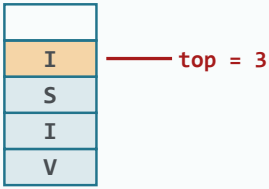
CALL STACK

reverse(s)

main()

Code Trace: Reversing a String with a Stack





s[3] = 'I' ≠ '\0' → **push('I')**.

Step 5 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&st, s[i]);    /* assume CAP suffices */
6   for (i = 0; pop(&st, &c); i++)
7     s[i] = (char)c;
8 }
```

VARIABLES

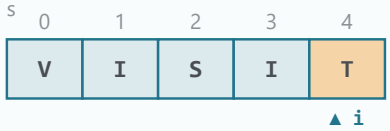
`s = "VISIT"` `i = 3` `st.top = 3`

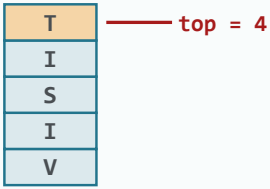
CALL STACK

`reverse(s)`

`main()`

Code Trace: Reversing a String with a Stack





s[4] = 'T' ≠ '\0' → **push('T')**.

Step 6 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&st, s[i]);    /* assume CAP suffices */
6   for (i = 0; pop(&st, &c); i++)
7     s[i] = (char)c;
8 }
```

VARIABLES

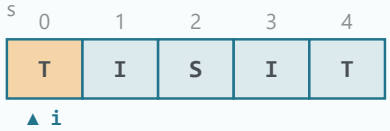
`s = "VISIT"` `i = 4` `st.top = 4`

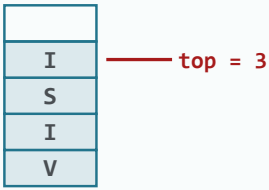
CALL STACK

`reverse(s)`

`main()`

Code Trace: Reversing a String with a Stack





pop → **'T'**, **s[0] = 'T'**.

Step 7 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&st, s[i]);    /* assume CAP suffices */
▶   for (i = 0; pop(&st, &c); i++)
▶     s[i] = (char)c;
8 }
```

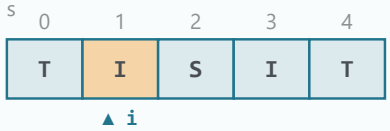
VARIABLES

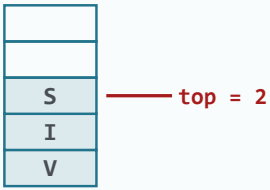
`s = "TISIT"` `i = 0` `c = 'T'` `st.top = 3`

CALL STACK

- `reverse(s)`
- `main()`

Code Trace: Reversing a String with a Stack





pop → **'I'**, **s[1] = 'I'**.

Step 8 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&st, s[i]);    /* assume CAP suffices */
▶   for (i = 0; pop(&st, &c); i++)
▶     s[i] = (char)c;
8 }
```

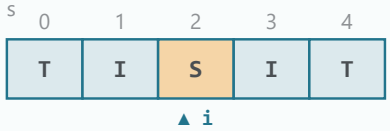
VARIABLES

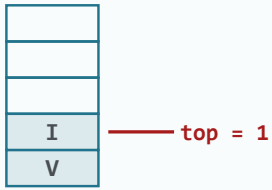
`s = "TISIT"` `i = 1` `c = 'I'` `st.top = 2`

CALL STACK

- `reverse(s)`
- `main()`

Code Trace: Reversing a String with a Stack





pop → **'S'**, **s[2] = 'S'**.

Step 9 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&st, s[i]);    /* assume CAP suffices */
▶   for (i = 0; pop(&st, &c); i++)
▶     s[i] = (char)c;
8 }
```

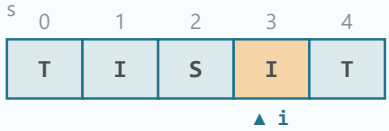
VARIABLES

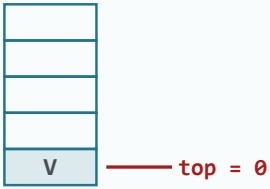
`s = "TISIT"` `i = 2` `c = 'S'` `st.top = 1`

CALL STACK

- `reverse(s)`
- `main()`

Code Trace: Reversing a String with a Stack





pop → **'I'**, **s[3] = 'I'**.

Step 10 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&st, s[i]);    /* assume CAP suffices */
▶   for (i = 0; pop(&st, &c); i++)
▶     s[i] = (char)c;
8 }
```

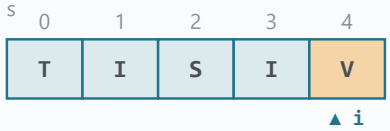
VARIABLES

`s = "TISIT"` `i = 3` `c = 'I'` `st.top = 0`

CALL STACK

- **reverse(s)**
- main()

Code Trace: Reversing a String with a Stack





pop → **'V'**, **s[4] = 'V'**.

Step 11 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&st, s[i]);    /* assume CAP suffices */
▶   for (i = 0; pop(&st, &c); i++)
▶     s[i] = (char)c;
8 }
```

VARIABLES

`s = "TISIV"` `i = 4` `c = 'V'` `st.top = -1`

CALL STACK

- **reverse(s)**
- main()

Code Trace: Reversing a String with a Stack



The stack is empty: `pop` returns false and the loop ends. Result: **"TISIV"**. n pushes + n pops → **O(n)**.

Step 12 / 12

```
1 void reverse(char *s) {
2   Stack st;  int c, i;
3   stack_init(&st);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&st, s[i]);    /* assume CAP suffices */
▶   for (i = 0; pop(&st, &c); i++)
7     s[i] = (char)c;
8 }
```

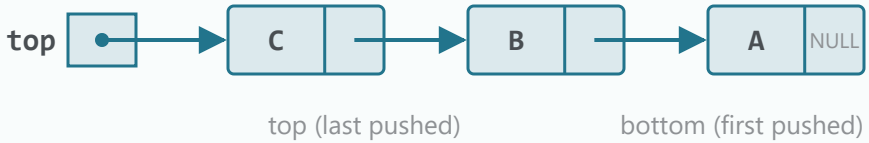
VARIABLES

`s = "TISIV"` `i = 5` `st.top = -1`

CALL STACK

`reverse(s)`
`main()`

Implementing a Stack with a Linked List



A, B, C pushed in that order: `top` points to the head of the list.

- Each item is a **node**: `data` plus a `link` to the next node.
- Insert and delete **at the head**: both O(1), no shifting.
- No capacity limit: overflow happens only if `malloc` fails.

Why the head and not the tail? Popping at the tail means stepping back from the last node, and finding the previous node in a singly linked list costs O(n). Hence **top = head of the list**.

INSTRUCTOR "Top holds the head of the list. Deleting means moving back from the last item, so either you keep the tail with two pointers or you keep the head with one."

Code Trace: Linked `push`



Start: `node *top = NULL;` (empty stack). Push 10, 20, 30 in turn.

Step 1 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

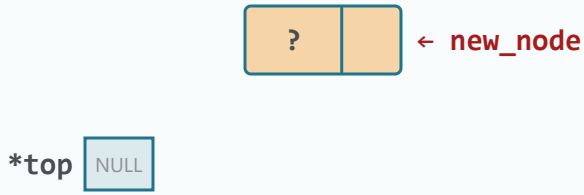
VARIABLES

`top(main) = NULL`

CALL STACK

`main()`

Code Trace: Linked `push`



malloc allocates a new node; if it is not NULL we continue. Its fields are still undefined.

Step 2 / 14

```
1 bool push(node **top, int x) {
▶   node *new_node = malloc(sizeof *new_node);
▶   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

*top = NULL x = 10

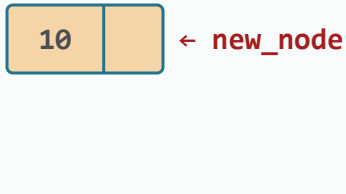
new_node = node(?)

CALL STACK

push(&top, 10)

main()

Code Trace: Linked `push`



new_node->data = 10.

Step 3 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
▶   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

`*top = NULL` `x = 10`

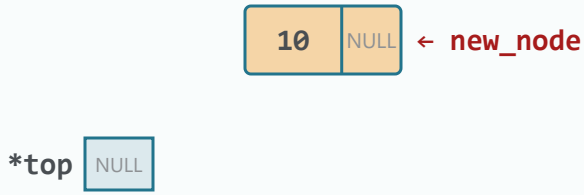
`new_node = node(10)`

CALL STACK

`push(&top, 10)`

`main()`

Code Trace: Linked `push`



new_node->link = *top: the new node points to the old top (NULL). Order matters: link first, then change `*top`.

Step 4 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
▶   new_node->link = *top;   /* point to old top */
6   *top = new_node;         /* new top */
7   return true;
8 }
```

VARIABLES

`*top = NULL` `x = 10`

`new_node = node(10)`

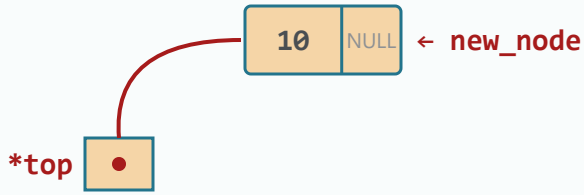
`new_node→link = NULL`

CALL STACK

`push(&top, 10)`

`main()`

Code Trace: Linked `push`



***top = new_node**: the caller's `top` now points to 10. Stack: 10.

Step 5 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
▶   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

*top = node(10) x = 10

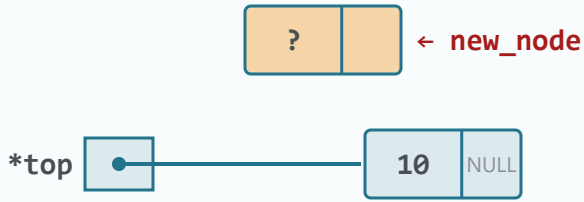
new_node = node(10)

CALL STACK

push(&top, 10)

main()

Code Trace: Linked `push`



malloc allocates a new node; if it is not NULL we continue. Its fields are still undefined.

Step 6 / 14

```
1 bool push(node **top, int x) {
▶   node *new_node = malloc(sizeof *new_node);
▶   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

*top = node(10) x = 20

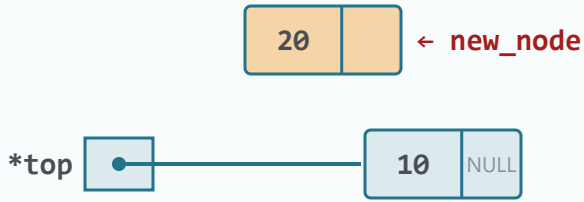
new_node = node(?)

CALL STACK

push(&top, 20)

main()

Code Trace: Linked `push`



new_node->data = 20.

Step 7 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
▶   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

*top = node(10) x = 20

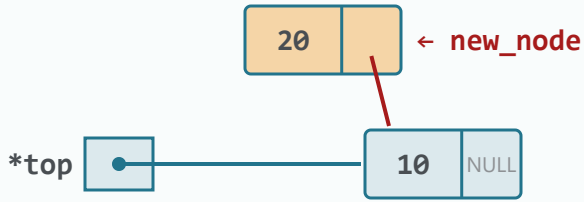
new_node = node(20)

CALL STACK

push(&top, 20)

main()

Code Trace: Linked `push`



new_node->link = *top: the new node points to the old top (node(10)). Order matters: link first, then change `*top`.

Step 8 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
▶   new_node->link = *top;   /* point to old top */
6   *top = new_node;         /* new top */
7   return true;
8 }
```

VARIABLES

`*top = node(10)` `x = 20`

`new_node = node(20)`

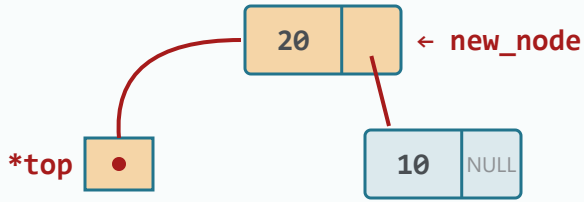
`new_node→link = node(10)`

CALL STACK

`push(&top, 20)`

`main()`

Code Trace: Linked `push`



***top = new_node**: the caller's `top` now points to 20. Stack: 20 → 10.

Step 9 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
▶   *top = new_node;        /* new top */
7   return true;
8 }
```

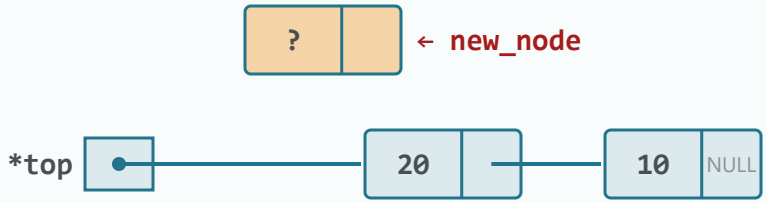
VARIABLES

`*top = node(20)` `x = 20` `new_node = node(20)`

CALL STACK

`push(&top, 20)`
`main()`

Code Trace: Linked `push`



malloc allocates a new node; if it is not NULL we continue. Its fields are still undefined.

Step 10 / 14

```
1 bool push(node **top, int x) {
▶   node *new_node = malloc(sizeof *new_node);
▶   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

`*top = node(20)` `x = 30`

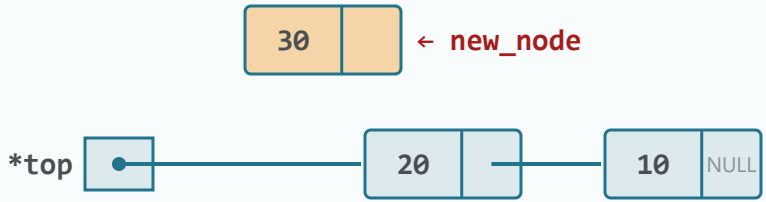
`new_node = node(?)`

CALL STACK

`push(&top, 30)`

`main()`

Code Trace: Linked `push`



new_node->data = 30.

Step 11 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
▶   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

*top = node(20) x = 30

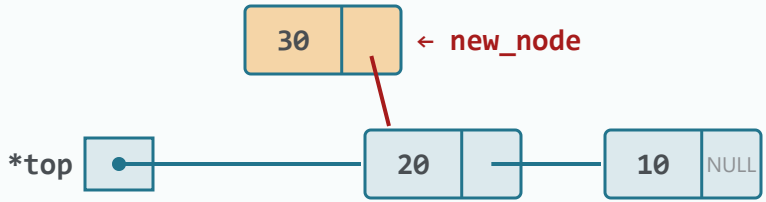
new_node = node(30)

CALL STACK

push(&top, 30)

main()

Code Trace: Linked `push`



new_node->link = *top: the new node points to the old top (node(20)). Order matters: link first, then change `*top`.

Step 12 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
▶   new_node->link = *top;   /* point to old top */
6   *top = new_node;         /* new top */
7   return true;
8 }
```

VARIABLES

`*top = node(20)` `x = 30`

`new_node = node(30)`

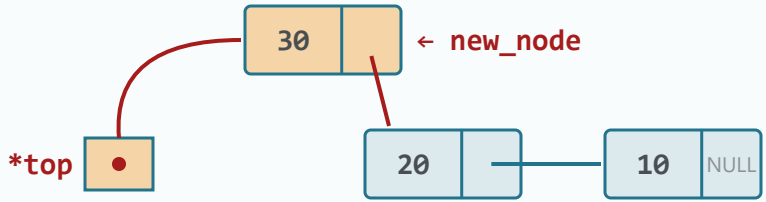
`new_node→link = node(20)`

CALL STACK

`push(&top, 30)`

`main()`

Code Trace: Linked `push`



***top = new_node**: the caller's `top` now points to 30. Stack: 30 → 20 → 10.

Step 13 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
▶   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

`*top = node(30)` `x = 30` `new_node = node(30)`

CALL STACK

`push(&top, 30)`
`main()`

Code Trace: Linked `push`



After three pushes: 30 → 20 → 10 → NULL. Each push does a constant amount of work: **O(1)**.

Step 14 / 14

```
1 bool push(node **top, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false; /* no memory */
4   new_node->data = x;
5   new_node->link = *top;  /* point to old top */
6   *top = new_node;        /* new top */
7   return true;
8 }
```

VARIABLES

top(main) = node(30)

CALL STACK

main()

Code Trace: Linked `pop`



Stack: 30 → 20 → 10. We call `pop` four times; `main` prints each successful result.

Step 1 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;        /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

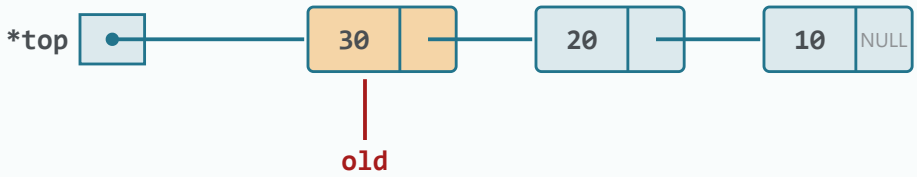
`top(main) = node(30)`

CALL STACK

`main()`

OUTPUT

Code Trace: Linked `pop`



old = *top: just a copy of a pointer; **no** new memory is allocated.

Step 2 / 14

```
1 bool pop(node **top, int *x) {
▶   node *old = *top;        /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(30)` `old = node(30)`

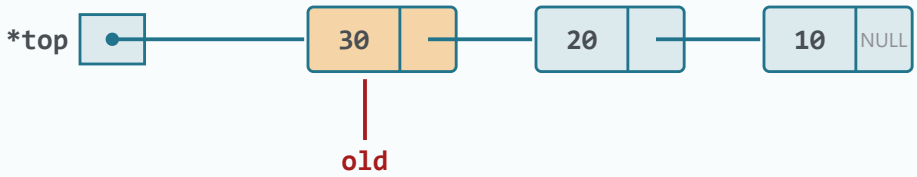
CALL STACK

`pop(&top, &x)`

`main()`

OUTPUT

Code Trace: Linked `pop`



***x = 30** (read the value before the node is freed).

Step 3 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;        /* no malloc! */
3   if (old == NULL) return false;  /* empty */
▶   *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(30)` `old = node(30)` `*x = 30`

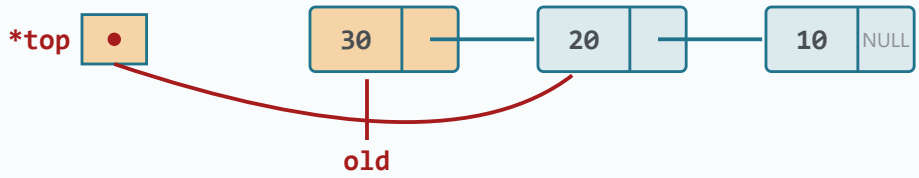
CALL STACK

`pop(&top, &x)`

`main()`

OUTPUT

Code Trace: Linked `pop`



***top = old->link** → node(20).

Step 4 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;         /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
▶   *top = old->link;         /* advance top */
6   free(old);                /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(20)` `old = node(30)` `*x = 30`

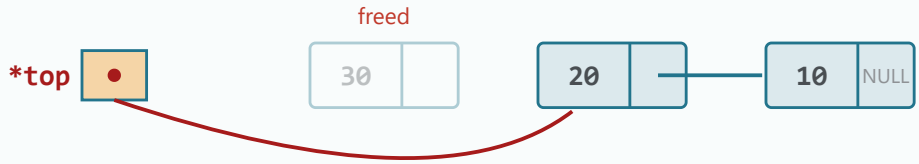
CALL STACK

`pop(&top, &x)`

`main()`

OUTPUT

Code Trace: Linked `pop`



free(old): node 30's memory goes back to the system. `main` prints 30.

Step 5 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;         /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;         /* advance top */
▶   free(old);                /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(20)` `old = (freed)` `*x = 30`

CALL STACK

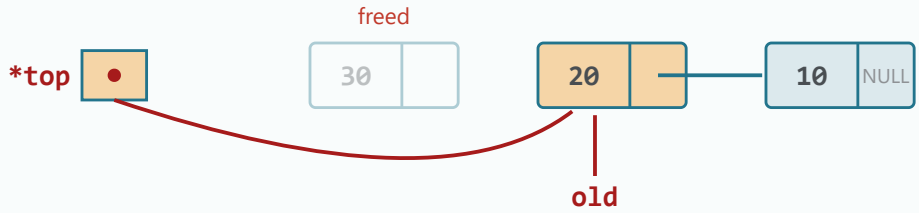
`pop(&top, &x)`

`main()`

OUTPUT

```
30
```

Code Trace: Linked `pop`



old = *top: just a copy of a pointer; **no** new memory is allocated.

Step 6 / 14

```
1 bool pop(node **top, int *x) {
▶   node *old = *top;        /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(20)` `old = node(20)`

CALL STACK

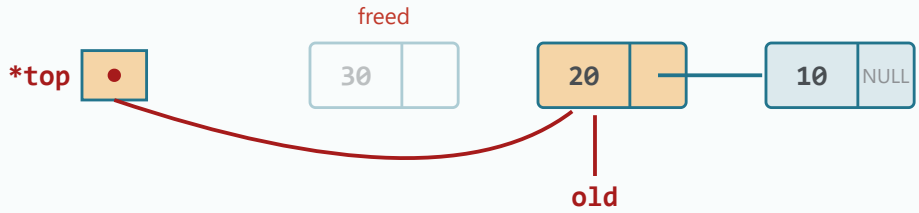
`pop(&top, &x)`

`main()`

OUTPUT

```
30
```

Code Trace: Linked `pop`



***x = 20** (read the value before the node is freed).

Step 7 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;        /* no malloc! */
3   if (old == NULL) return false;  /* empty */
▶   *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(20)` `old = node(20)` `*x = 20`

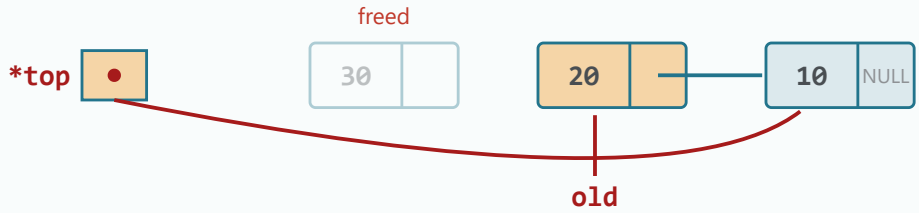
CALL STACK

`pop(&top, &x)`
`main()`

OUTPUT

```
30
```

Code Trace: Linked `pop`



***top = old->link** → node(10).

Step 8 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;         /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
▶   *top = old->link;         /* advance top */
6   free(old);                /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(10)` `old = node(20)` `*x = 20`

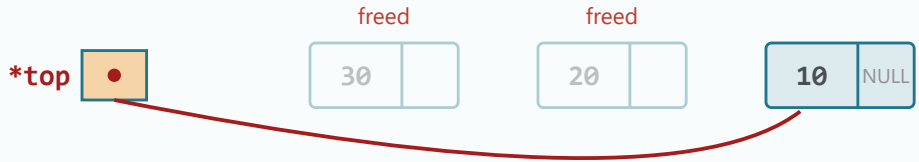
CALL STACK

`pop(&top, &x)`
`main()`

OUTPUT

```
30
```

Code Trace: Linked `pop`



free(old): node 20's memory goes back to the system. `main` prints 20.

Step 9 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;         /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;         /* advance top */
▶   free(old);                /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(10)` `old = (freed)` `*x = 20`

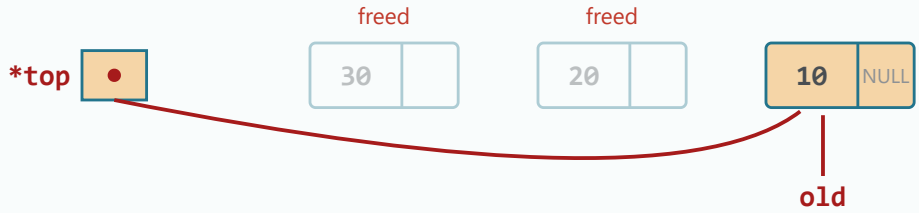
CALL STACK

`pop(&top, &x)`
`main()`

OUTPUT

```
30 20
```

Code Trace: Linked `pop`



old = *top: just a copy of a pointer; **no** new memory is allocated.

Step 10 / 14

```
1 bool pop(node **top, int *x) {
▶   node *old = *top;        /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = node(10)` `old = node(10)`

CALL STACK

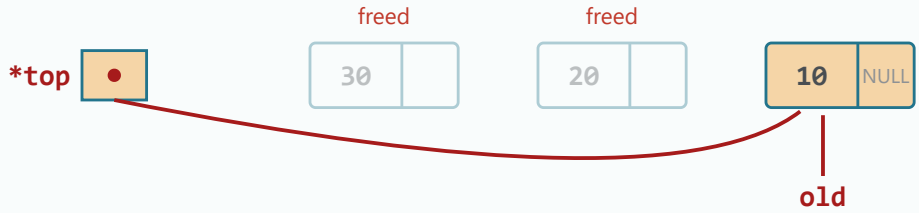
`pop(&top, &x)`

`main()`

OUTPUT

```
30 20
```

Code Trace: Linked `pop`



***x = 10** (read the value before the node is freed).

Step 11 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;        /* no malloc! */
3   if (old == NULL) return false;  /* empty */
▶  *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

`*top` = **node(10)** `old` = **node(10)** `*x` = **10**

CALL STACK

pop(&top, &x)
main()

OUTPUT

```
30 20
```

Code Trace: Linked `pop`



***top = old->link** → NULL.

Step 12 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;         /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
▶   *top = old->link;         /* advance top */
6   free(old);                /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = NULL` `old = node(10)` `*x = 10`

CALL STACK

`pop(&top, &x)`
`main()`

OUTPUT

```
30 20
```

Code Trace: Linked `pop`



free(old): node 10's memory goes back to the system. `main` prints 10.

Step 13 / 14

```
1 bool pop(node **top, int *x) {
2   node *old = *top;         /* no malloc! */
3   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;         /* advance top */
▶   free(old);                /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = NULL` `old = (freed)` `*x = 10`

CALL STACK

`pop(&top, &x)`
`main()`

OUTPUT

```
30 20 10
```

Code Trace: Linked `pop`



old = NULL → the stack is empty, **return false**. The loop stops here.

Step 14 / 14

```
1 bool pop(node **top, int *x) {
▶   node *old = *top;        /* no malloc! */
▶   if (old == NULL) return false;  /* empty */
4   *x = old->data;
5   *top = old->link;        /* advance top */
6   free(old);               /* release memory */
7   return true;
8 }
```

VARIABLES

`*top = NULL` `old = NULL`

CALL STACK

`pop(&top, &x)`

`main()`

OUTPUT

```
30 20 10
```

Who Owns the Memory in a Linked Stack

```c
typedef struct node {
    int data;
    struct node *link;
} node;

void stack_clear(node **top) {
    while (*top != NULL) {
        node *old = *top;
        *top = old->link;
        free(old);
    }
}                   /* afterwards *top == NULL */
```

- `old` is a **pointer variable**, not a new node: it needs no `malloc`.
- Save the link first (`*top = old→link`), **then** call `free` : the other way round reads freed memory.
- Call `stack_clear` before the program ends: every `malloc` needs a matching `free` .

INSTRUCTOR "You don't need to malloc space for every pointer. There's no such rule!"

INSTRUCTOR "If you don't learn pointers now, you may still not know them in your senior year."

Array-Based or Linked?

Criterion	Fixed array	Growable array	Linked list
Capacity	Fixed at compile time	×2 as needed	Until memory runs out
push / pop	O(1)	O(1) amortized	O(1), a malloc per push
Extra memory per item	None	None (except unused capacity)	One pointer (8 bytes)
Cache behavior	Very good	Very good	Poor (nodes scattered)
Overflow	Possible	Only if realloc fails	Only if malloc fails
Bug risk	Bounds checks	realloc's return value	forgotten free, dangling pointer

If the maximum size is known, use a fixed array; if not, a growable array is usually fastest in practice. A linked stack suits cases where the nodes already live in other structures or are created and destroyed one at a time.

The Call Stack Overflows Too

The call stack is a stack: each call pushes a frame holding the return address, parameters, and locals, and `return` pops it. The most recently called function finishes first.

Infinite recursion

Recursion with no base case, or one it never reaches, pushes a frame per call; when the system stack (typically a few MB) fills up, the program crashes with a **stack overflow**.

Large local arrays

A local array such as `int a[1000000];` lives in the frame and can overflow the stack on its own. Put large data on the heap with `malloc`.

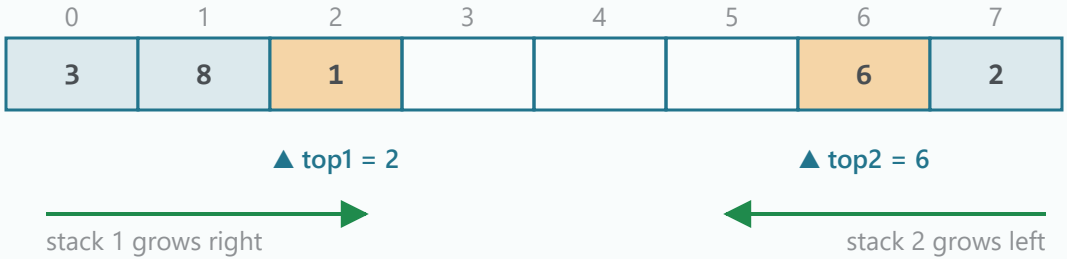
Returning a local's address

`return &local;` is a bug: once the frame is popped, other calls reuse that memory (a dangling pointer).

Removing recursion

Any recursive algorithm can be turned into a loop that maintains its own stack.

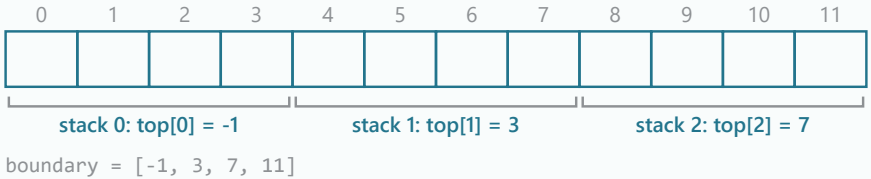
Multiple Stacks: Start with Two



- Stack 1 grows **up from 0**, stack 2 grows **down from m−1**.
- Empty: `top1 = -1` , `top2 = m` .
- Full: `top1 + 1 == top2` , when the two meet.

Advantage: even if one stack holds few items and the other many, **no cell is wasted**; the array fills only when the total reaches m. With three or more stacks, this trick no longer works.

Code Trace: n Stacks, One Memory



MEM = 12, n = 3: 4 cells each; **top[i] = boundary[i] = (m/n)·i − 1**, **boundary[n] = m − 1**.

Step 1 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
3    if (top[i] == boundary[i+1]) return false; /* full */
4    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`top = [-1, 3, 7]`

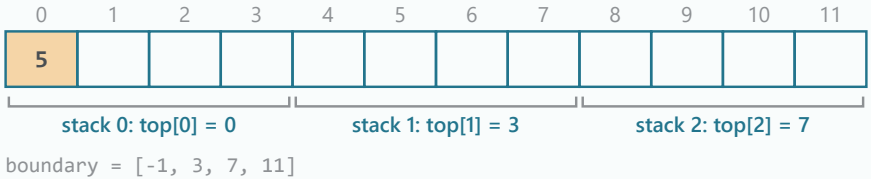
`boundary = [-1, 3, 7, 11]`

CALL STACK

`main()`

OUTPUT

Code Trace: n Stacks, One Memory



Room; **memory[0] = 5**, top[0] = 0.

Step 2 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
▶    if (top[i] == boundary[i+1]) return false; /* full */
▶    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 0` `x = 5` `top = [0, 3, 7]`

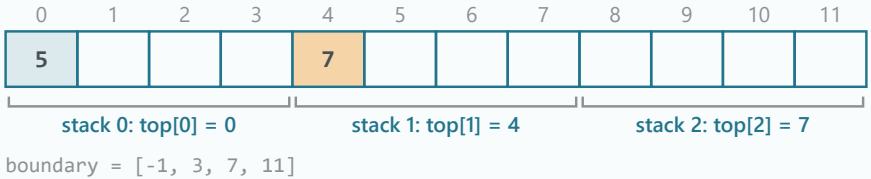
CALL STACK

`push(0, 5)`

`main()`

OUTPUT

Code Trace: n Stacks, One Memory



Room; **memory[4] = 7**, top[1] = 4.

Step 3 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
▶    if (top[i] == boundary[i+1]) return false; /* full */
▶    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 1` `x = 7` `top = [0, 4, 7]`

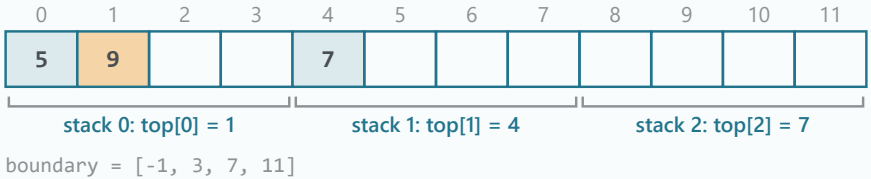
CALL STACK

`push(1, 7)`

`main()`

OUTPUT

Code Trace: n Stacks, One Memory



Room; **memory[1] = 9**, top[0] = 1.

Step 4 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
▶    if (top[i] == boundary[i+1]) return false; /* full */
▶    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 0` `x = 9` `top = [1, 4, 7]`

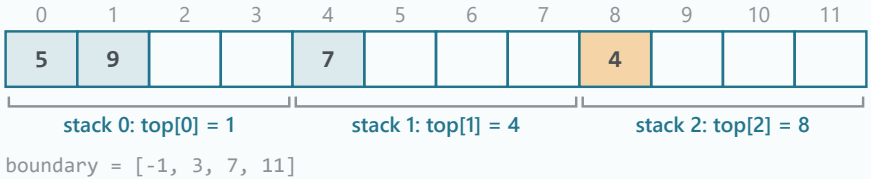
CALL STACK

`push(0, 9)`

`main()`

OUTPUT

Code Trace: n Stacks, One Memory



Room; **memory[8] = 4**, top[2] = 8.

Step 5 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
▶    if (top[i] == boundary[i+1]) return false; /* full */
▶    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 2` `x = 4` `top = [1, 4, 8]`

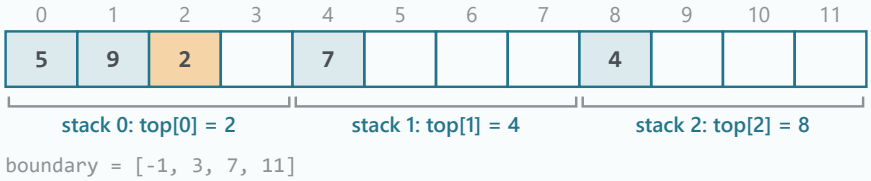
CALL STACK

`push(2, 4)`

`main()`

OUTPUT

Code Trace: n Stacks, One Memory



Room; **memory[2] = 2**, top[0] = 2.

Step 6 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
▶    if (top[i] == boundary[i+1]) return false; /* full */
▶    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 0` `x = 2` `top = [2, 4, 8]`

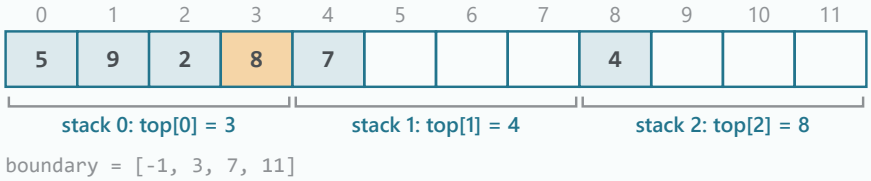
CALL STACK

`push(0, 2)`

`main()`

OUTPUT

Code Trace: n Stacks, One Memory



Room; **memory[3] = 8**, top[0] = 3.

Step 7 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
▶    if (top[i] == boundary[i+1]) return false; /* full */
▶    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 0` `x = 8` `top = [3, 4, 8]`

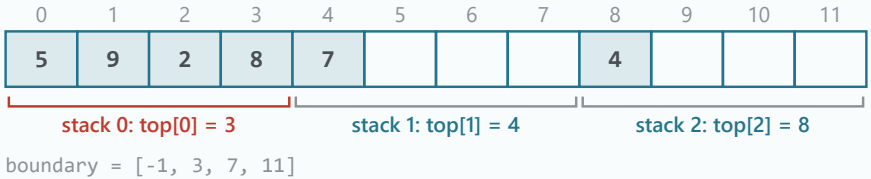
CALL STACK

`push(0, 8)`

`main()`

OUTPUT

Code Trace: n Stacks, One Memory



top[0] == boundary[1] (3 == 3) → **stack 0 is full**, yet 6 cells are free! Horowitz's `stack_full` would shift stacks.

Step 8 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
▶    if (top[i] == boundary[i+1]) return false; /* full */
4    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
8    if (top[i] == boundary[i]) return false; /* empty */
9    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 0` `x = 6` `top = [3, 4, 8]`

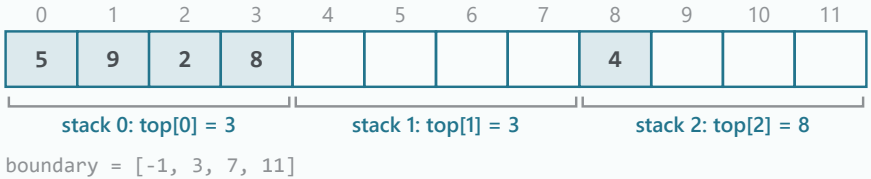
CALL STACK

`push(0, 6)`
`main()`

OUTPUT

```
push(0,6): full
```

Code Trace: n Stacks, One Memory



top[1] ≠ boundary[1] → not empty; ***x = 7**, top[1] = 3.

Step 9 / 9

```
1  /* stack i: boundary[i]+1 .. boundary[i+1] */
2  bool push(int i, int x) {
3    if (top[i] == boundary[i+1]) return false; /* full */
4    memory[++top[i]] = x;
5    return true;
6  }
7  bool pop(int i, int *x) {
▶    if (top[i] == boundary[i]) return false; /* empty */
▶    *x = memory[top[i]--];
10   return true;
11 }
```

VARIABLES

`i = 1` `*x = 7` `top = [3, 3, 8]`

CALL STACK

`pop(1, &x)`

`main()`

OUTPUT

```
push(0,6): full
pop(1): 7
```

Sharing Memory Among n Stacks: Pros and Cons

A one-dimensional array (`memory[MEMORY_SIZE]`) can be split into n segments to hold n stacks; each stack's `top` and `boundary` values are kept in two auxiliary arrays.

Pros

- \+ One `malloc` , one array: easy to manage.
- \+ push/pop are still O(1).
- \+ The same idea works for queues (n queues).

Cons

- – When one segment fills, you get a “full” error even if memory has room elsewhere.
- – Shifting stacks to make room costs O(m): shifting is expensive.
- – Segment sizes are hard to predict in advance.

Horowitz–Sahni's fix: `stack_full(i)` looks for free space first to the right, then to the left, and shifts the stacks in between by one cell. In the worst case every push can cost O(m).

LAB · C CODE STEP BY STEP

Lab: Linked Stack

Three pushes, one pop

Input (number = push, pop) `5, 3, pop, 8` Apply

CODE

```
 1 bool push(node **top, int x) {
 2   node *new_node = malloc(sizeof *new_node);
 3   if (new_node == NULL) return false;
 4   new_node->data = x;
 5   new_node->link = *top;  /* old top */
 6   *top = new_node;        /* new top */
 7   return true;
 8 }
 9 bool pop(node **top, int *x) {
10   node *old = *top;     /* no malloc! */
11   if (old == NULL) return false;  /* empty */
12   *x = old->data;
13   *top = old->link;     /* advance top */
14   free(old);            /* release memory */
```

empty stack: top = NULL

`top` NULL

OUTPUT (none yet)

The stack starts empty: `top = NULL`. Each `push` allocates a new node and links it on top; `pop` unlinks the top node and frees its memory.

SUMMARY

Which Output Orders Are Possible?

We push 1, 2, 3, 4 in this order and may pop any number of times in between. Which of the output orders below can we get?

Output order	Possible?
`1 2 3 4`	▶ show
`4 3 2 1`	▶ show
`2 4 3 1`	▶ show
`3 1 4 2`	▶ show

General rule: after some x is output, the items pushed before x and not yet popped must come out **in reverse order**. The number of possible orders for n items is a Catalan number (14 for n = 4).

SUMMARY

Complexity Summary

Operation	Fixed array	Growable array	Linked
push	O(1)	O(1) amortized, O(n) worst case	O(1)
pop	O(1)	O(1)	O(1)
peek / empty / full	O(1)	O(1)	O(1)
Memory for n items	MAX (fixed)	≤ 2n	n · (item + pointer)
Cleanup	O(1)	O(1) (one free)	O(n)
fact(n) call stack	O(n) frames (whatever the implementation)		
Multiple-stack push	O(1); making room O(m)	—	—

SUMMARY

Common Mistakes

1. Popping an empty stack or pushing onto a full one: skipping the bounds check.
2. Writing `*top--`: it decrements the pointer; the correct form is `(*top)--`.
3. Confusing the value of `top` with the item count (count = top + 1).
4. In a linked push, writing `*top = new_node` first and losing the old list.
5. In pop, reading the value after `free`, or never freeing the node.
6. Calling `malloc` for a temporary pointer that doesn't need it, and leaking memory.

SUMMARY

Exercises

1. Make the array stack's `push` return `bool` instead of calling `exit`.
2. Write `push(k, x)` and `pop(k)` for two stacks in one array (k = 1, 2).
3. Wrap the linked stack in a `struct` holding `top` and a `size` field.
4. Write `void reverse(char *s)`: reverse a string with a stack.
5. With only push/pop, return a stack's bottom item (a helper stack is allowed).
6. At most how many frames are on the stack during `fact(5)`? Draw them with `main`.

▶ **Hints / answers**

SUMMARY

Chapter Summary

LIFO

Insert and delete only at the top; last in, first out.

Array + top

top = −1 empty, top = MAX − 1 full; `++*top` and `(*top)--`.

Error handling

Return a status instead of calling exit; the caller decides about overflow/underflow.

Linked stack

top = head of list; push/pop O(1); free every node you pop.

Call stack

Each call is a frame: return address, locals, previous frame.

Multiple stacks

One memory, top[i] and boundary[i]; a full segment forces shifting.

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 3.1–3.2, 3.7, and 4.3.
- T. H. Cormen et al., *Introduction to Algorithms*, 3rd ed., Section 10.1 (stacks) and Chapter 17 (amortized analysis).
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 4.

← Chapter 5: Polynomials and Sparse Matrices

Course page

Chapter 7: Queues

The End

Chapter 6: Stacks

Next chapter: Chapter 7: Queues · Course page