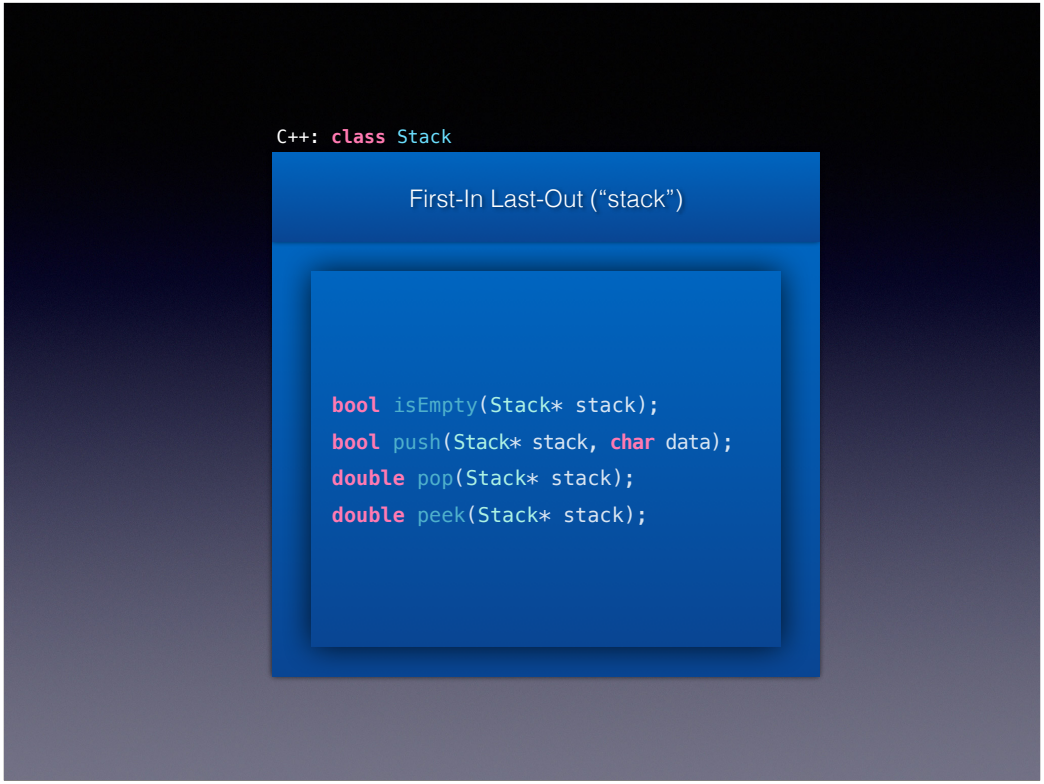
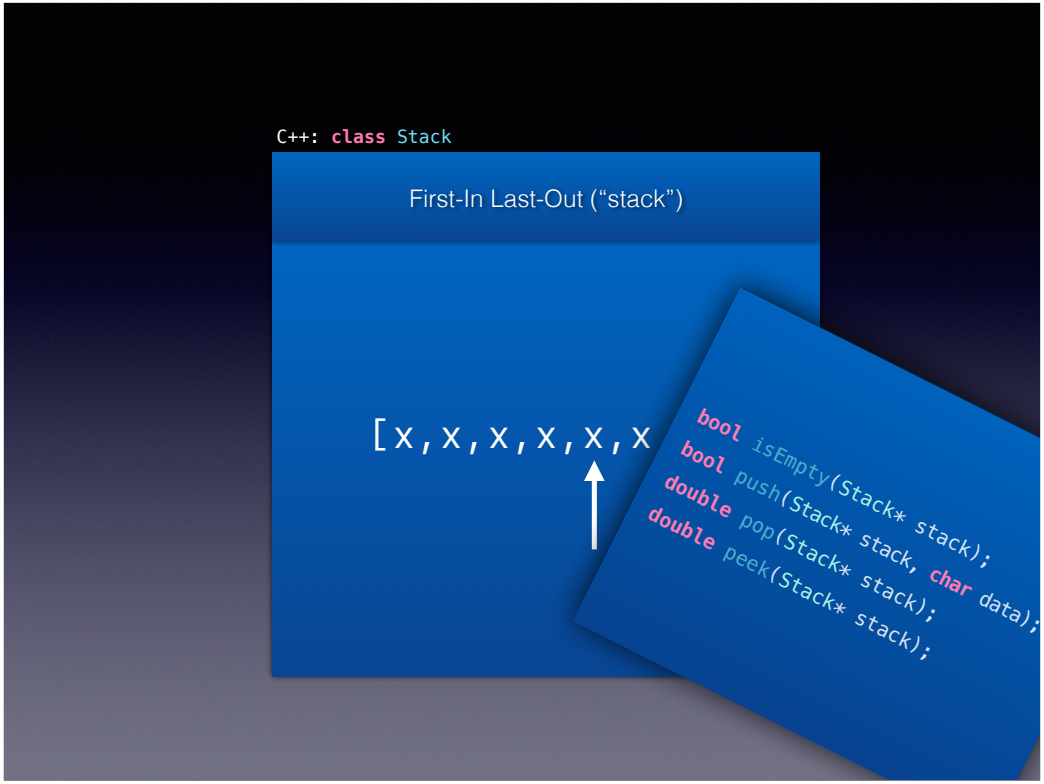
- FILO — First-In Last-Out
- Implementation as Array or LinkedList

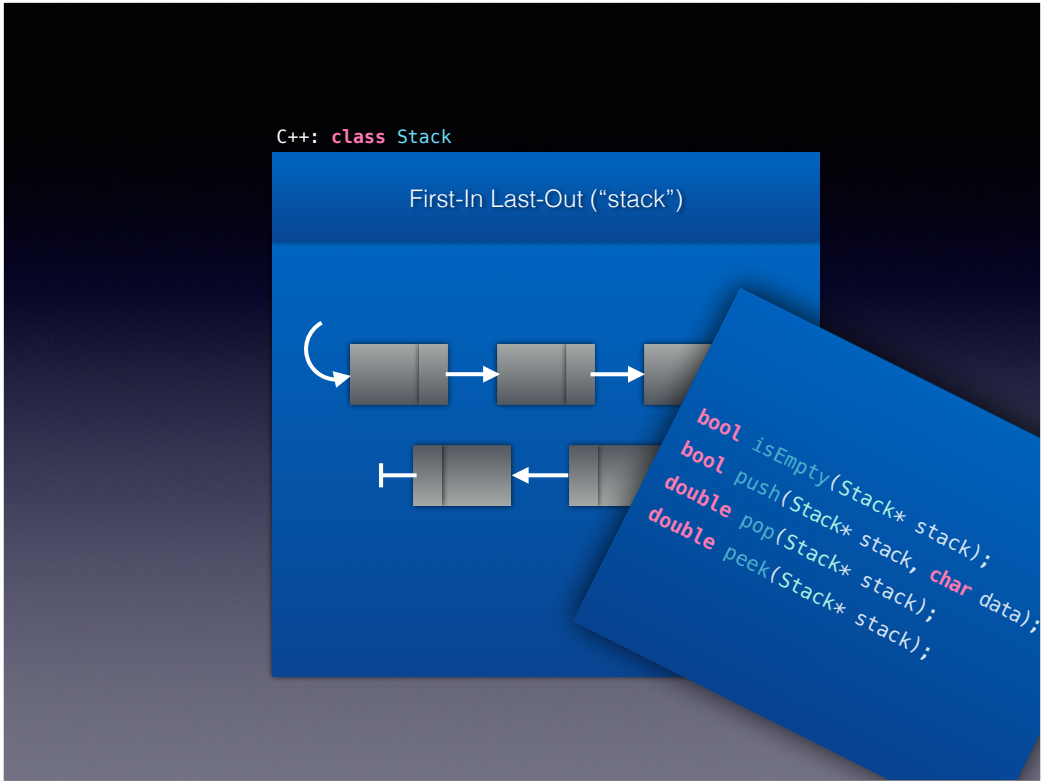


Here is a “class” called Stack, it is a data structure that implements the First-In Last-Out rule, also known as a Stack. There are some top-level functions that do stuff to the stack.

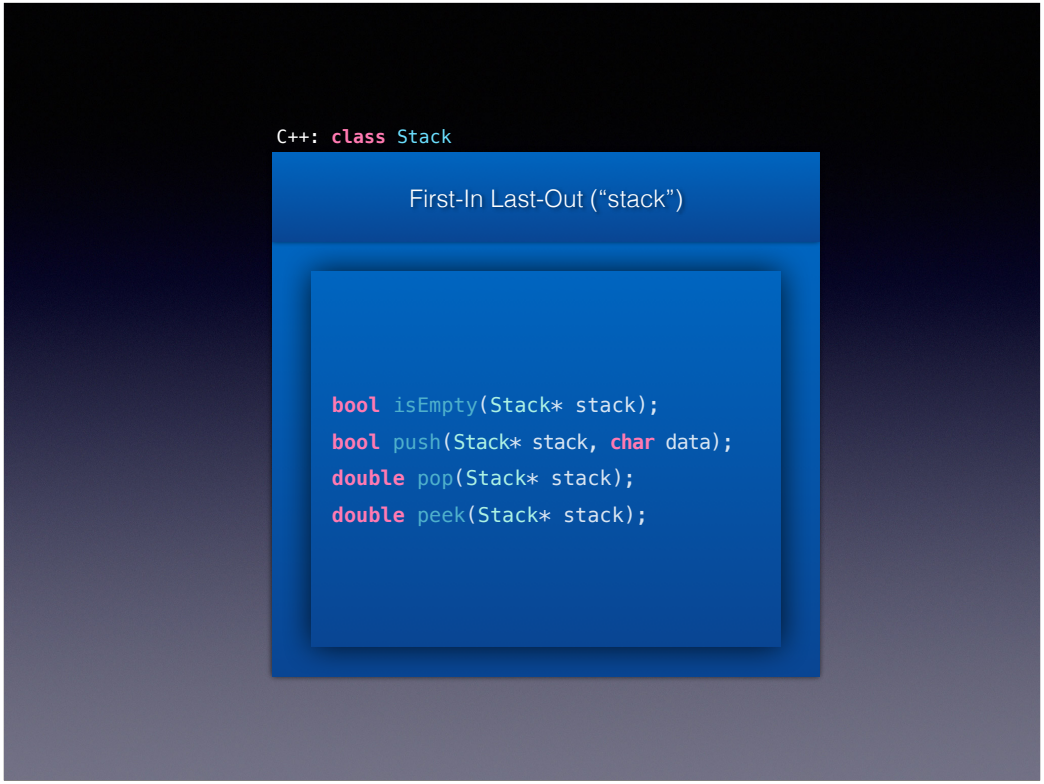
- isEmpty checks if the stack is empty.
- push adds an item to the stack. It returns `true` if it was successfully added, or `false` if it failed.
- pop deletes the top item and returns the value.
- peek “peeks” at the top item without deleting the value.



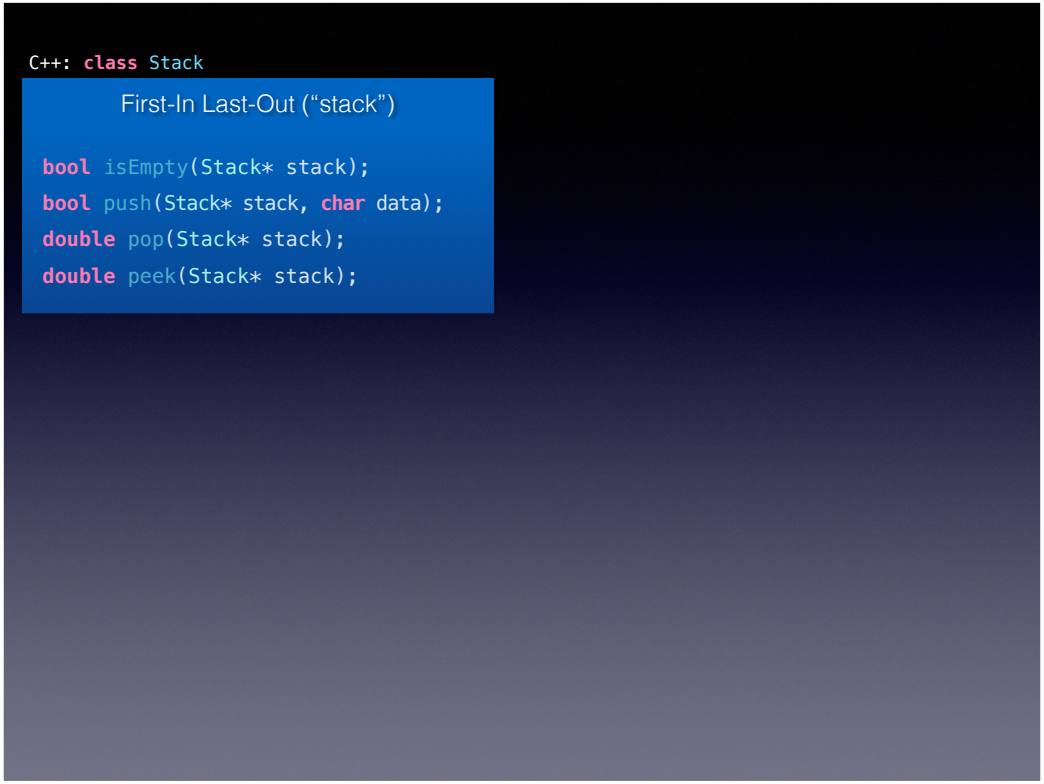
You could implement this as an array with a cursor variable…



Or as a linked list.



So, that’s the difference between the specification, the methods or functions that are exposed, versus the implementation, or what is under-the-hood.



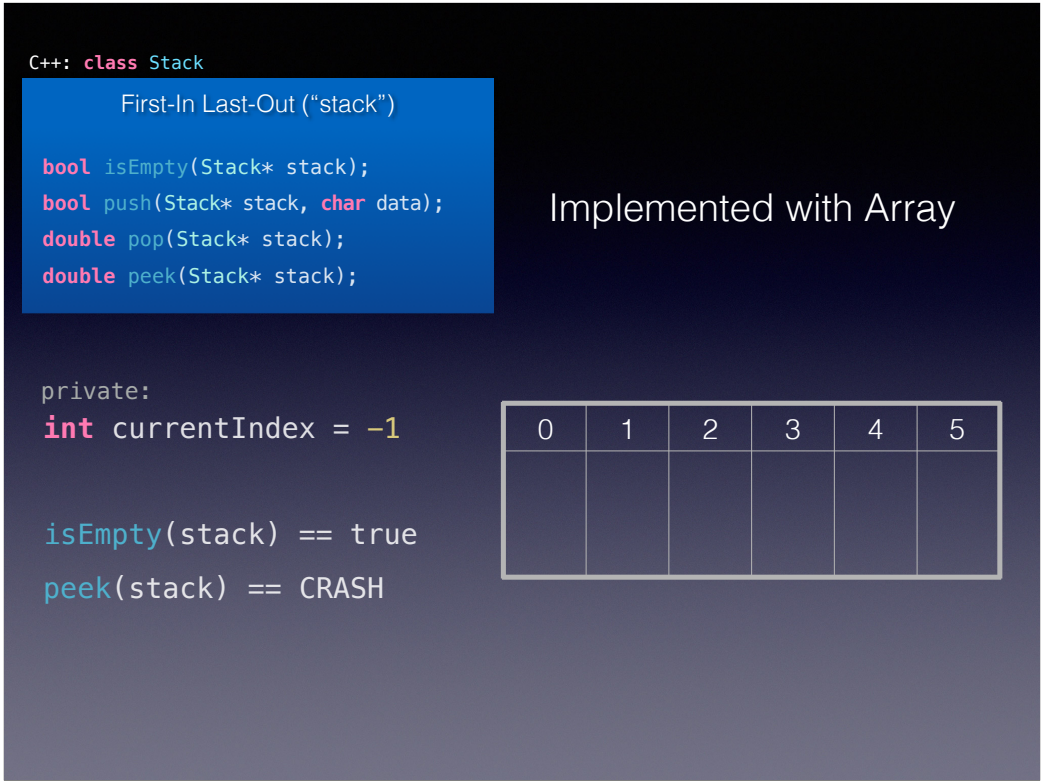
Let’s look at how a stack could look if it was implemented as an array.

C++: `class Stack`

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array



We have to keep track of a "private" variable, which is pointing to the current index. Don't worry about what a "private" variable is yet, you will learn about that when Andy talks about encapsulation and object-oriented programming.

I'm going to keep track of the values of the functions, isEmpty and peek.

Now, I'm going to push a value to the stack, the letter 'g'.

C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array

```
private:
int currentIndex = -1
```

0	1	2	3	4	5

```
isEmpty(stack) == true
peek(stack) == CRASH
```

```
push(stack, 'g')
```

C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array

```
private:
int currentIndex = -1
```

0	1	2	3	4	5
g					

```
isEmpty(stack) == true
peek(stack) == CRASH
```

```
push(stack, 'g')
```

C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

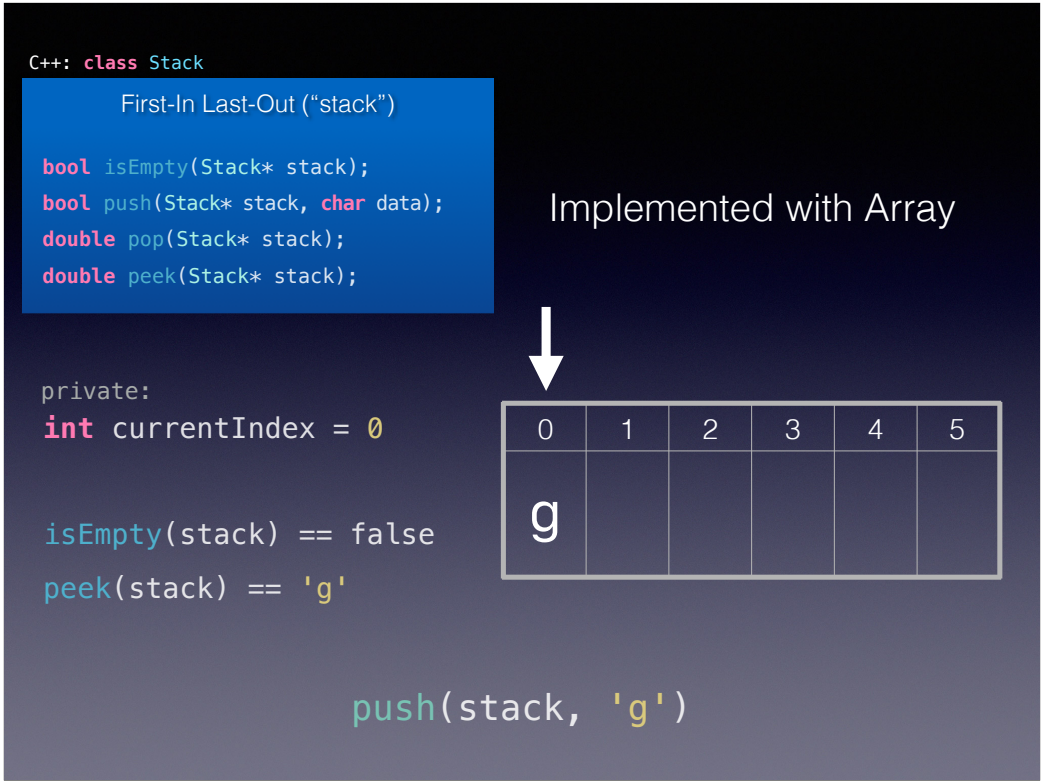
Implemented with Array

```
private:
int currentIndex = -1
```

0	1	2	3	4	5
g					

```
isEmpty(stack) == true
peek(stack) == CRASH
```

```
push(stack, 'g')
```



You’ll notice the variables update, I incremented the currentIndex and if I call isEmpty or peek, you’ll notice that their values are also different now. For example, the peek function returns the value at the current index.

C++: class Stack

First-In Last-Out ("stack")

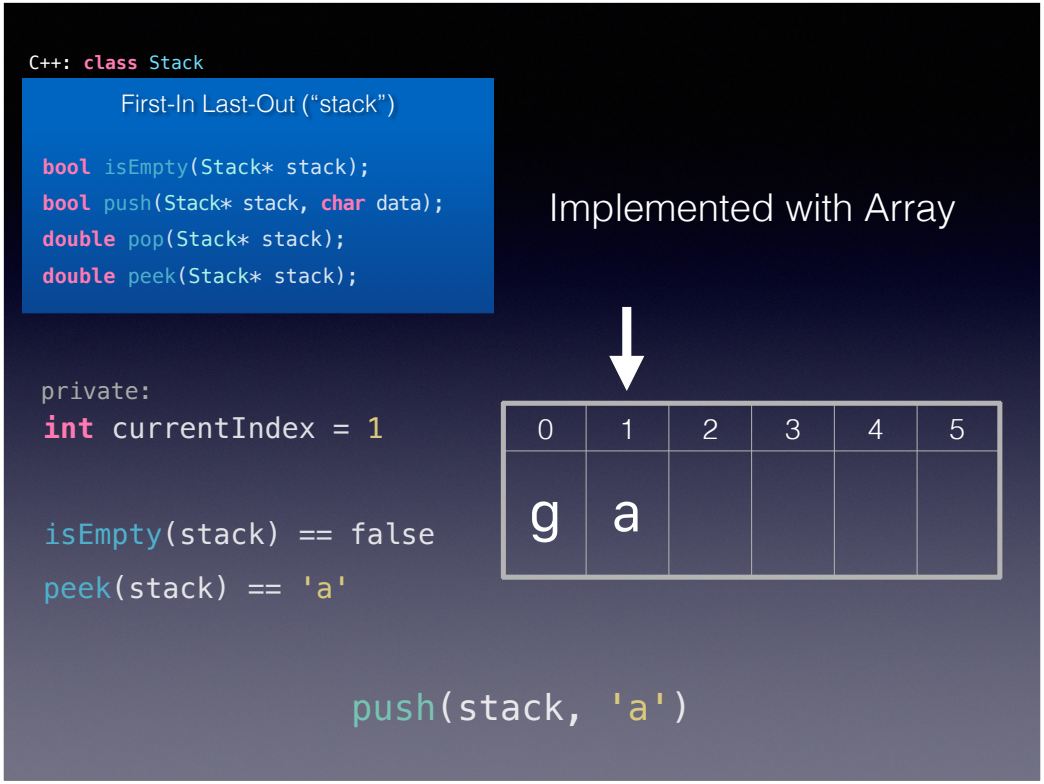
```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array


```
private:
int currentIndex = 0
```

```
isEmpty(stack) == false
peek(stack) == 'g'
```

```
push(stack, 'g')
```



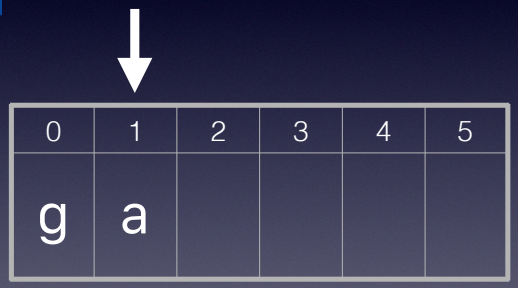
Let’s continue, I’ll push ‘a’

C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

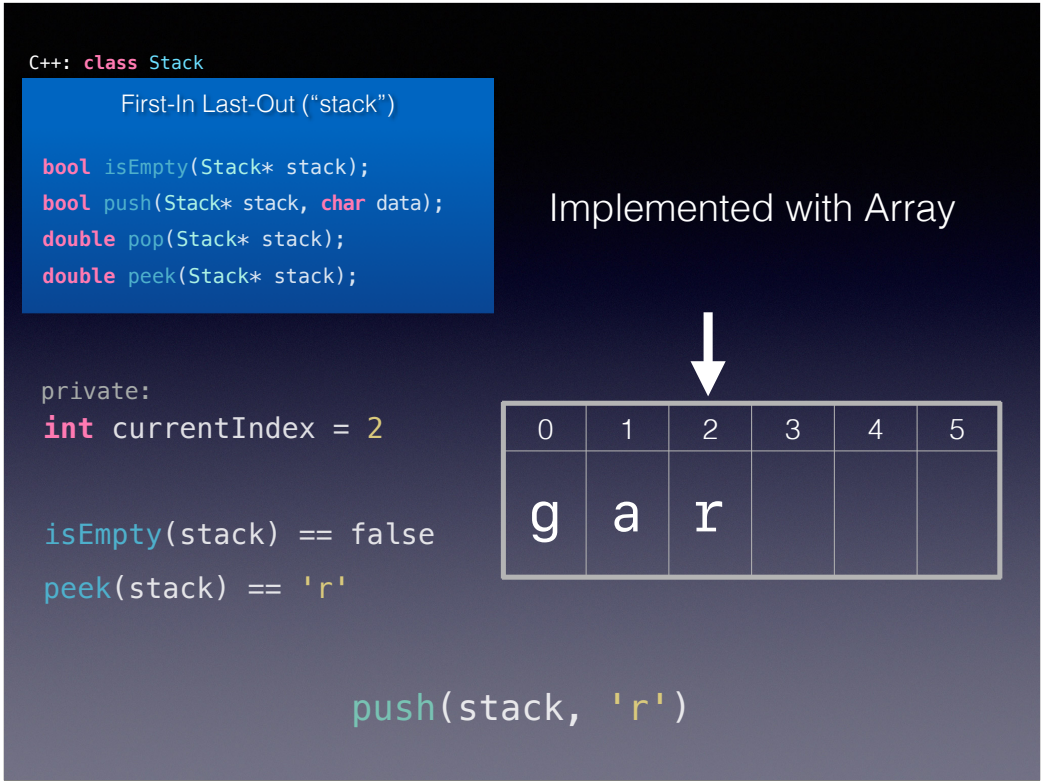
Implemented with Array




```
private:
int currentIndex = 1
```

```
isEmpty(stack) == false
peek(stack) == 'a'
```

```
push(stack, 'a')
```



'r'

C++: class Stack

First-In Last-Out ("stack")

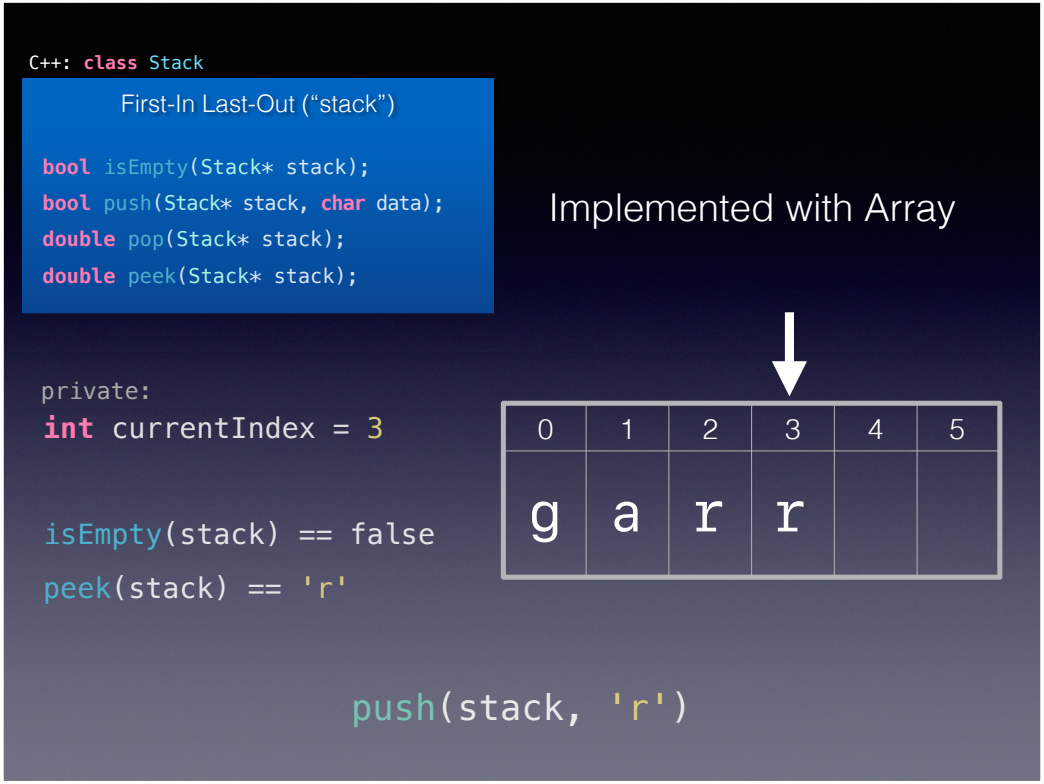
```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array


```
private:
int currentIndex = 2
```

```
isEmpty(stack) == false
peek(stack) == 'r'
```

```
push(stack, 'r')
```



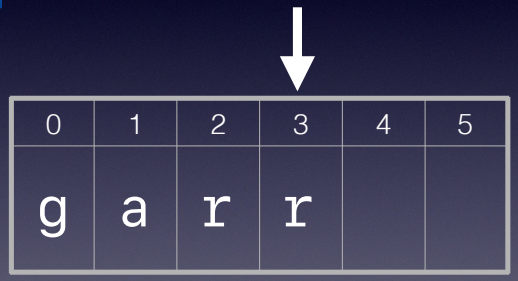
'r'

C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

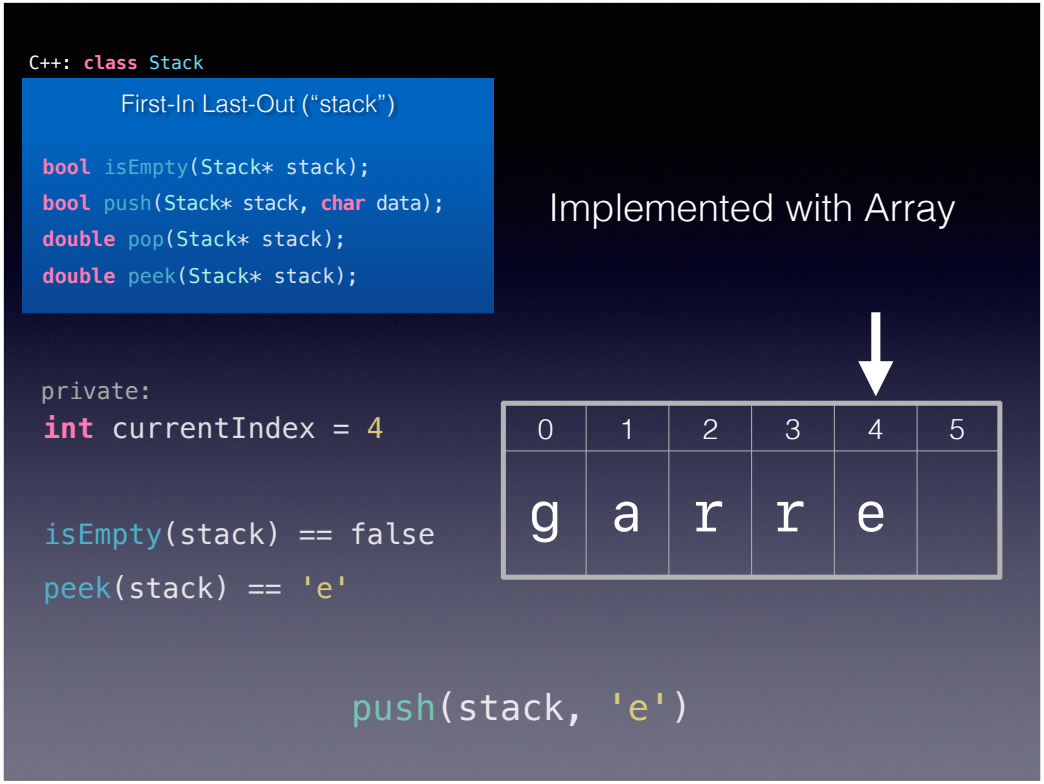
Implemented with Array




```
private:
int currentIndex = 3
```

```
isEmpty(stack) == false
peek(stack) == 'r'
```

```
push(stack, 'r')
```



'e'

C++: class Stack

First-In Last-Out ("stack")

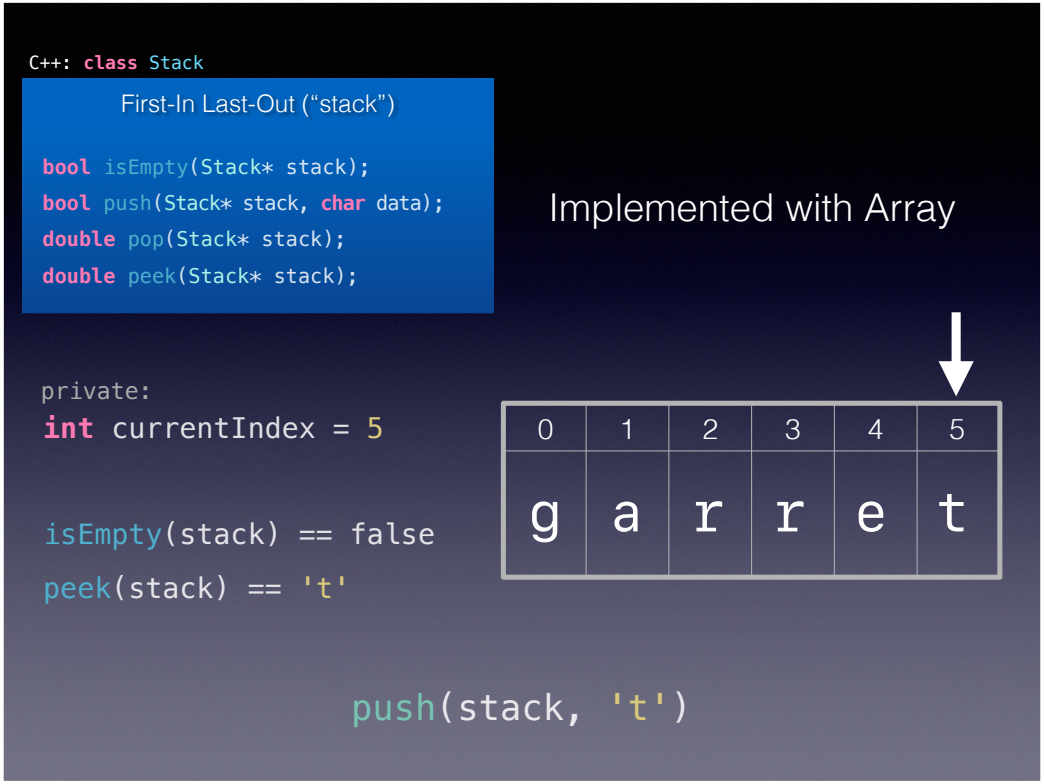
```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array

```
private:
int currentIndex = 4
```


```
isEmpty(stack) == false
peek(stack) == 'e'
```

```
push(stack, 'e')
```



’t’

Can anyone predict what the major con of using an array is?

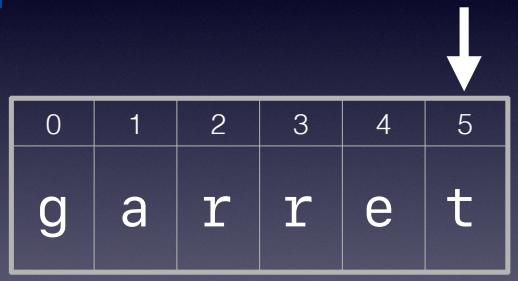
C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

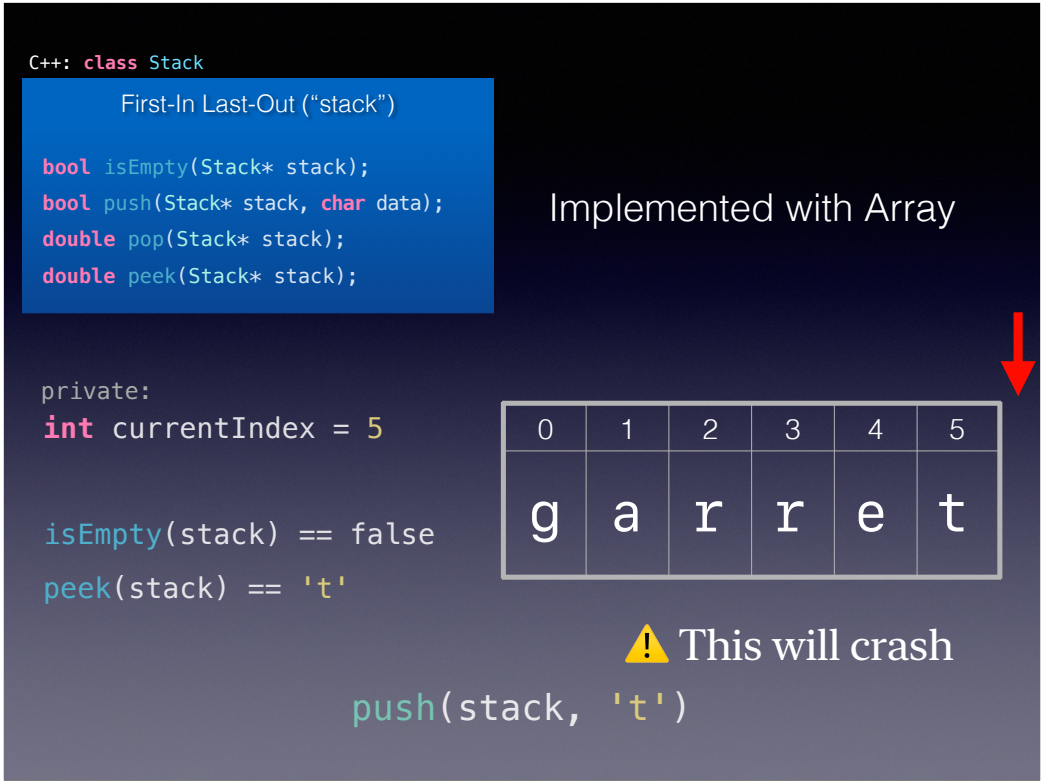
Implemented with Array

```
private:
int currentIndex = 5
```

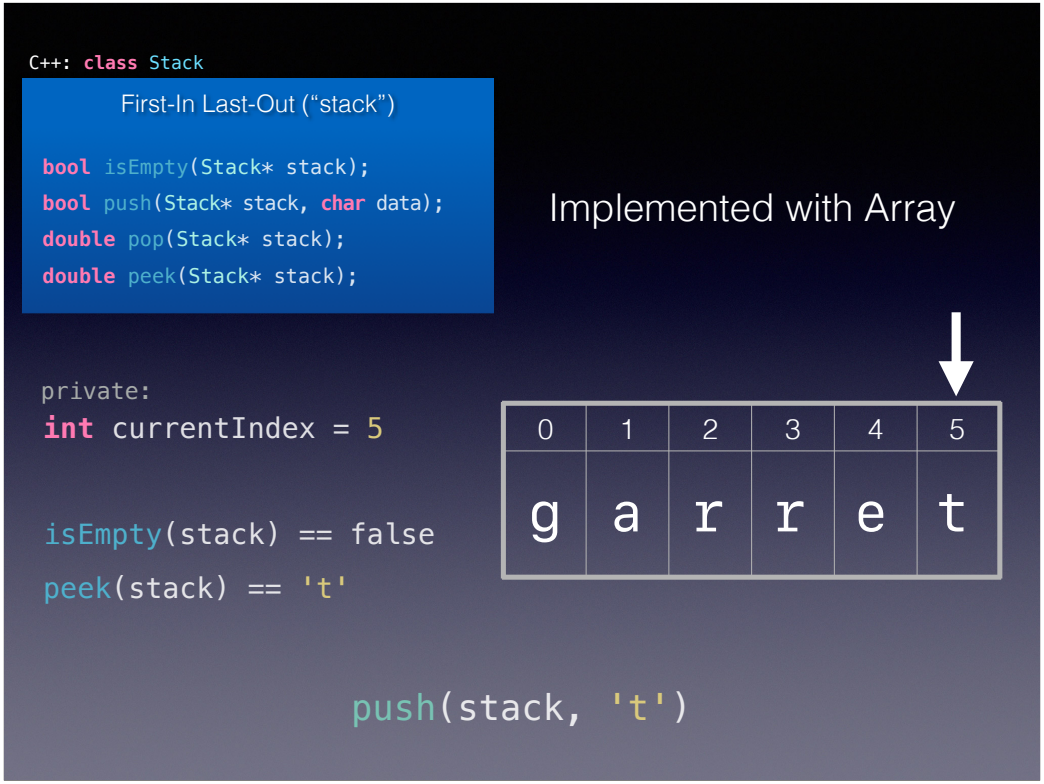



```
isEmpty(stack) == false
peek(stack) == 't'
```

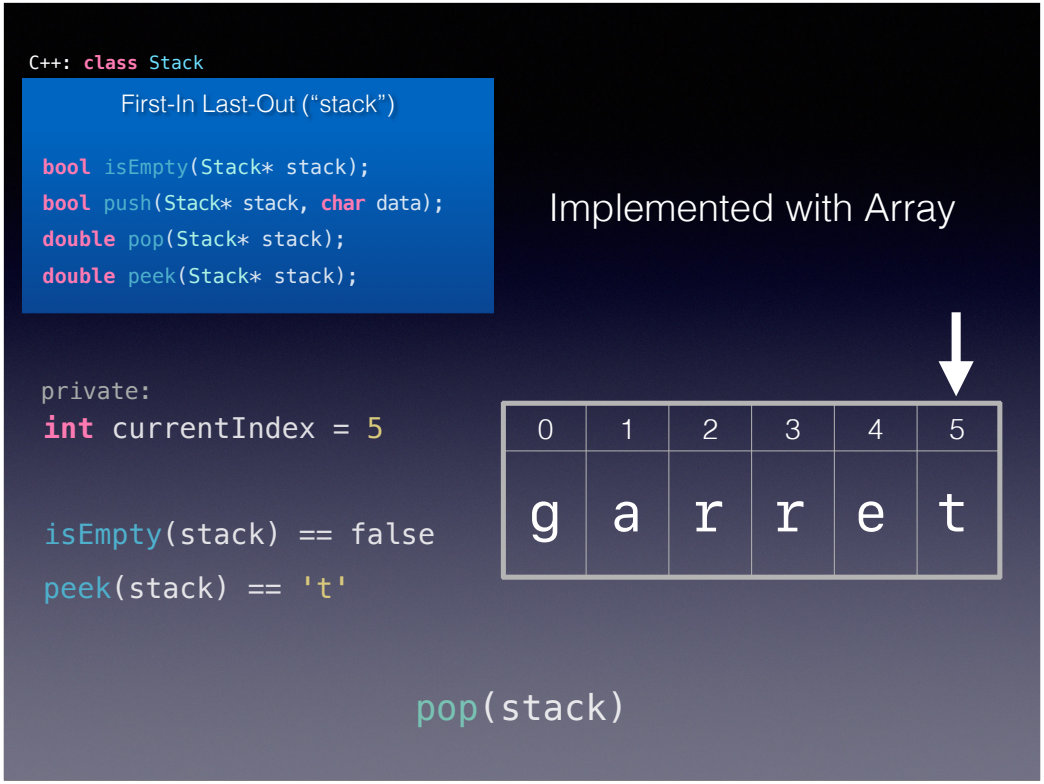
```
push(stack, 't')
```



It has a fixed size, if we try to push more, it won't work.



Ok, let's go back.



Now, let’s try popping an item from the stack.

C++: class Stack

First-In Last-Out ("stack")

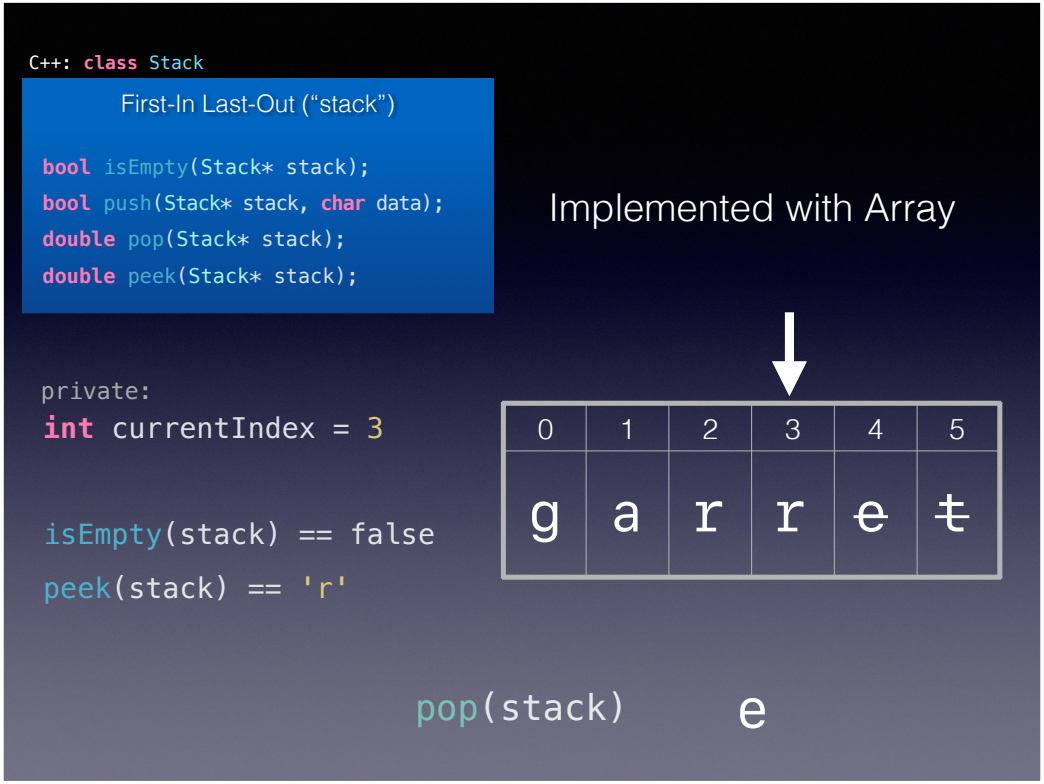
```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array


```
private:
int currentIndex = 4
```

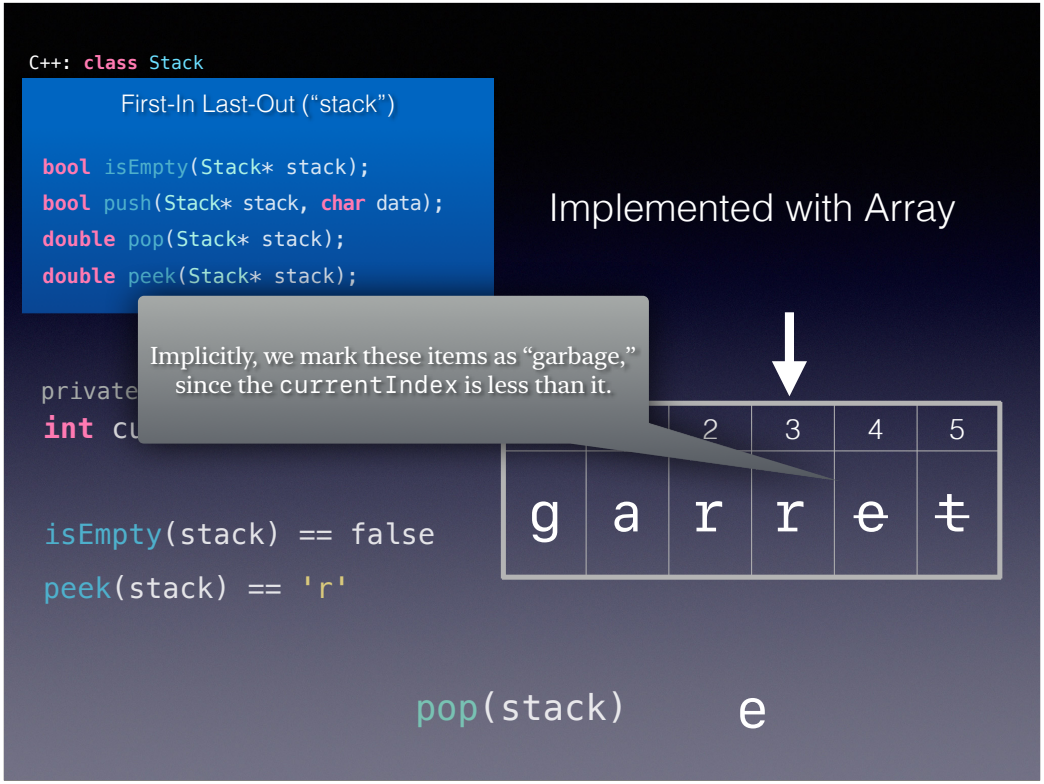
```
isEmpty(stack) == false
peek(stack) == 'e'
```

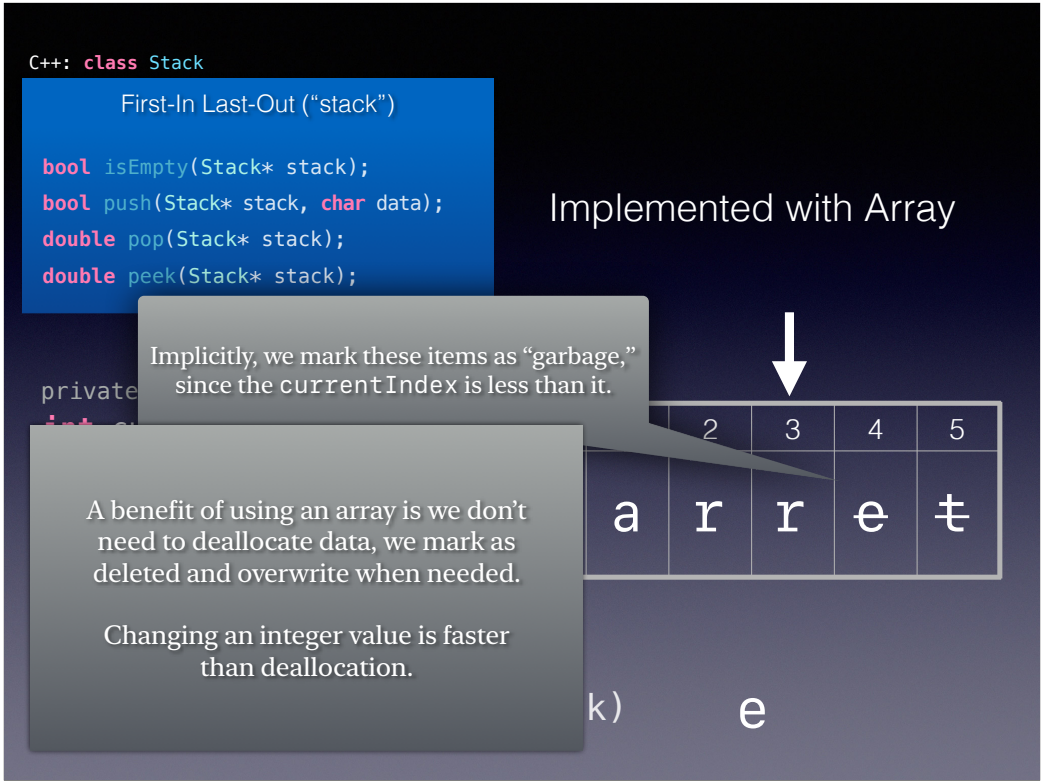
```
pop(stack)    t
```



You'll notice that I'm not actually removing the values. Implicitly, we are marking each item as "deleted", or "garbage" since the currentIndex is less than it.

We don't need to delete the data, so we won't. Just decrementing the currentIndex variable is a lot faster than deallocating the space. This is similar to how your computer's hard drive works. When you delete a file, you aren't actually removing the file. Your computer will mark it as deleted, and when the time comes that you need to use that space, it just overwrites the "deleted" data.



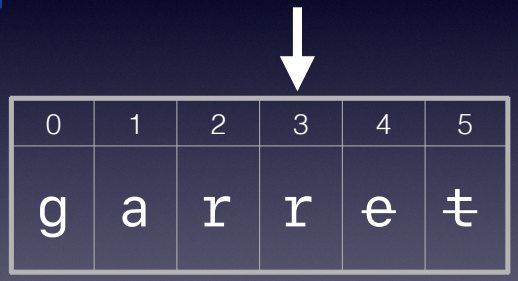


C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array




```
private:
int currentIndex = 3
```

```
isEmpty(stack) == false
peek(stack) == 'r'
```

pop(stack) e

C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array


```
private:
int currentIndex = 4
```

```
isEmpty(stack) == false
peek(stack) == 'a'
```

```
push(stack, 'a')
```

C++: class Stack

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with Array

```
private:
int currentIndex = 4
```


```
isEmpty(stack) == false
peek(stack) == 'a'
```

```
push(stack, 'a')
```

C++: `class Stack`

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with LinkedList

Do you want to use:

- ☐ singly-linked list
- ☐ doubly-linked list

C++: `class Stack`

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Implemented with LinkedList

Do you want to use:

☑ singly-linked list

☐ doubly-linked list

First-In Last-Out (“stack”)

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

First-In Last-Out (“stack”)

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Task 1: Implement a Stack using a singly-linked list in C.

First-In Last-Out (“stack”)

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Task 1: Implement a Stack using a singly-linked list in C.

Hint: Stacks only care about what’s on top.

Author: Alan Chu (ualch9@gmail.com)

First-In Last-Out ("stack")

```
bool isEmpty(Stack* stack);
bool push(Stack* stack, char data);
double pop(Stack* stack);
double peek(Stack* stack);
```

Task 1: Implement a Stack using a singly-linked list in C.

Hint: Stacks only care about what's on top.

Task 2: Write Unit Tests for the stack