

# Collections, Part Two

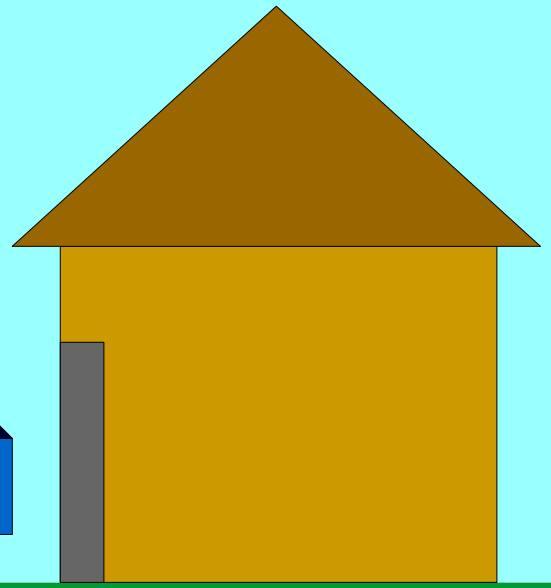
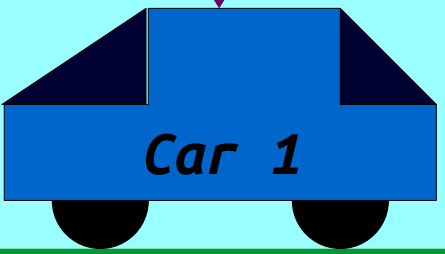
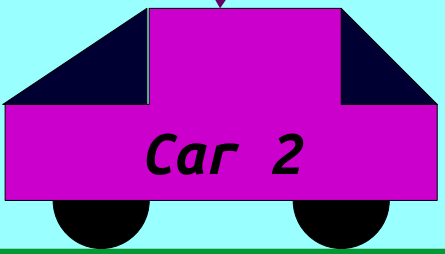
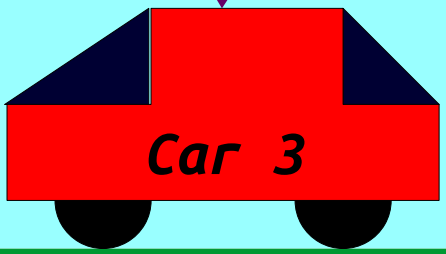
# Outline for Today

- ***Stacks***
  - Pancakes meets parsing!
- ***Queues***
  - Playing some music!

Stack

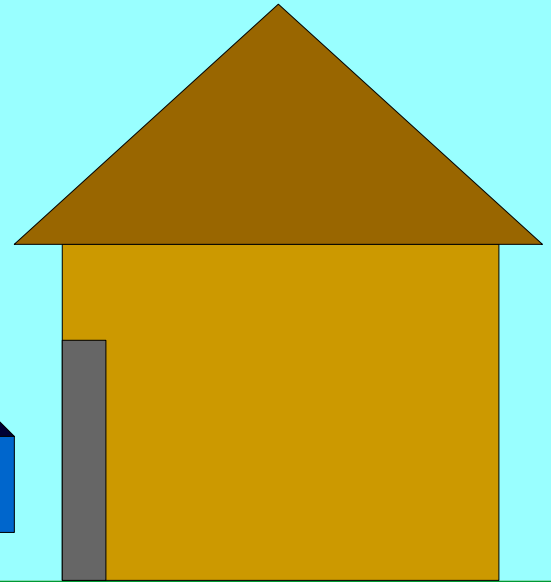
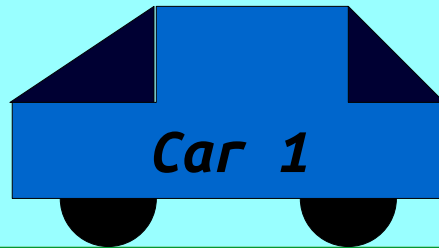
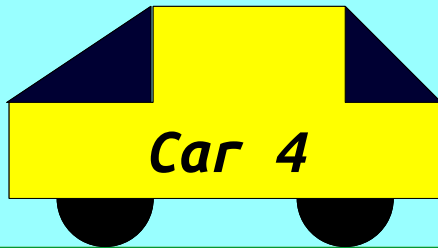
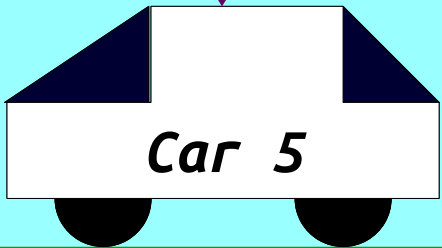
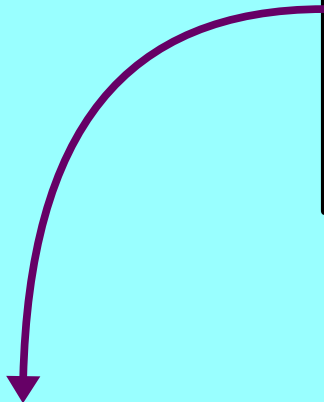
This car  
can't leave...

... until these  
two do.

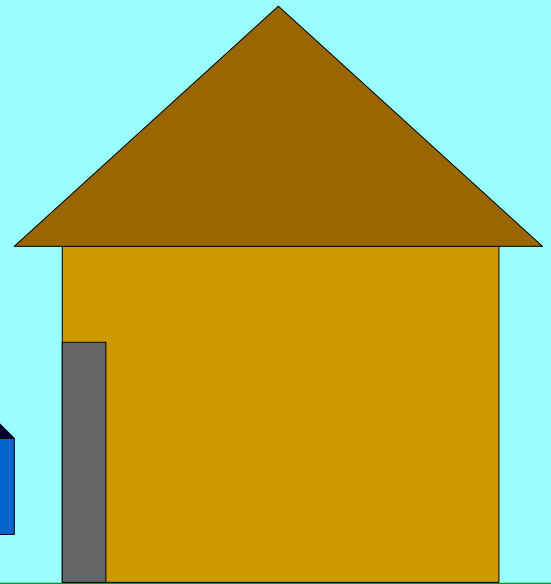
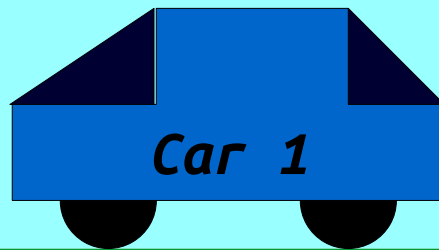
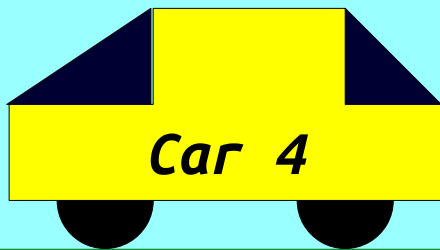
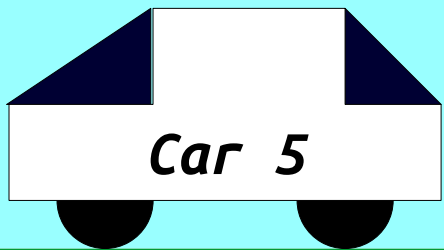


*Thanks to Nick Troccoli for this example!*

Any new car precedes all the old cars. Only this car can leave.



*Thanks to Nick Troccoli for this example!*



*Thanks to Nick Troccoli for this example!*

# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be *pushed* on top of the stack or *popped* from the top of the stack.

# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.





# Stack

137

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.



# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.



# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.

42

137



# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.



# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.

271

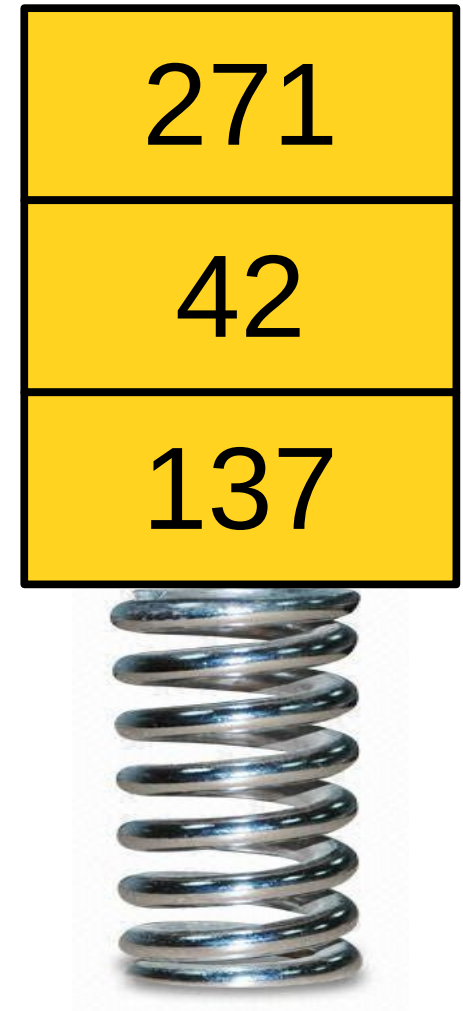
42

137



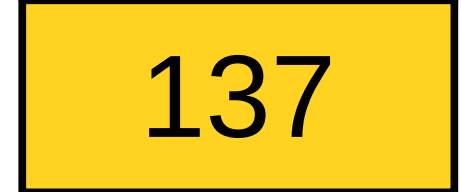
# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.



# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.



# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be *pushed* on top of the stack or *popped* from the top of the stack.





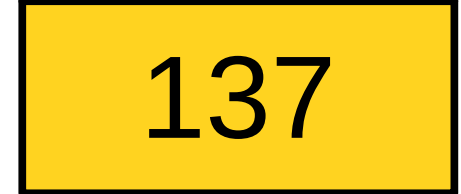
# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.



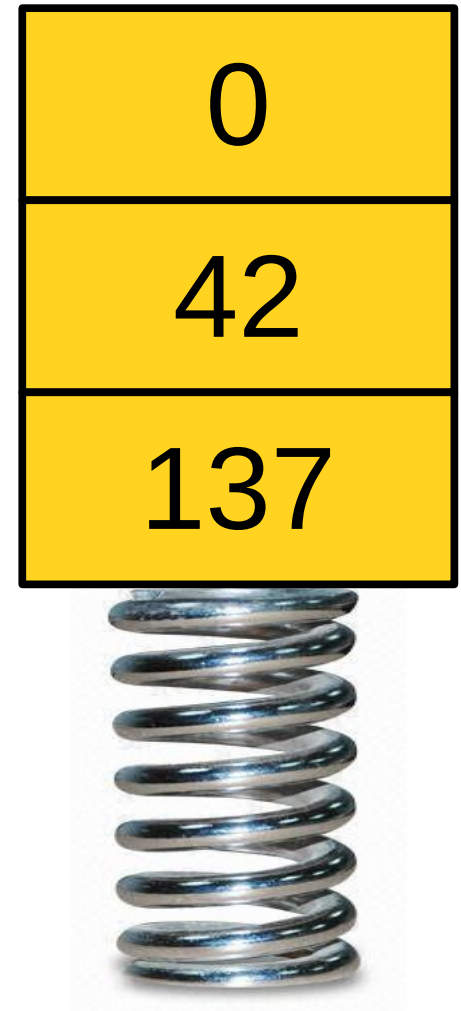
# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.



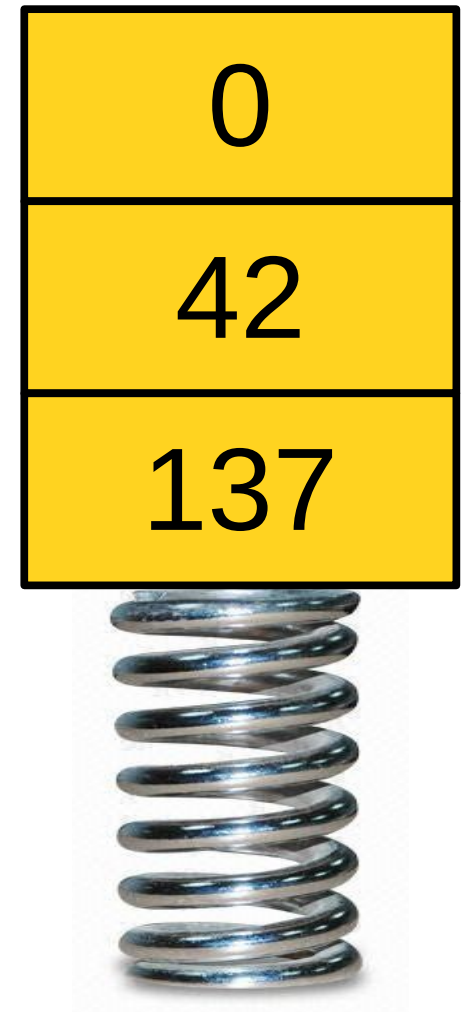
# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.



# Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the topmost element of a Stack can be accessed.
- Do you see why we call it the *call stack* and talk about *stack frames*?



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

Formulate a hypothesis,  
but ***don't post anything  
in chat just yet.***

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

Now, *private chat me your best guess*. Not sure? Just answer “??”

# Stack

What does this code print?

```
Stack<char> s1, s2;
```

```
s1.push('a');
```

```
s1.push('b');
```

```
s1.push('c');
```

```
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}
```

```
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1



s2

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1



s2



# Stack

What does this code print?

```
Stack<char> s1, s2;
```

```
s1.push('a');
```

```
s1.push('b');
```

```
s1.push('c');
```

```
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}
```

```
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1



s2

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1

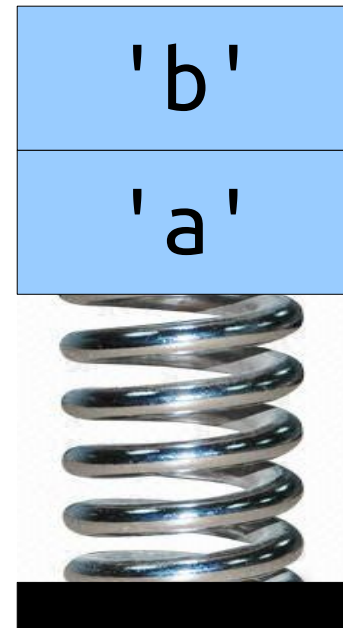


s2

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1

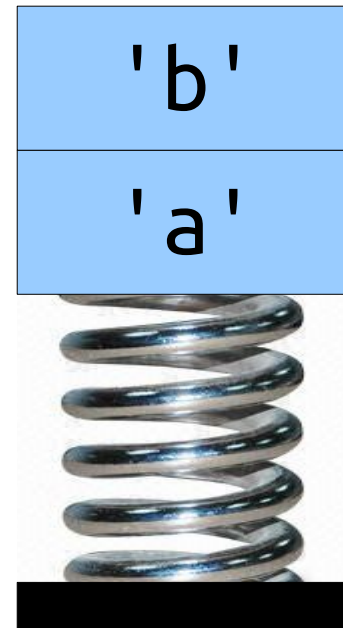


s2

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1

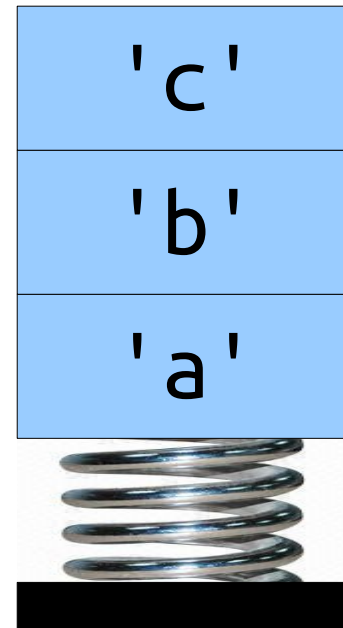


s2

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1



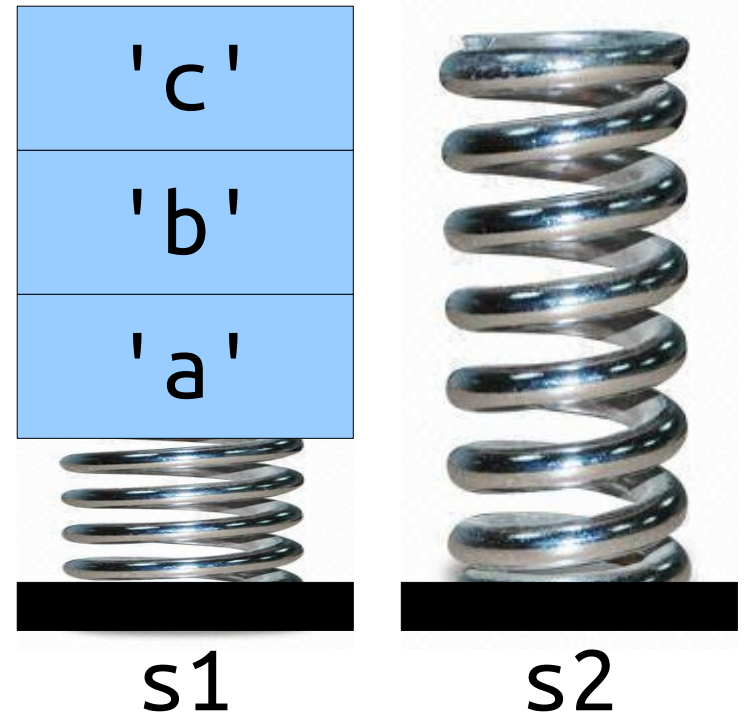
s2

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');
```

```
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



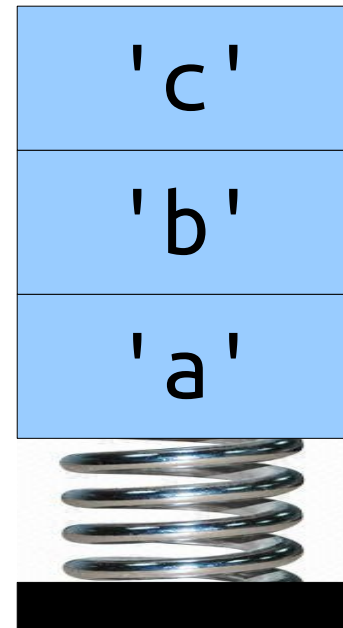
# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');
```

```
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}
```

```
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



s1



s2

# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

'c'

'b'

'a'



s1



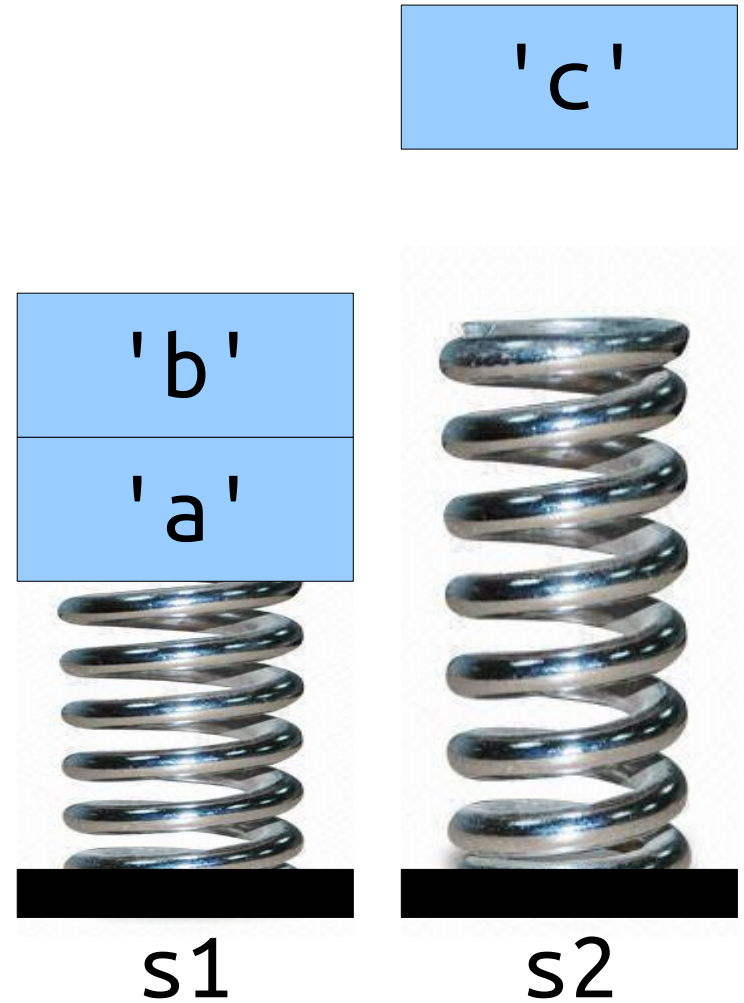
s2



# Stack

What does this code print?

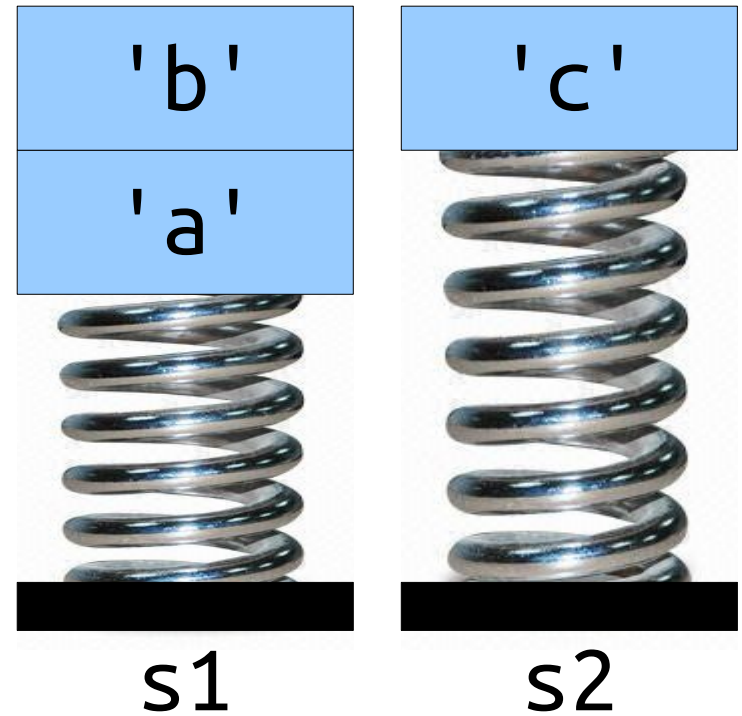
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

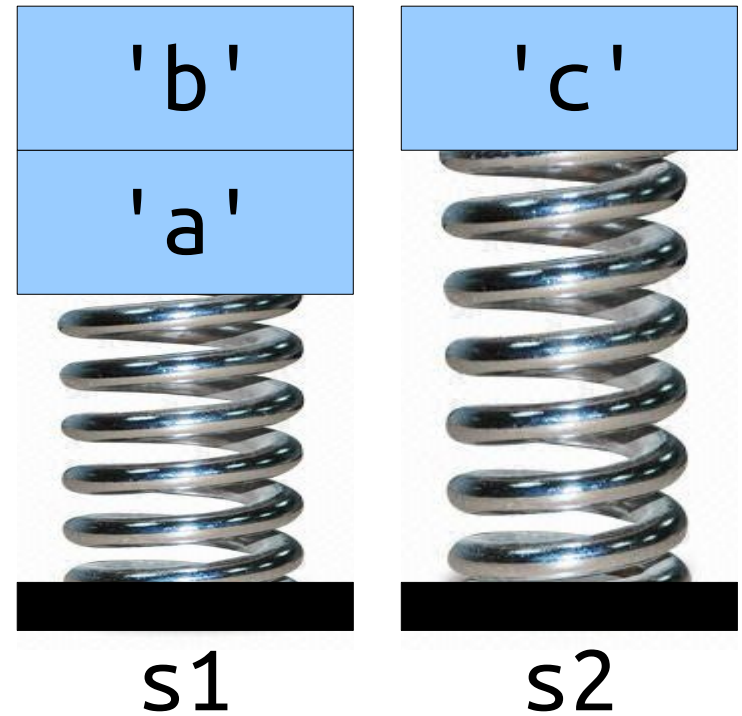


# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');
```

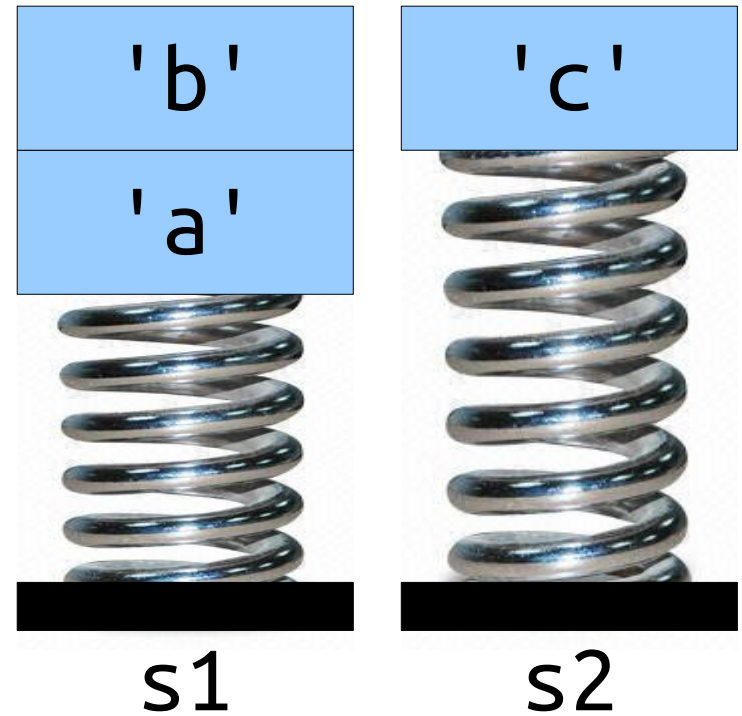
```
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

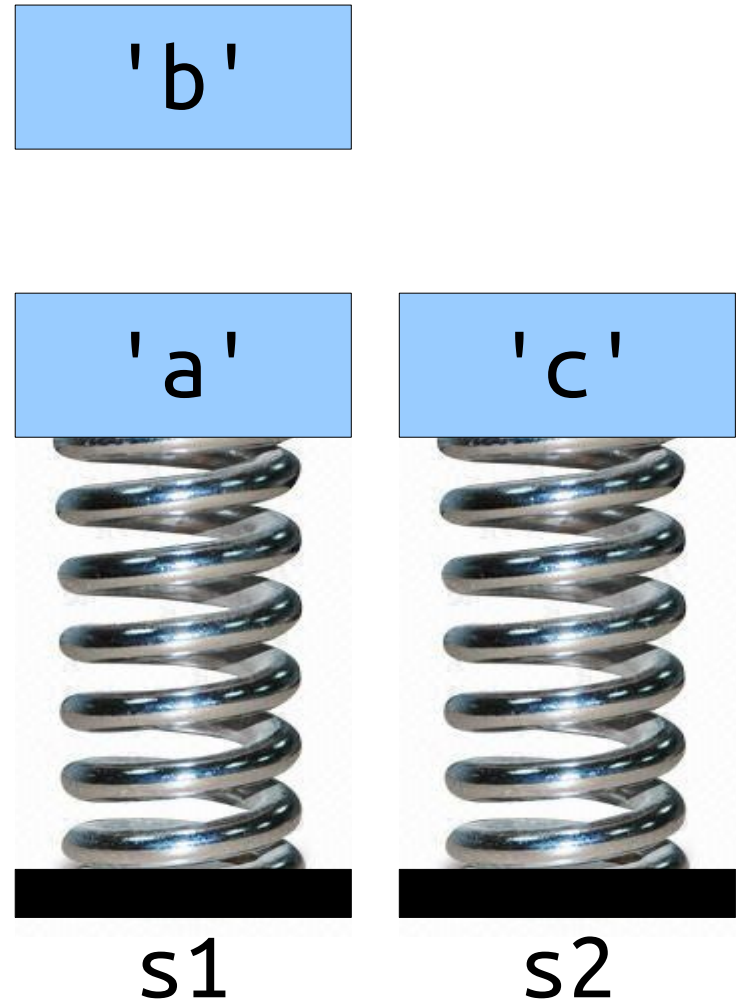
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

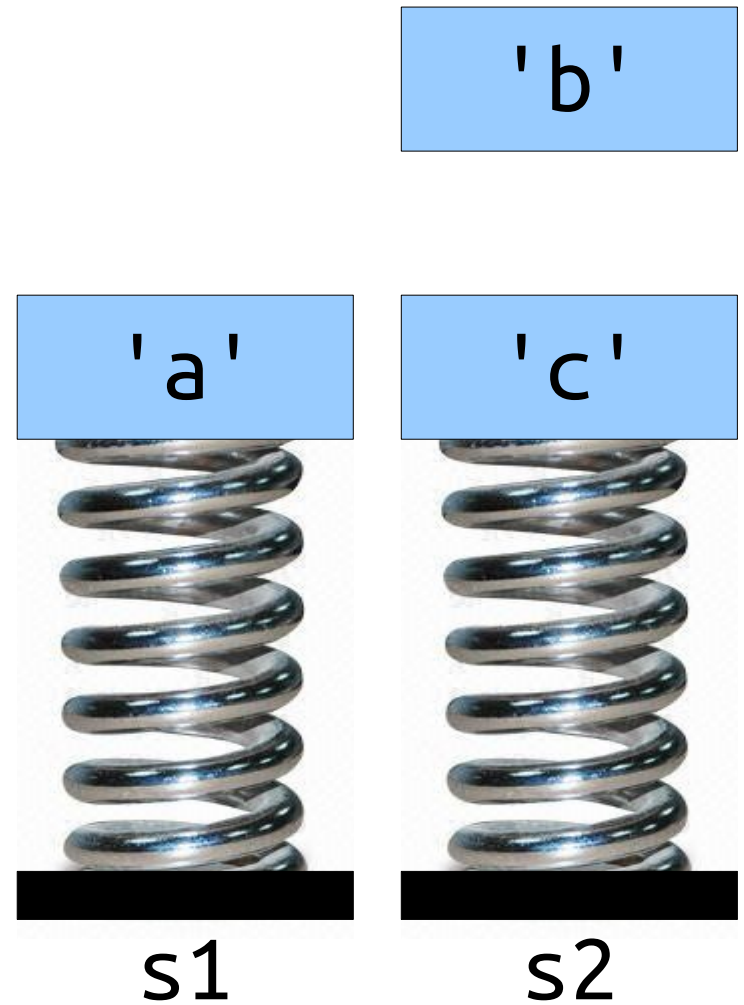
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

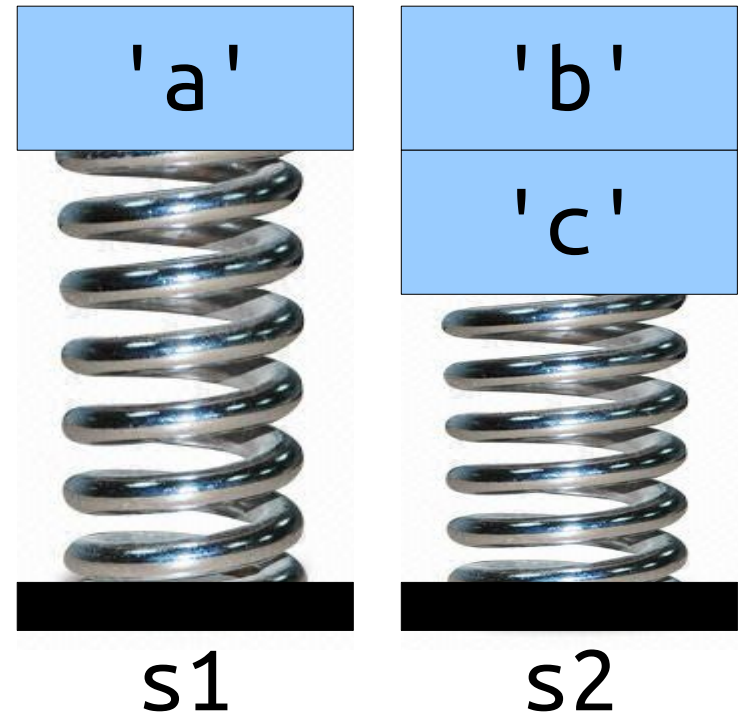
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

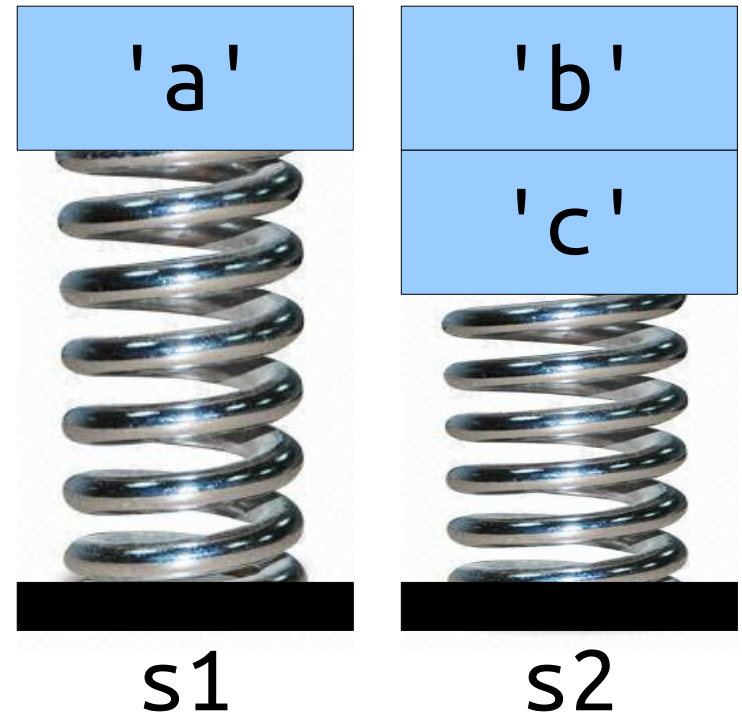


# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');
```

```
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

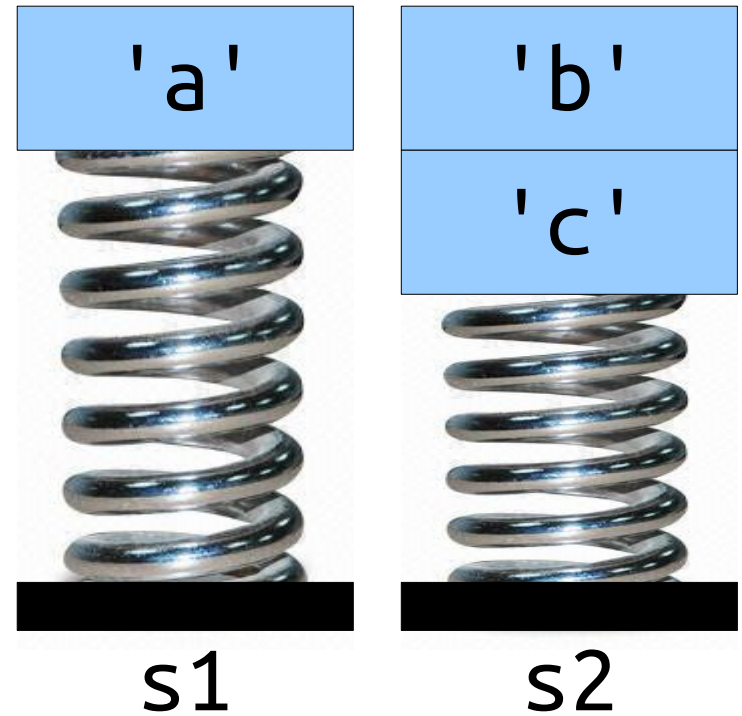




# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

'a'



s1

'b'  
'c'

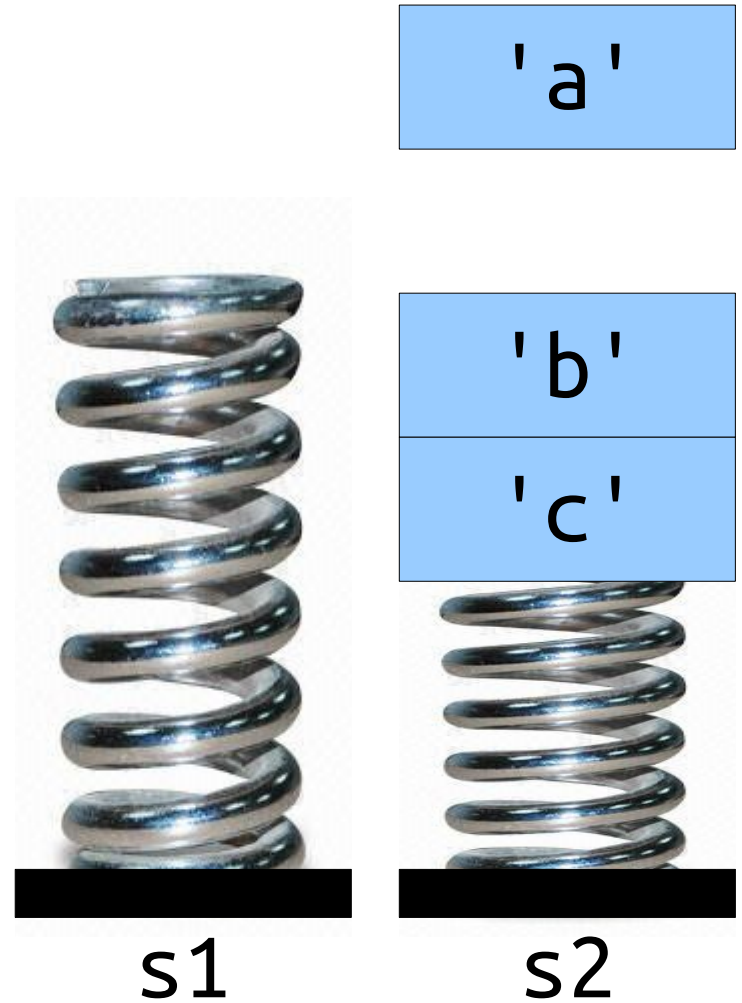


s2

# Stack

What does this code print?

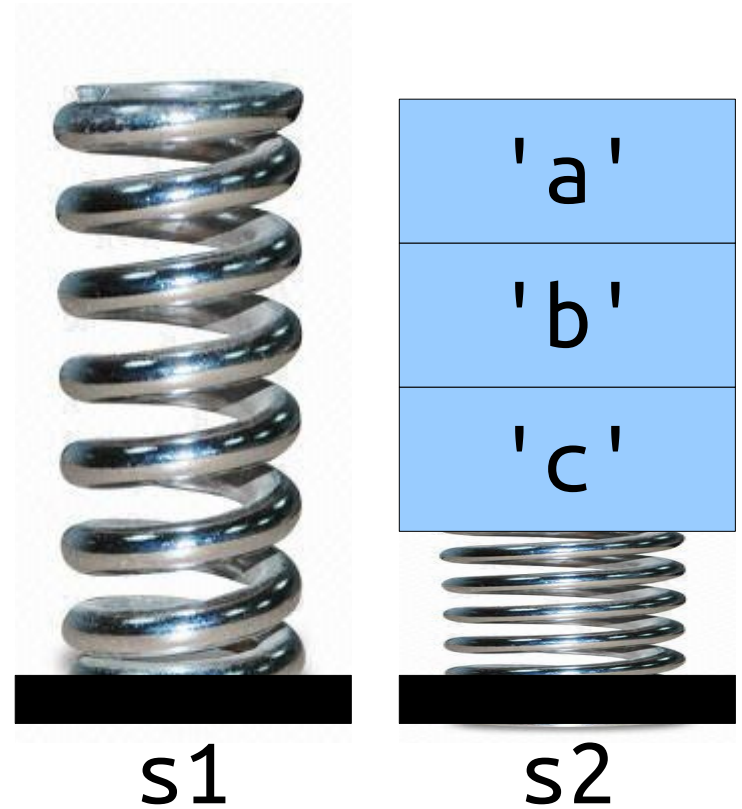
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

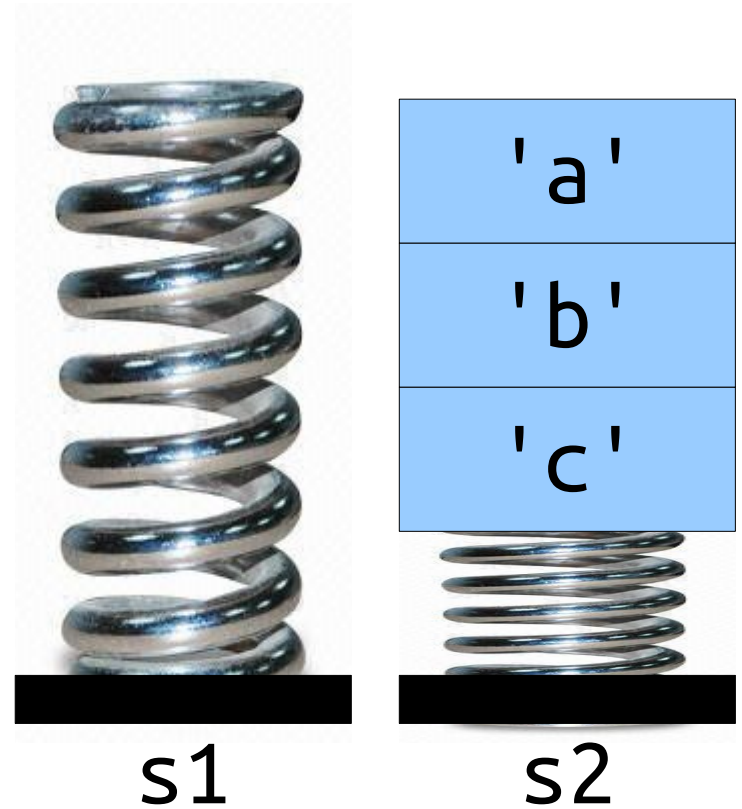
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

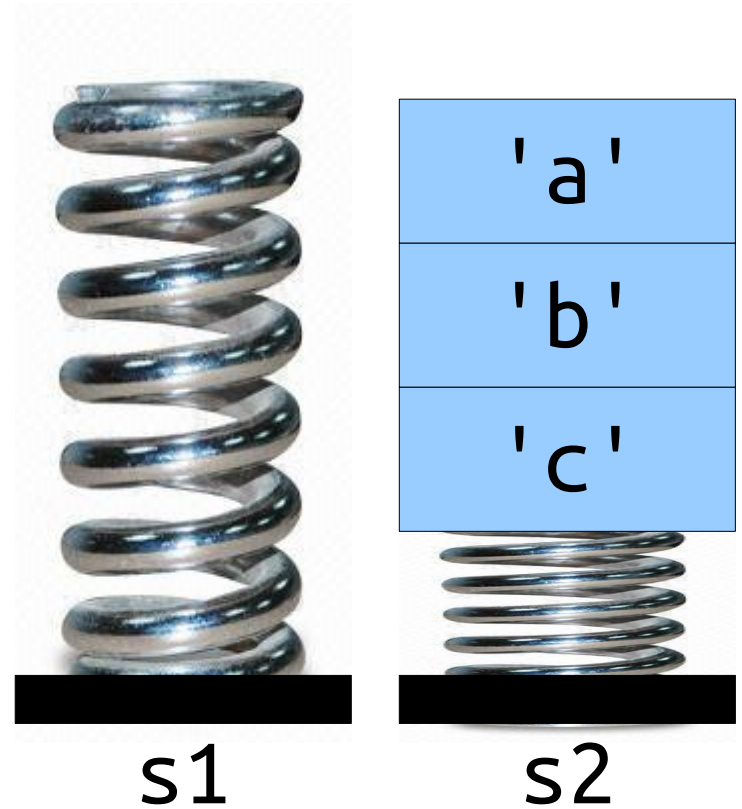
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

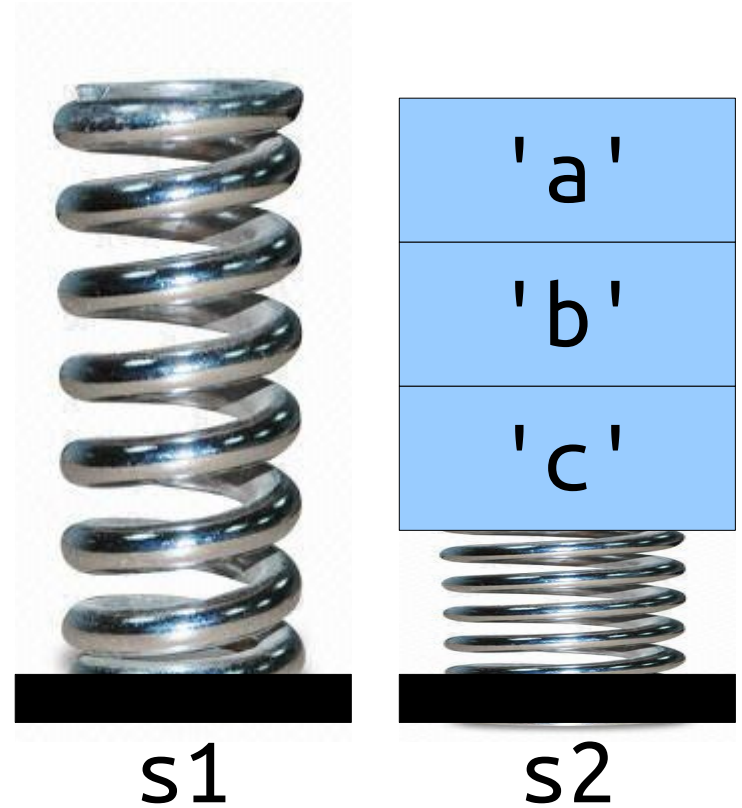
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

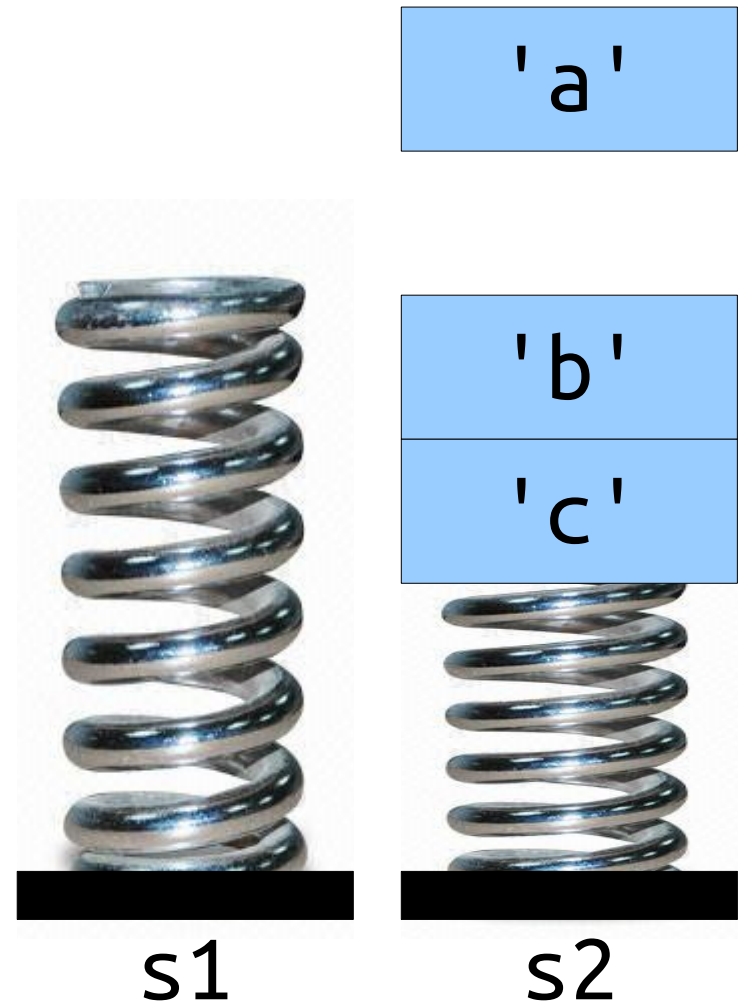
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



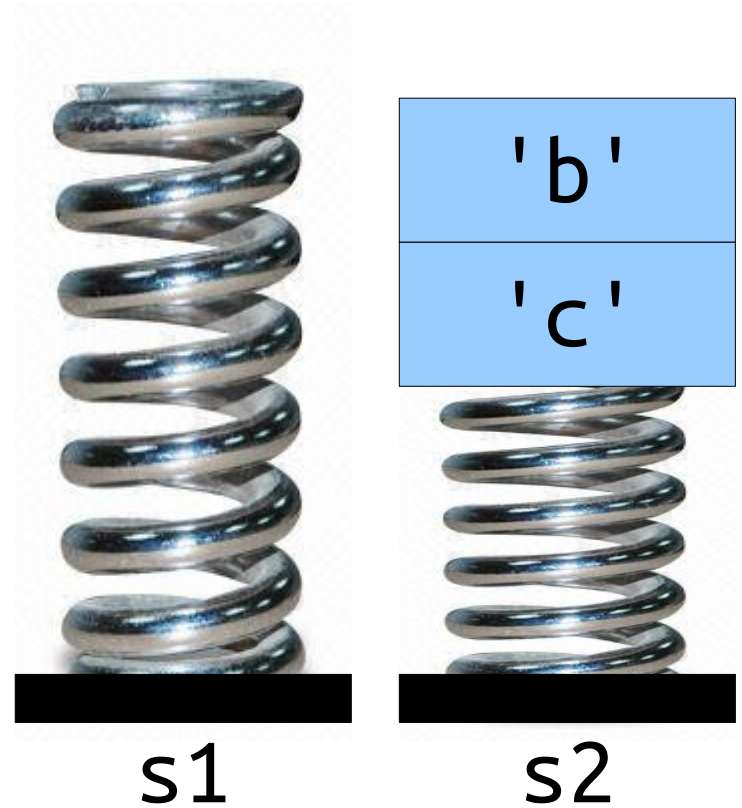


# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

'a'

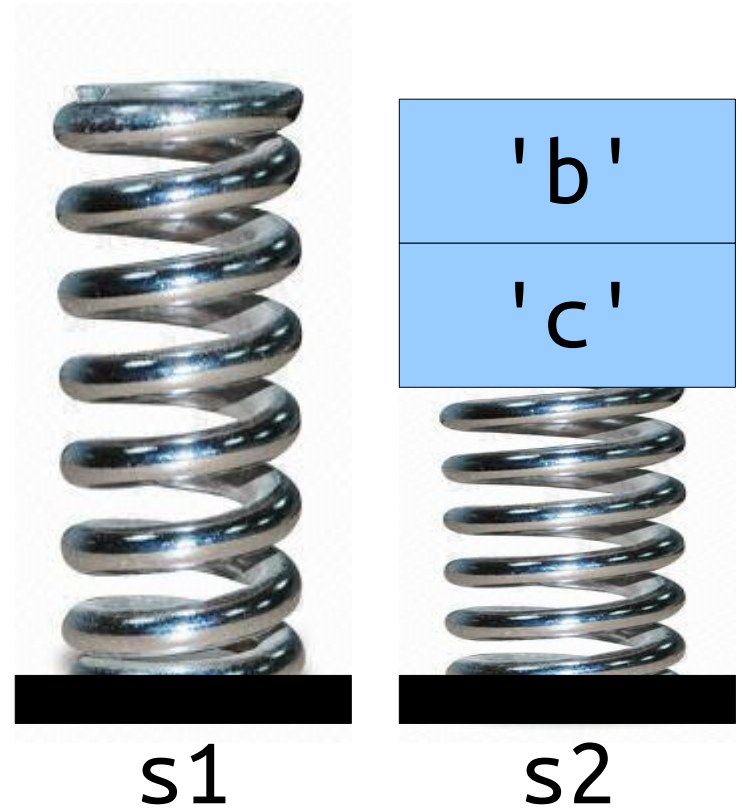


# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

'a'

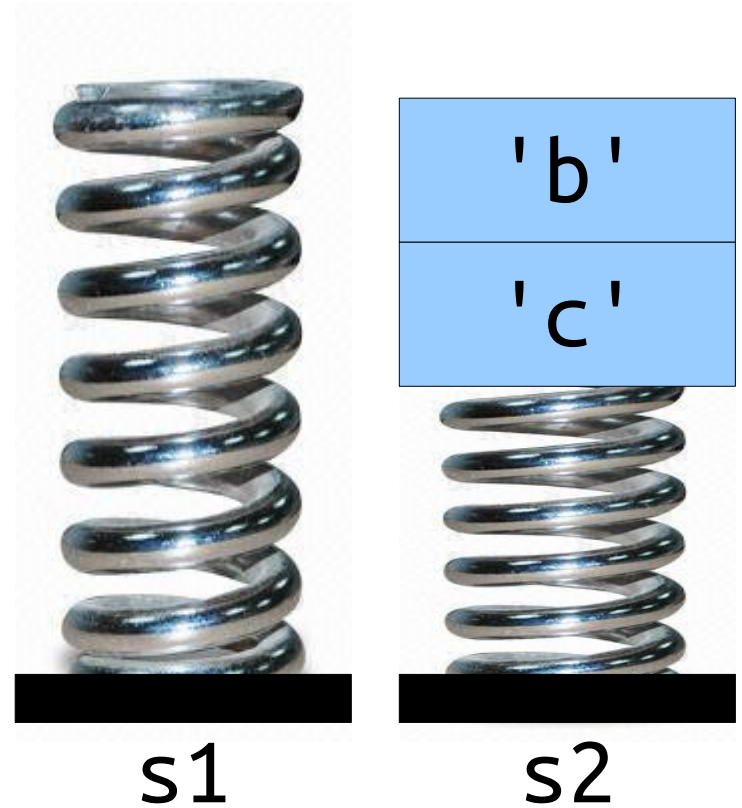


# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

'a'



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

'a'



s1

'b'

'c'

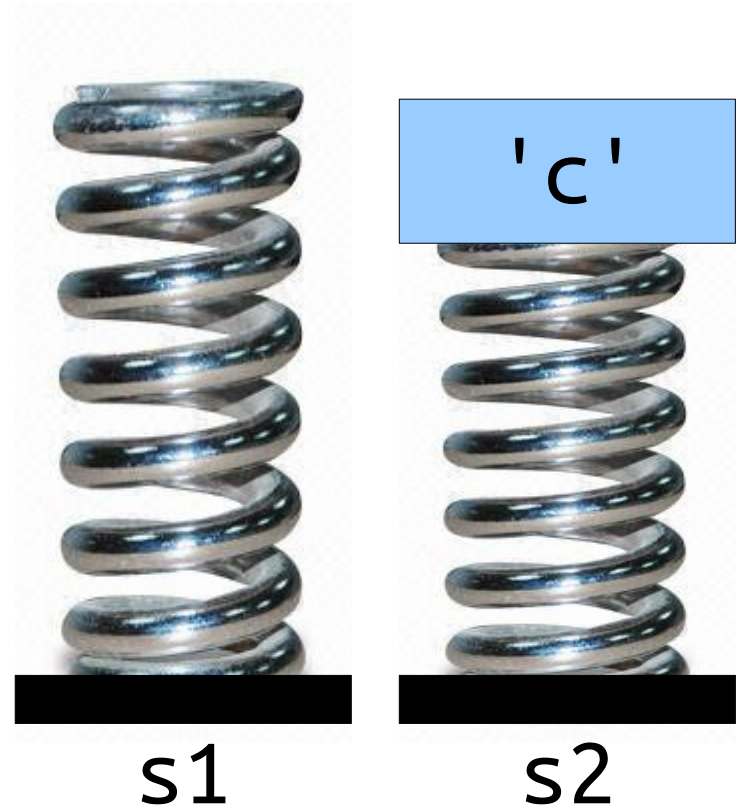


s2

# Stack

What does this code print?

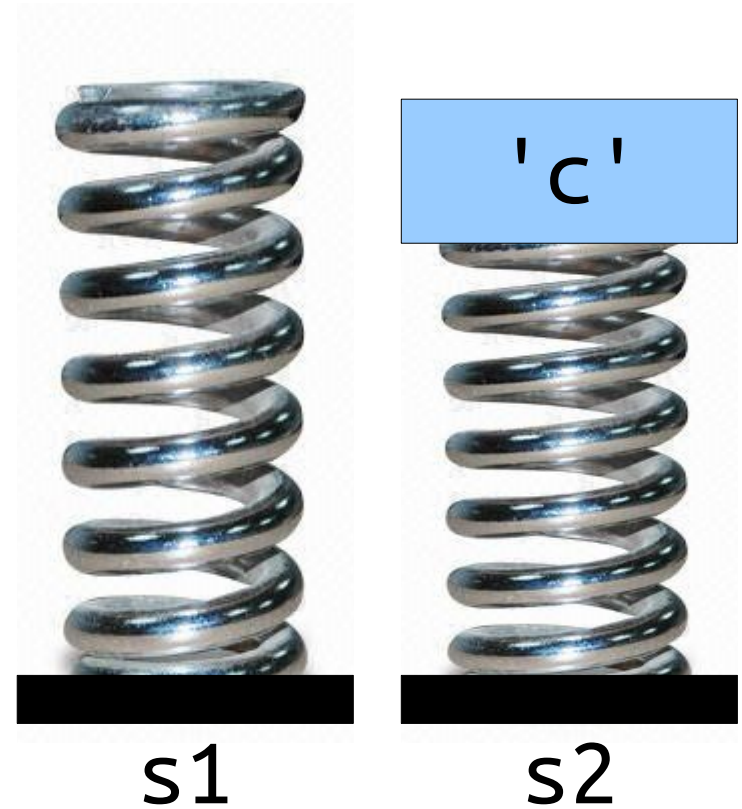
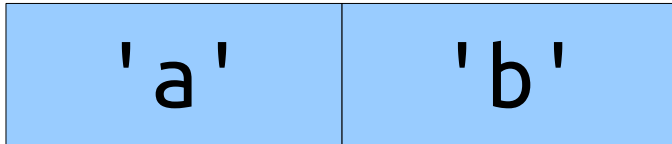
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

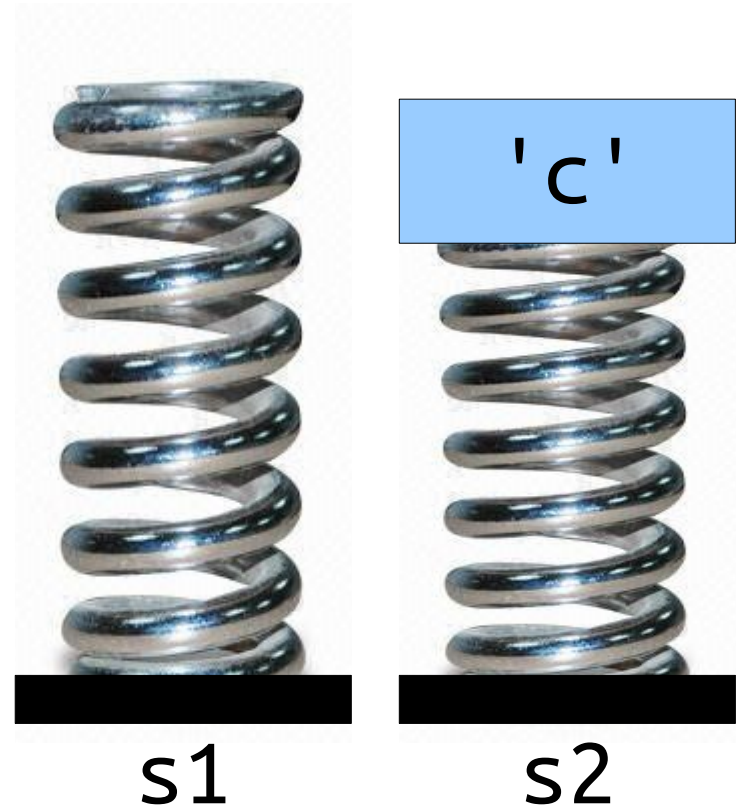
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```

'a' 'b'

'c'



s1



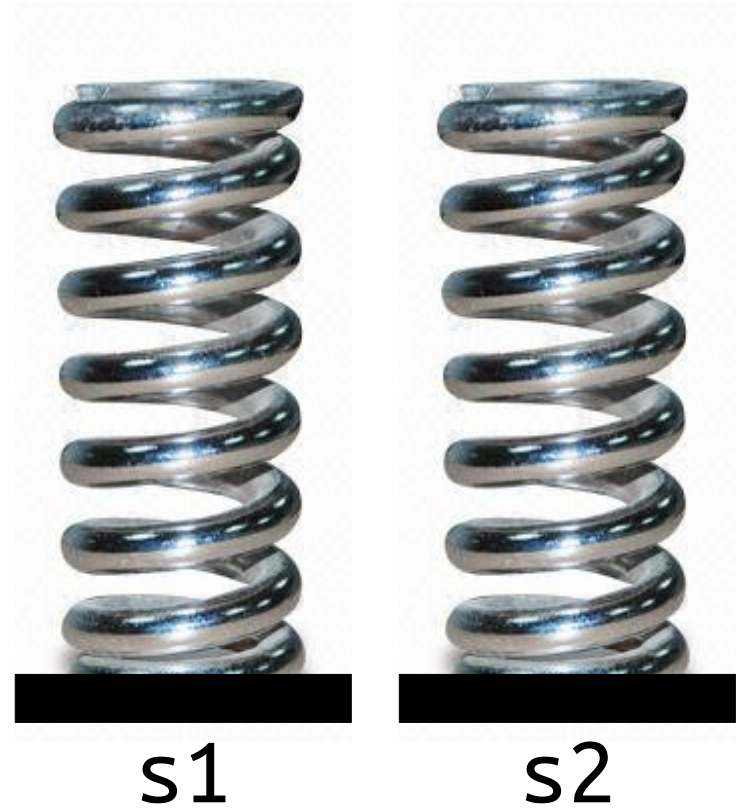
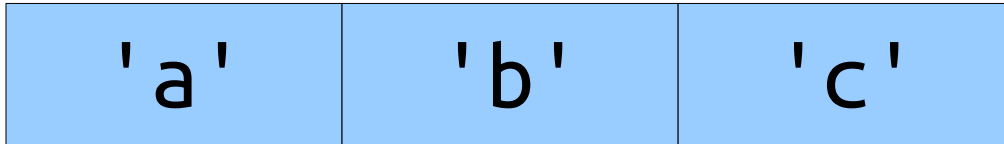
s2



# Stack

What does this code print?

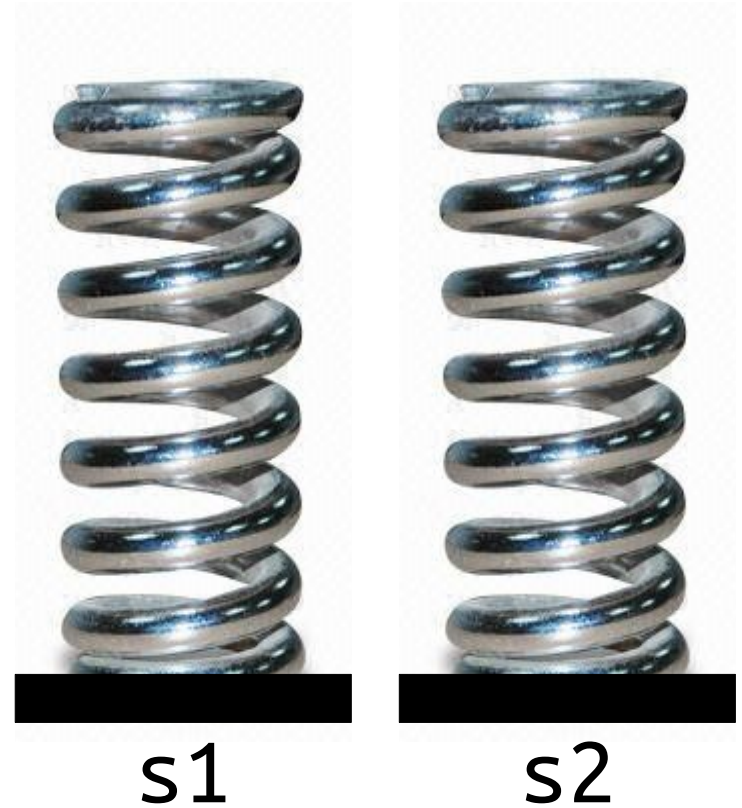
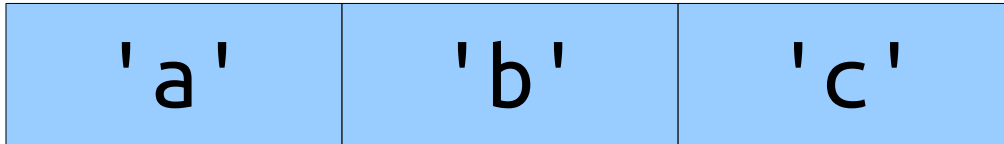
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

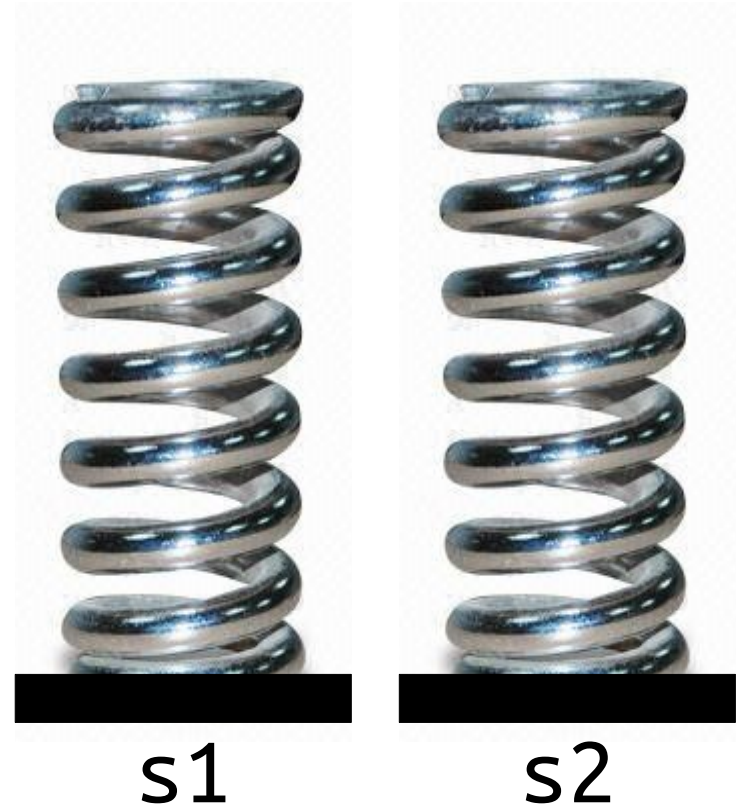
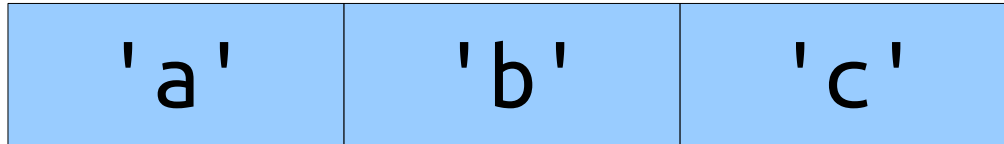
```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

What does this code print?

```
Stack<char> s1, s2;  
s1.push('a');  
s1.push('b');  
s1.push('c');  
  
while (!s1.isEmpty()) {  
    s2.push(s1.pop());  
}  
  
while (!s2.isEmpty()) {  
    cout << s2.pop() << endl;  
}
```



# Stack

- Technically speaking, anything you can do with a Stack you can also do with a Vector.
- So why do we have the Stack type as well?
  - **Clarity:** Many problems can be modeled elegantly using a stack. Representing those stacks in code with a Stack makes the code easier to understand.
  - **Error-Prevention:** The Stack has fewer operations than a Vector. If you're trying to model a stack, this automatically eliminates a large class of errors.
  - **Efficiency:** Stacks can be slightly faster than Vectors because they don't need to support as many operations. (More on that later in the quarter.)

An Application: ***Balanced Parentheses***

# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }  
^
```

# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^





# Balancing Parentheses

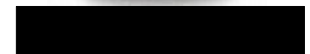
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

^



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo(^) { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo(^) { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo(^) { if (x * (y + z[1]) < 137) { x = 1; } }
```





# Balancing Parentheses

```
int foo(^) { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo(^) { if (x * (y + z[1]) < 137) { x = 1; } }
```



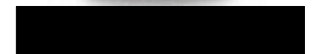
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```





# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



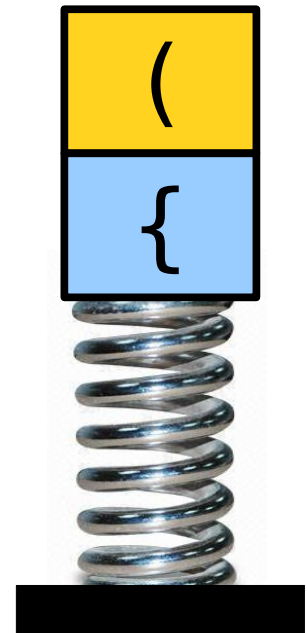
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



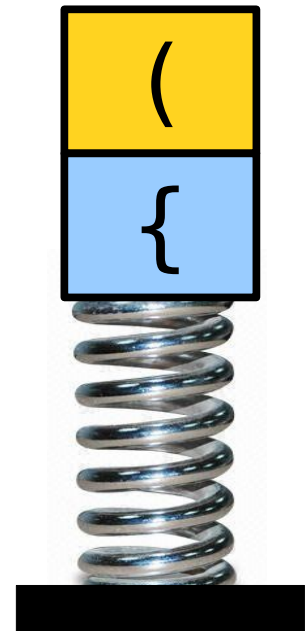
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



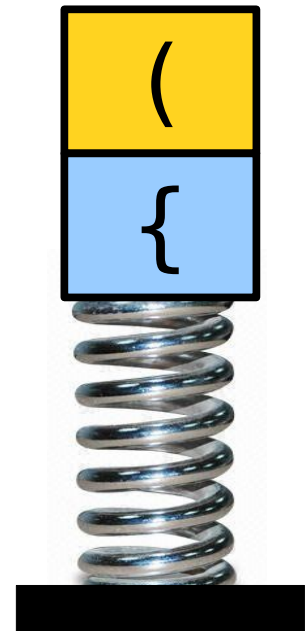
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



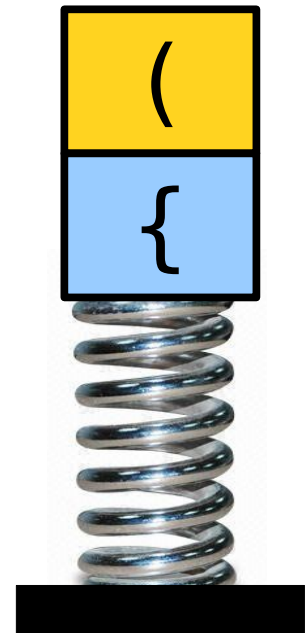
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



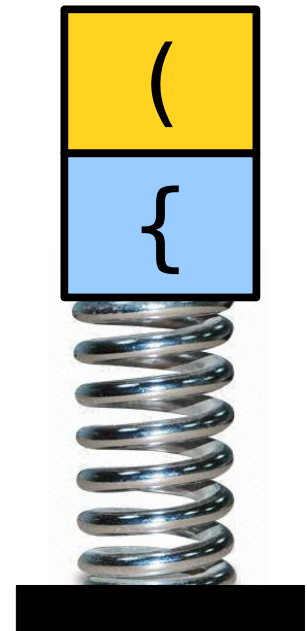
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



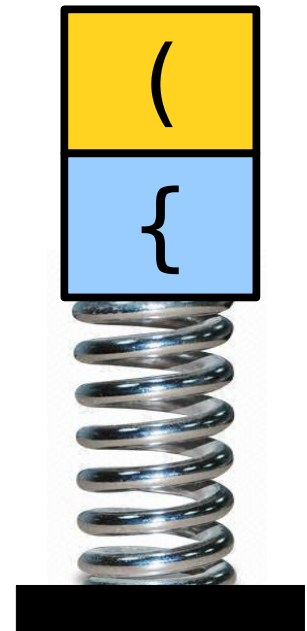
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

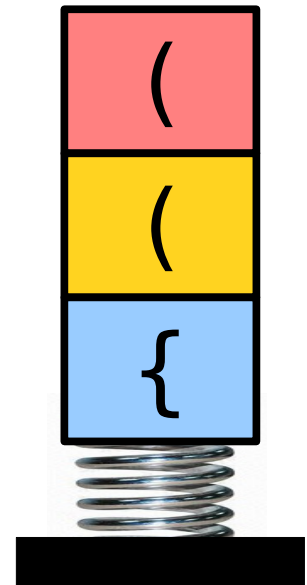
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```





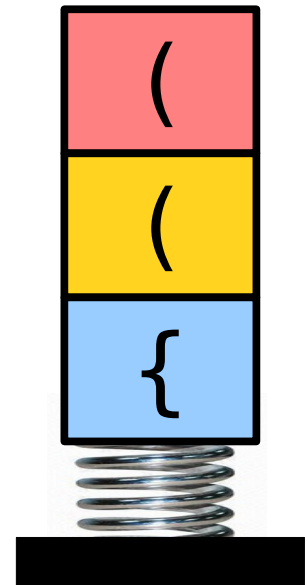
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



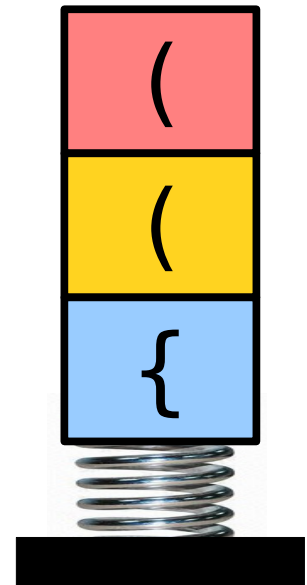
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



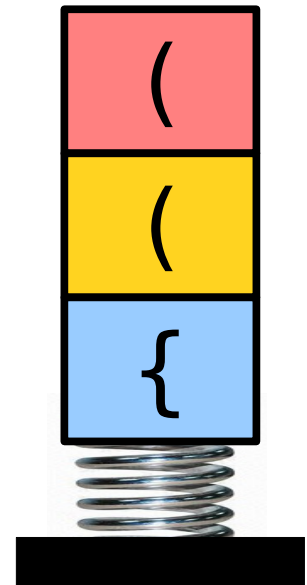
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



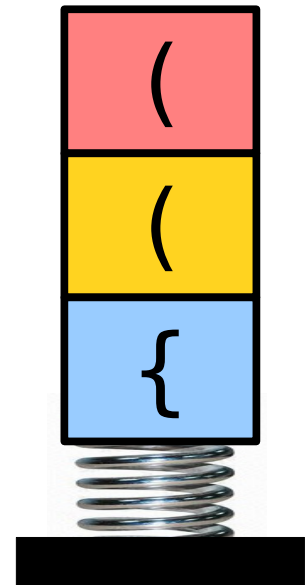
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



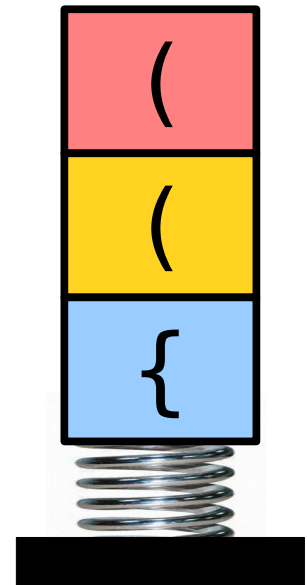
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



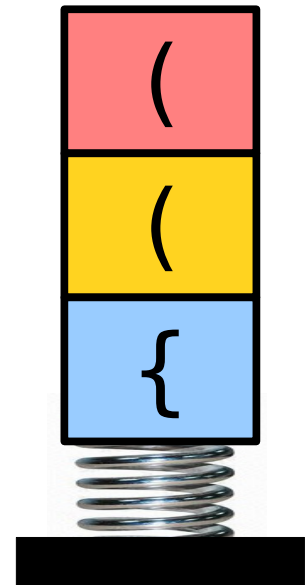
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



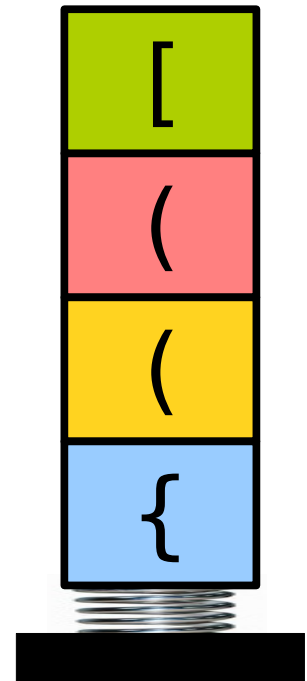
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

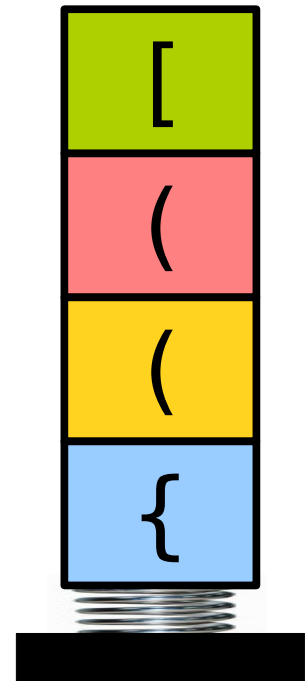
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```





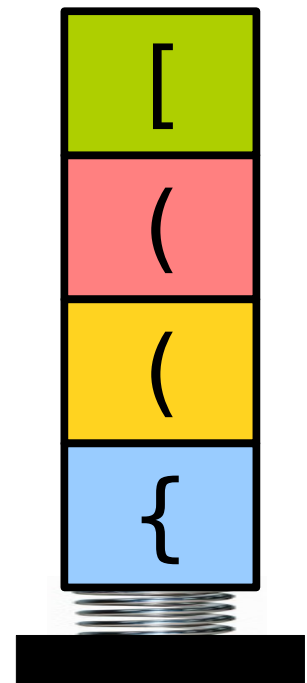
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



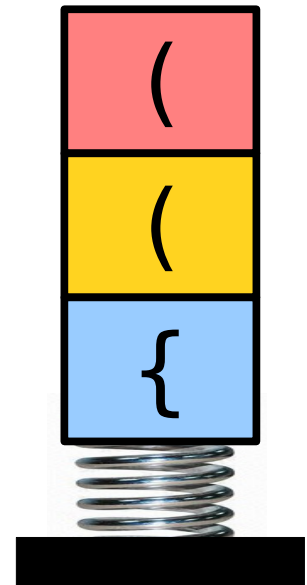
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



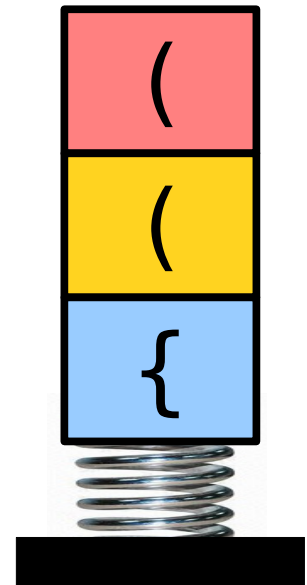
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



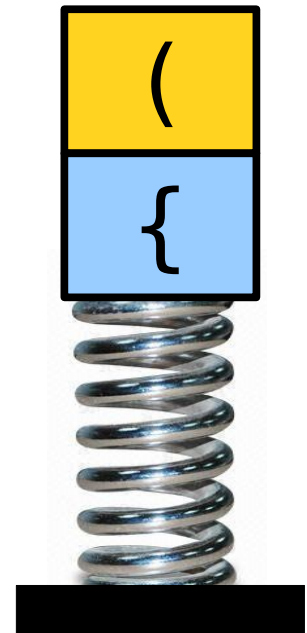
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



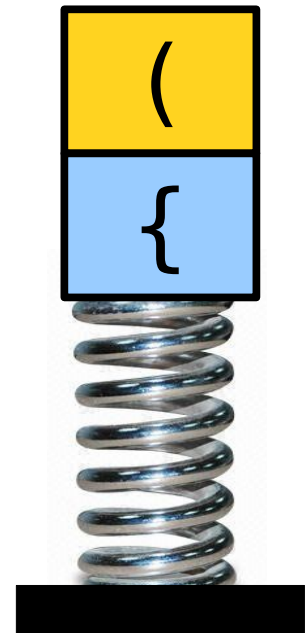
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



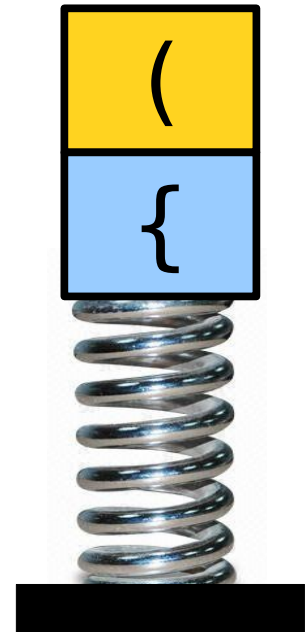
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



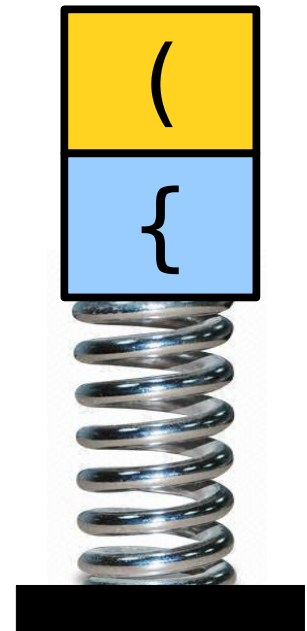
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

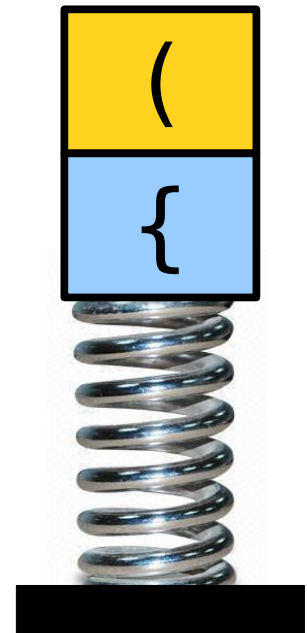
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```





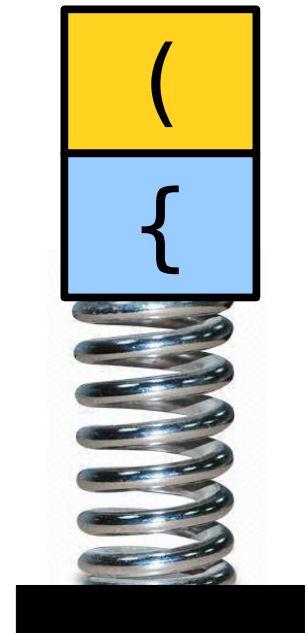
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



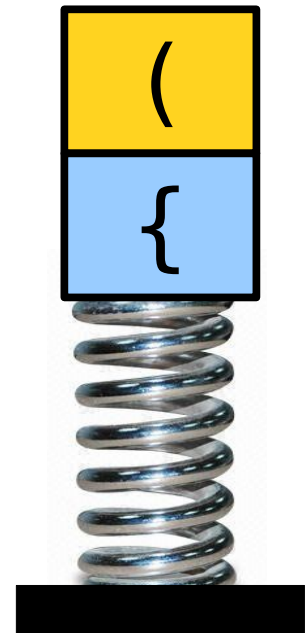
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



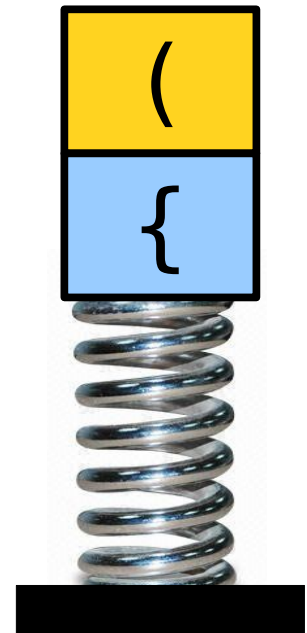
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



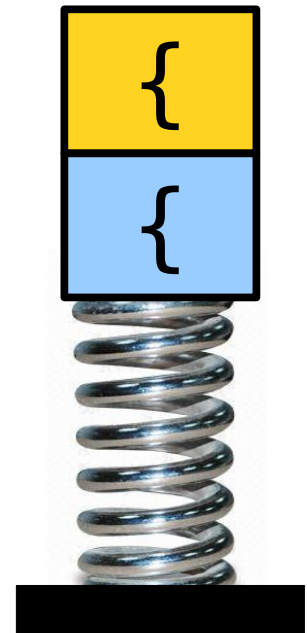
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

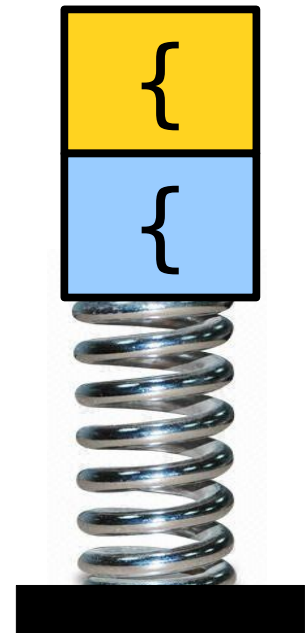
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```





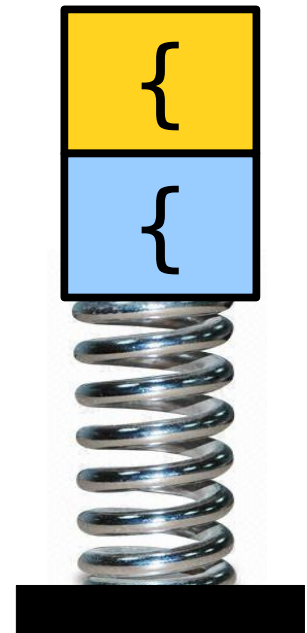
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



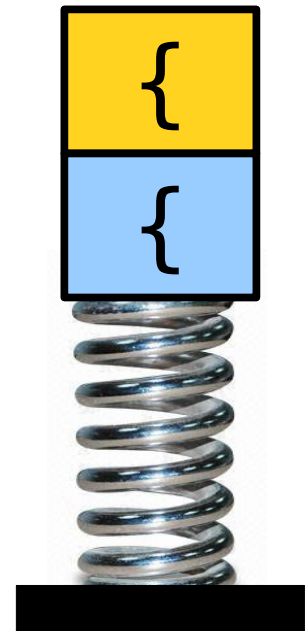
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



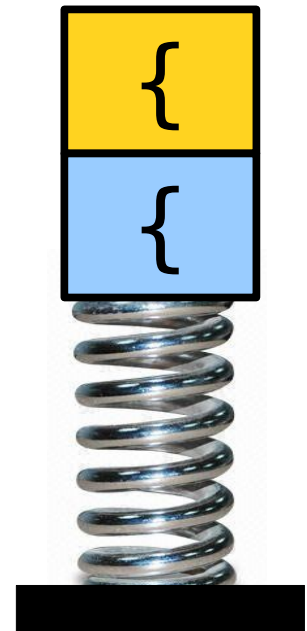
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



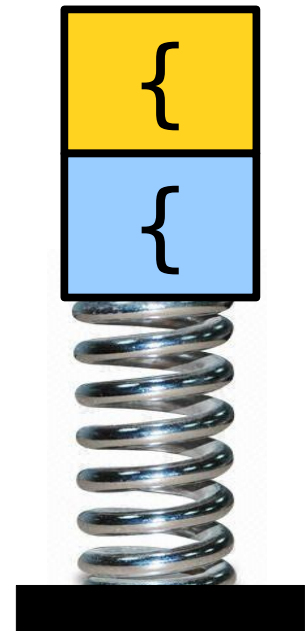
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



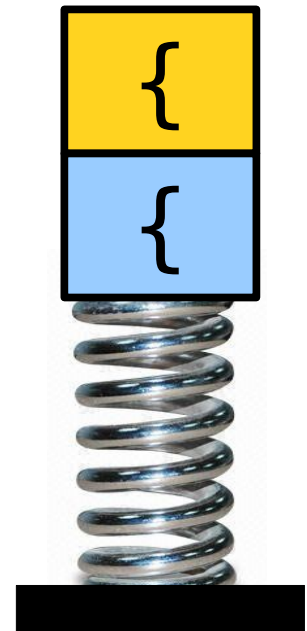
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



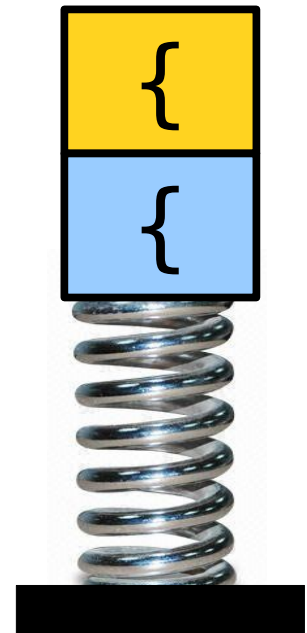
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



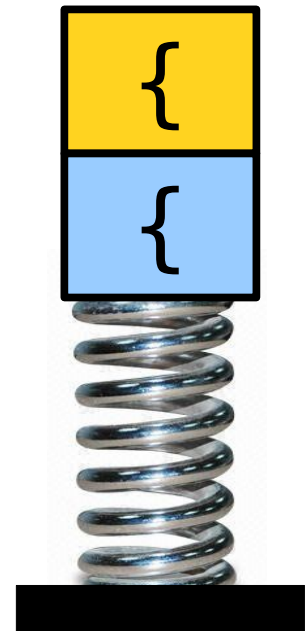
# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

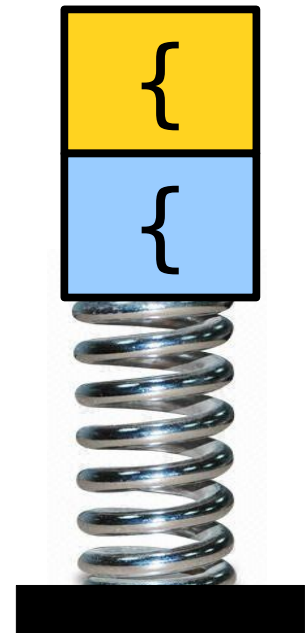
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```





# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }  
^
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } } ^
```



# Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```



# Balancing Parentheses

( [ ) ]





# Balancing Parentheses

( [ ) ]  
^



# Balancing Parentheses

( [ ) ]  
^



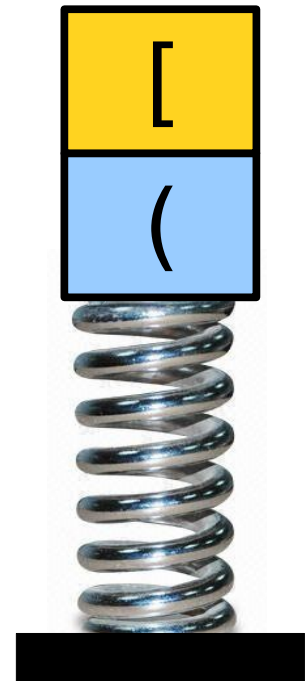
# Balancing Parentheses

( [ ) ]  
^



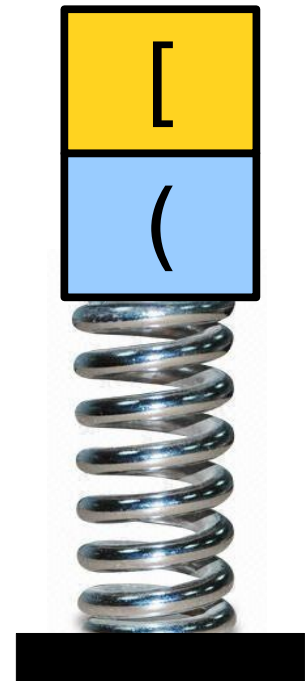
# Balancing Parentheses

( [ ) ]  
^



# Balancing Parentheses

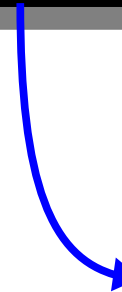
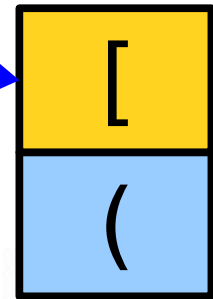
( [ ) ]  
          ^



# Balancing Parentheses

( [ ) ]  
          ^

Oops! Wrong type of  
parenthesis here.



# Balancing Parentheses

((



# Balancing Parentheses

( ( ^





# Balancing Parentheses

( (



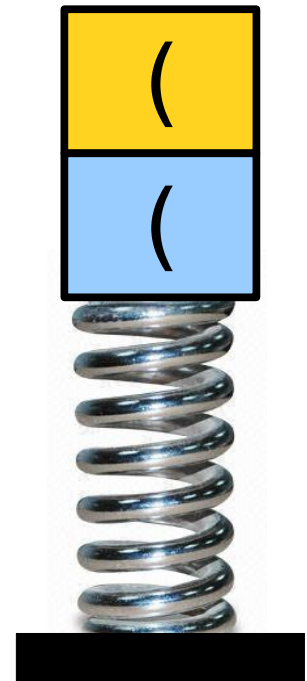
# Balancing Parentheses

( (



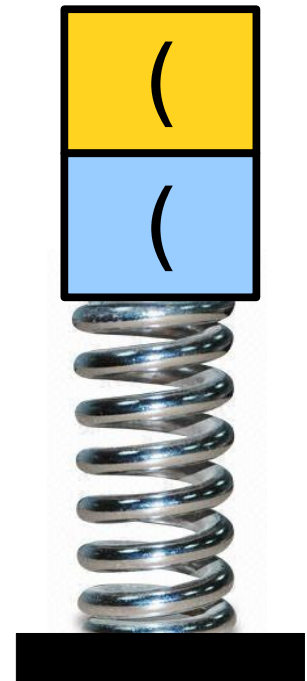
# Balancing Parentheses

( (



# Balancing Parentheses

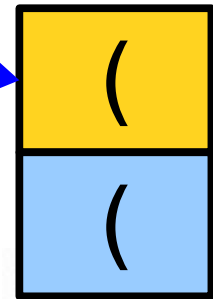
((



# Balancing Parentheses

( (

Oops! We never  
matched this.



# Balancing Parentheses

)



# Balancing Parentheses

)  
^



# Balancing Parentheses

Oops! There's  
nothing on the stack  
to match.

)  
^





# Our Algorithm

- For each character:
  - If it's an open parenthesis or brace, push it onto the stack.
  - If it's a close parenthesis or brace:
    - If the stack is empty, report an error.
    - If the character doesn't pair with the character on top of the stack, report an error.
- At the end, return whether the stack is empty (nothing was left unmatched).

# More Stack Applications

- Stacks show up all the time in *parsing*, recovering the structure in a piece of text.
  - Often used in natural language processing; take CS224N for details!
  - Used all the time in compilers – take CS143 for details!
  - There’s a deep theorem that says that many structures appearing in natural language are perfectly modeled by operations on stacks; come talk to me after class if you’re curious!
- They’re also used as building blocks in larger algorithms for doing things like
  - making sure a city’s road networks are navigable (finding *strongly connected components*; take CS161 for details!) and
  - searching for the best solution to a problem – stay tuned!

**Time-Out for Announcements!**

# Assignment 1

- Assignment 1 is due this Friday at 10:30AM, half an hour before the start of class.
- Have questions?
  - Call into the LaIR!
  - Ask on EdStem!
  - Email your section leader!

# Discussion Sections

- Discussion sections have started! You should have received an email with your section time and section leader's name.
- Don't have a section? You can sign up for any open section by visiting

**<https://cs198.stanford.edu/>**

logging in via “CS106 Sections Login,” and picking a section of your choice.

# Upcoming Career Fair

The **Computer Forum Winter Career Fair** is next week - January 19! Hosted virtually on Career Fair Plus. You can sign up for sessions starting January 12 at 8am PT! Stanford students only.

## **Candidate Checklist**

## **How to Book Meetings**

Best Practices:

- Be courteous of our employers time and your fellow students and only sign up for what you will be able to attend
- If you are interested in a company, so are other students, so please only sign up for one session per company
- Have your video on during your 1:1 sessions

We will have many VCs attending the fair so make sure to check out their booth to see which of their portfolio companies will be hosting sessions!

Important: Please note the Stanford Computer Forum policies regarding no-shows here. If you do not show up for a session you signed up for, your participation in future Computer Forum events may be revoked. By signing up and not showing up, you are taking away a spot from another student.

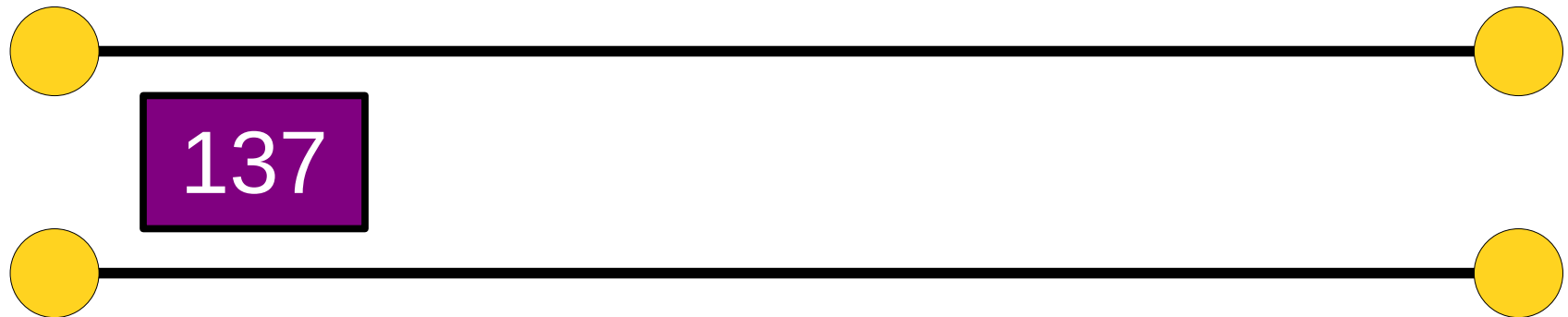
```
lecture.pop();
```

Queue



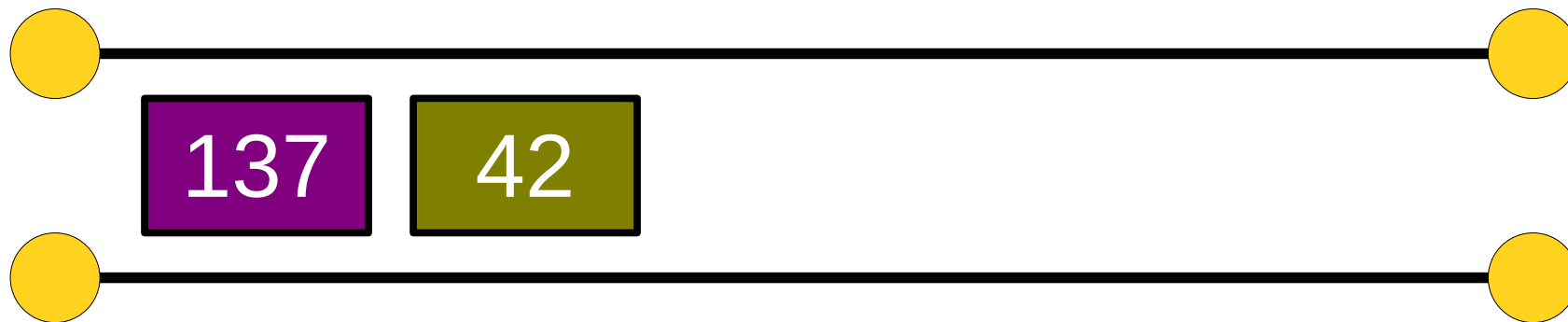
# Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.



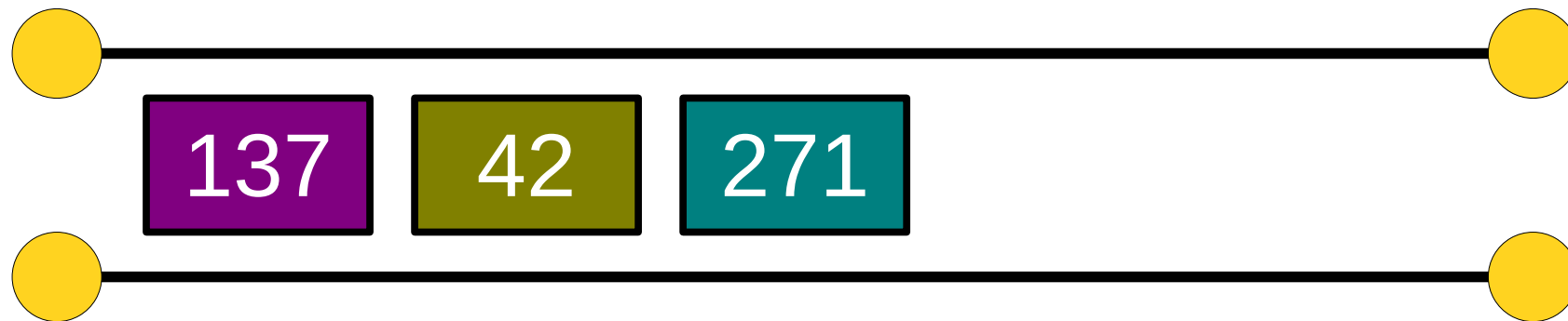
# Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.



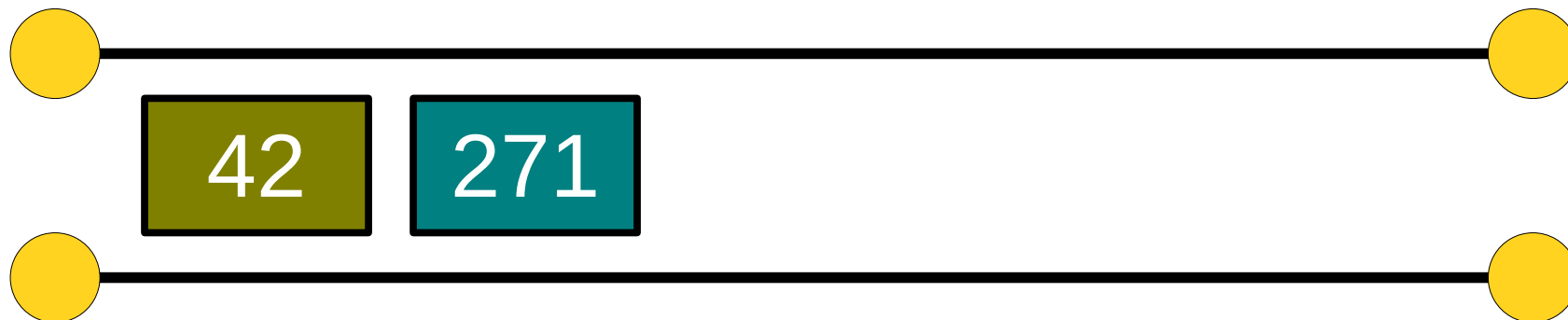
# Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.



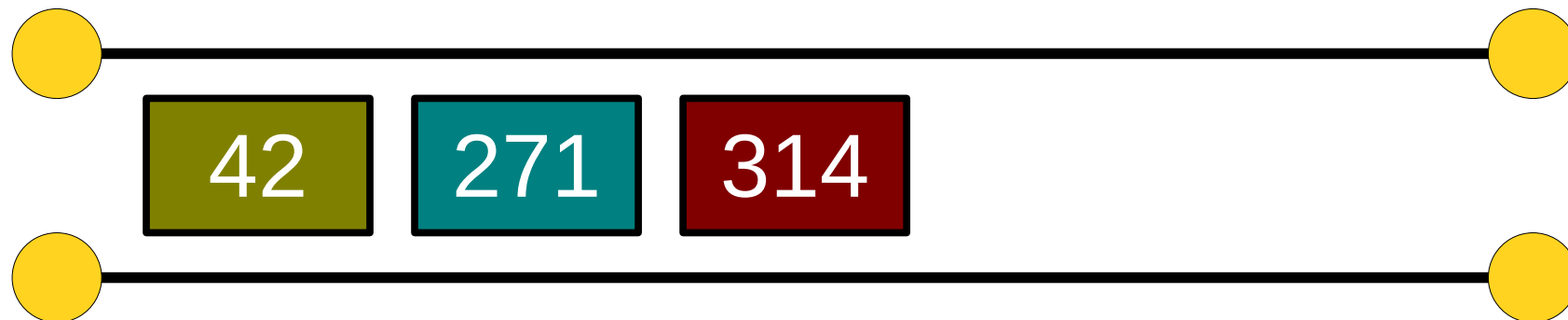
# Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.



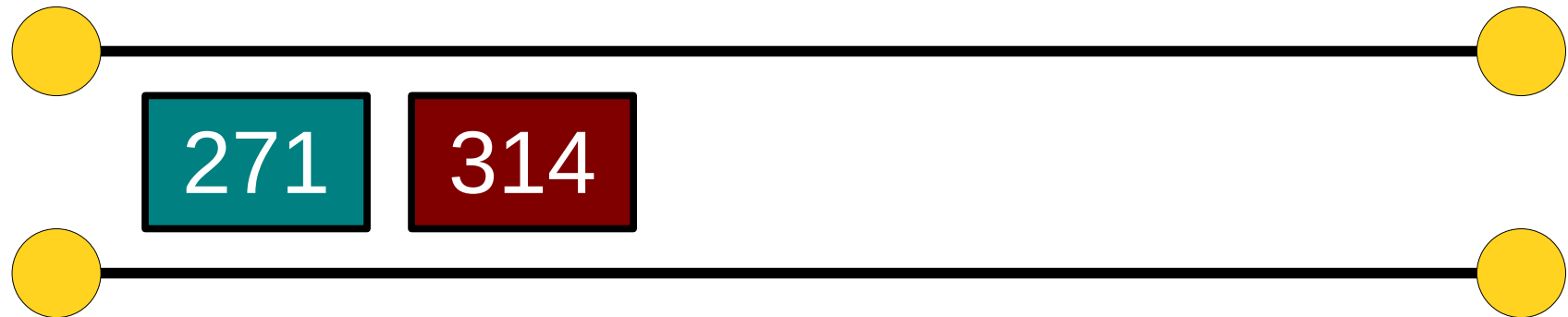
# Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.



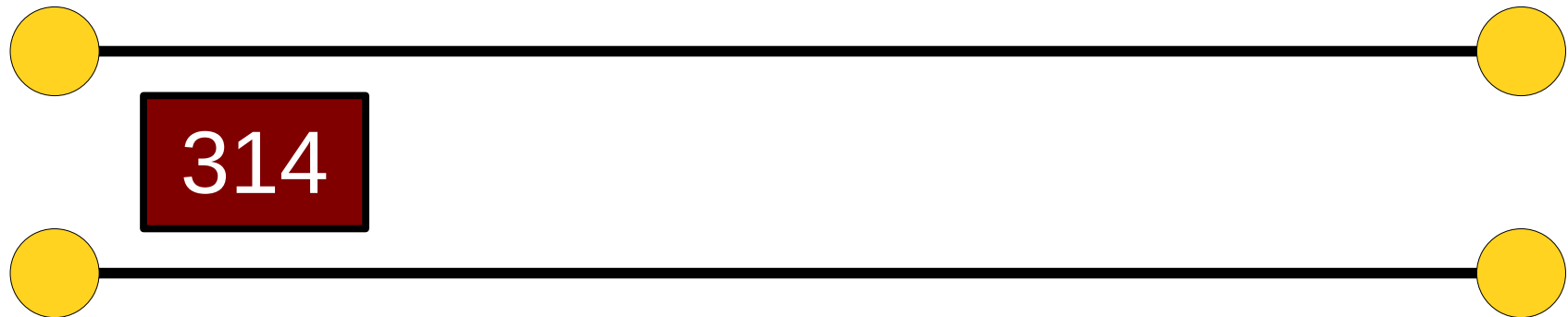
# Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.



# Queue

- A **Queue** is a data structure representing a waiting line.
- Objects can be **enqueued** to the back of the line or **dequeued** from the front of the line.
- No other objects in the queue are visible.
- Example: A checkout counter.



# Queue

- What does this code print?

```
Queue<char> q1, q2;
q1.enqueue('a');
q1.enqueue('b');
q1.enqueue('c');

while (!q1.isEmpty()) {
    q2.enqueue(q1.dequeue());
}

while (!q2.isEmpty()) {
    cout << q2.dequeue() << endl;
}
```

Formulate a hypothesis,  
but ***don't post anything  
in chat just yet.***



# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');  
  
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

Now, *private chat me your best guess*. Not sure? Just answer “??”

# Queue

- What does this code print?

```
Queue<char> q1, q2;
```

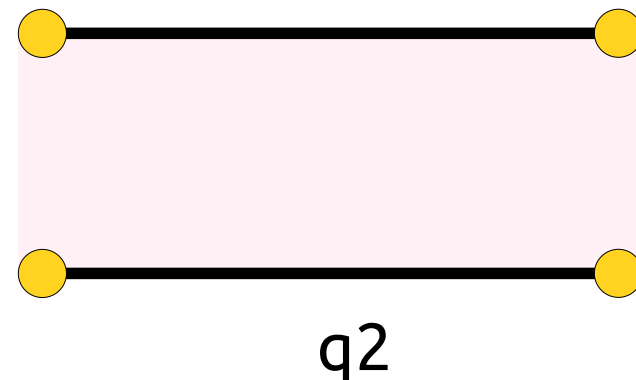
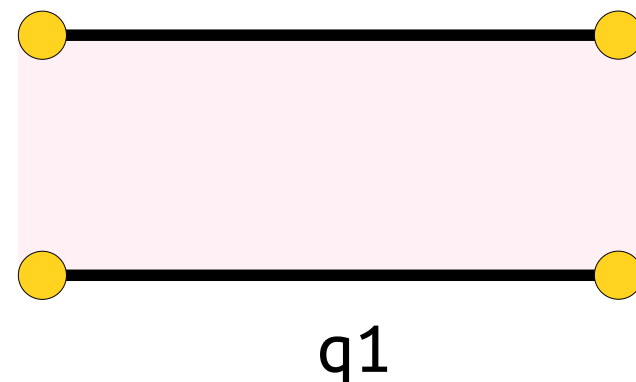
```
q1.enqueue('a');
```

```
q1.enqueue('b');
```

```
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



# Queue

- What does this code print?

```
Queue<char> q1, q2;
```

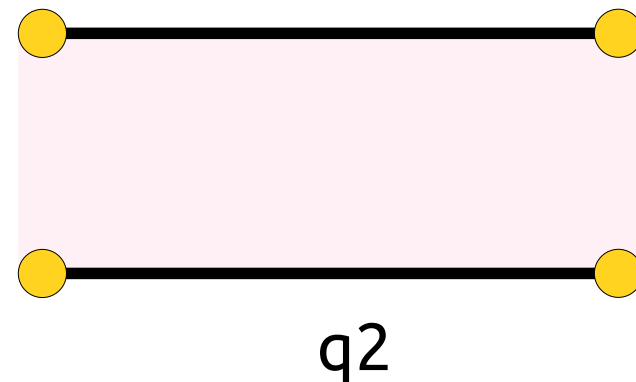
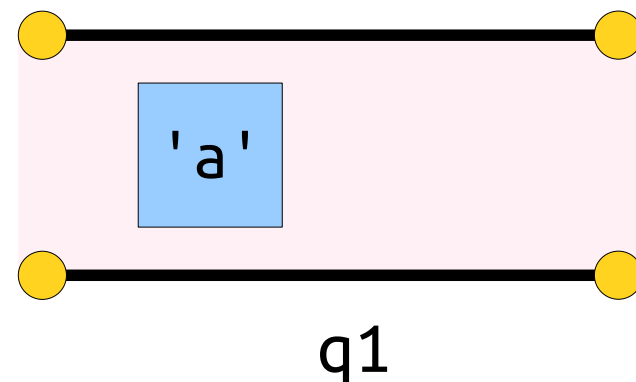
```
q1.enqueue('a');
```

```
q1.enqueue('b');
```

```
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



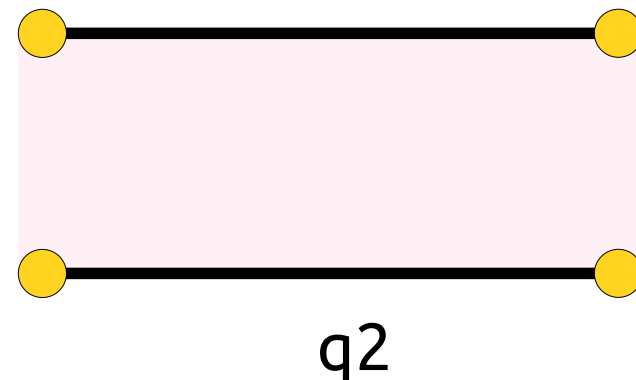
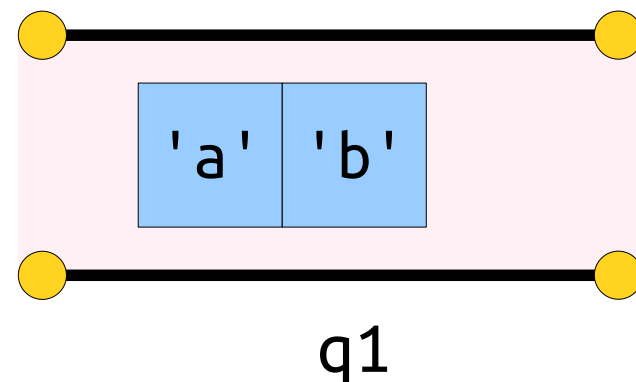
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

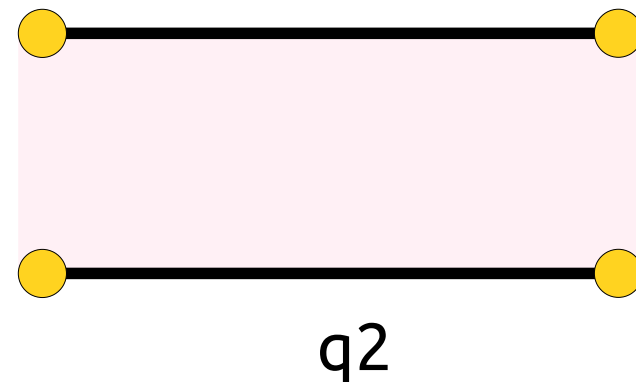
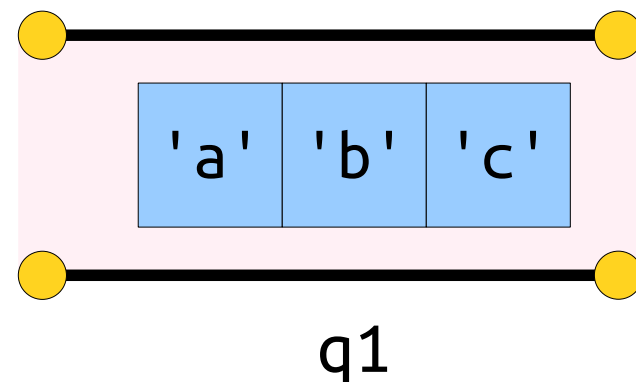
```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');  
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

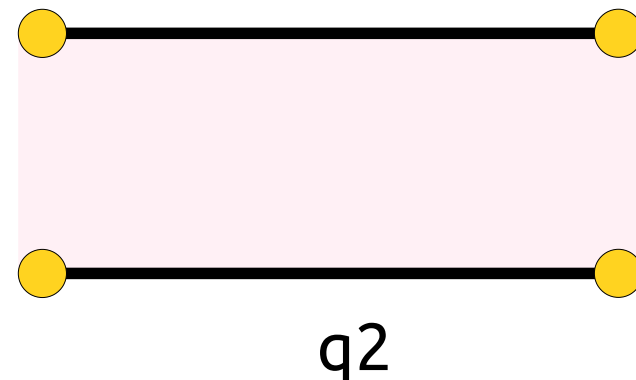
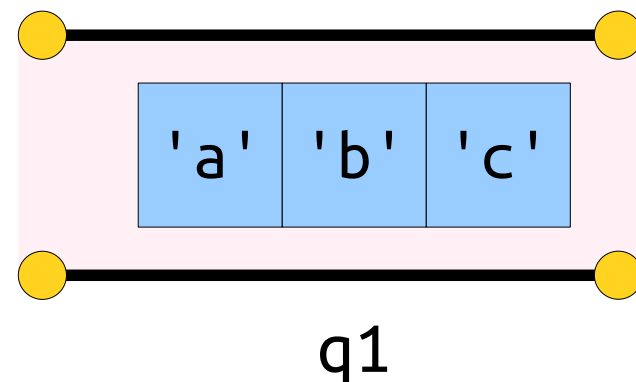


# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



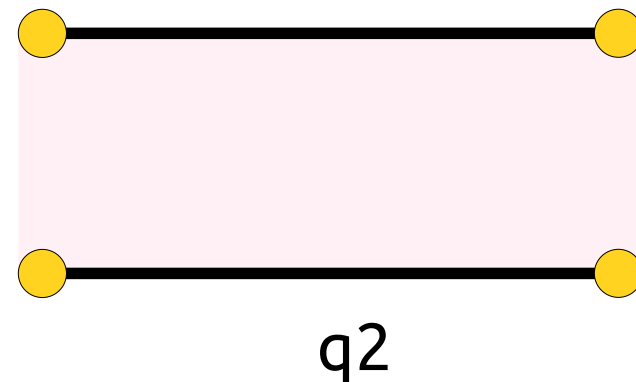
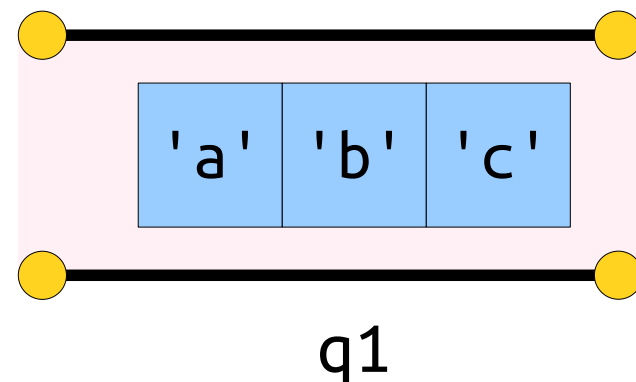
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



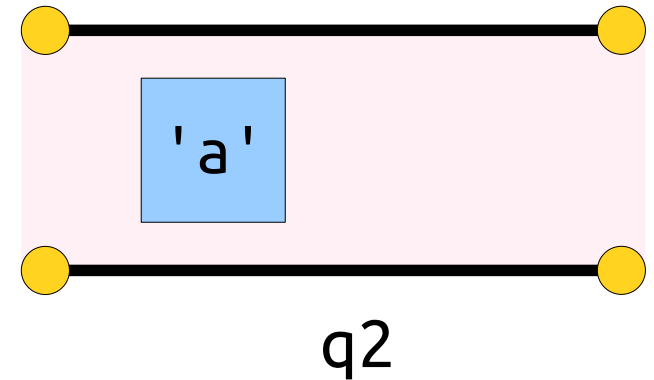
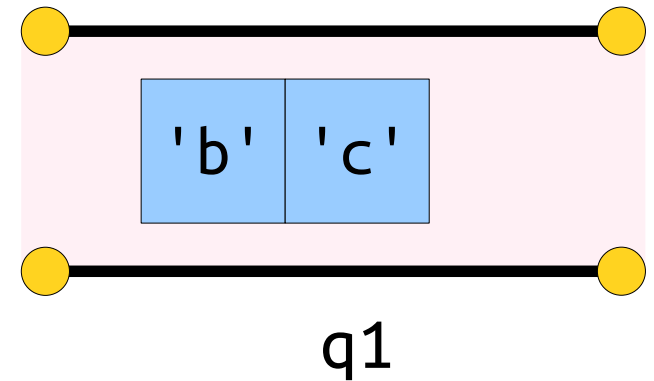
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



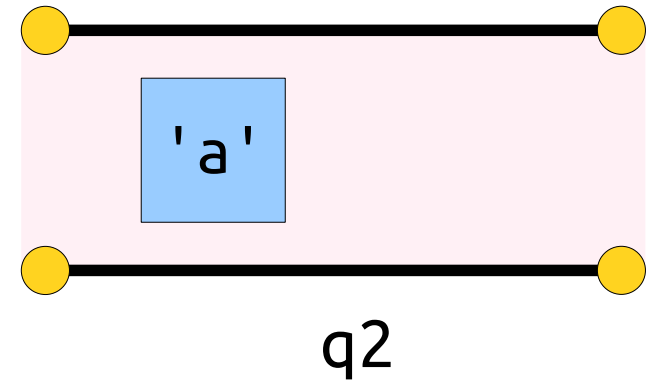
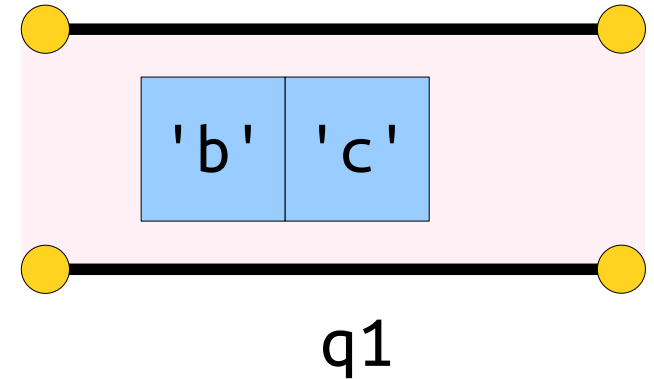


# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



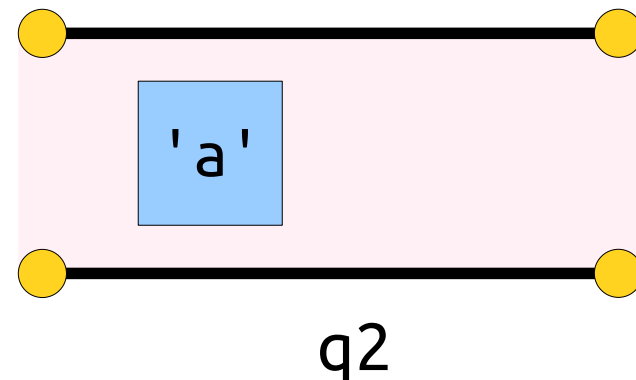
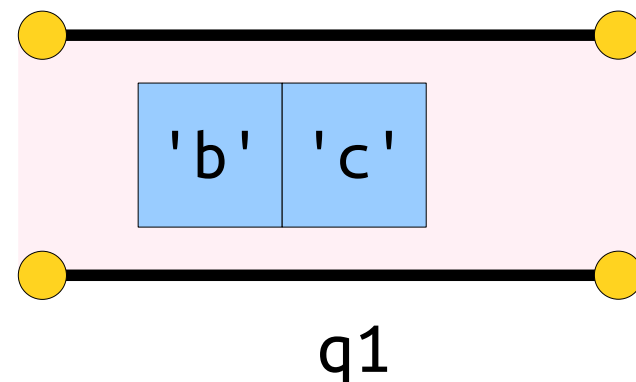
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



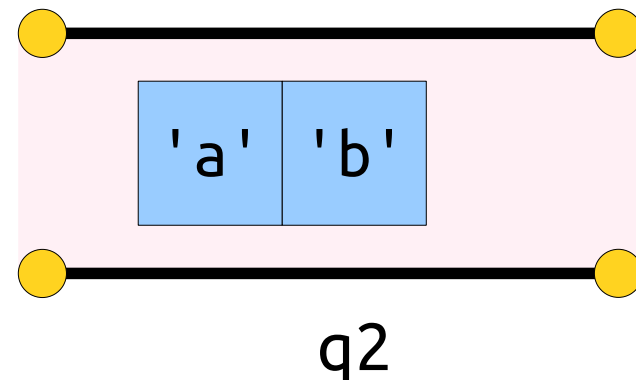
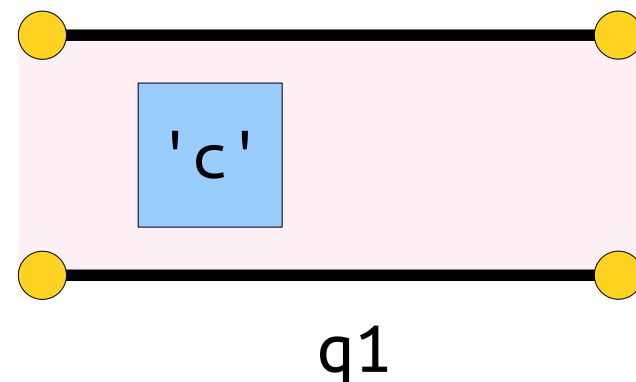
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

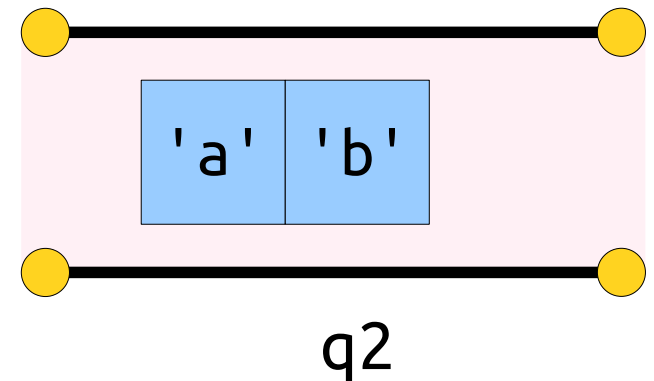
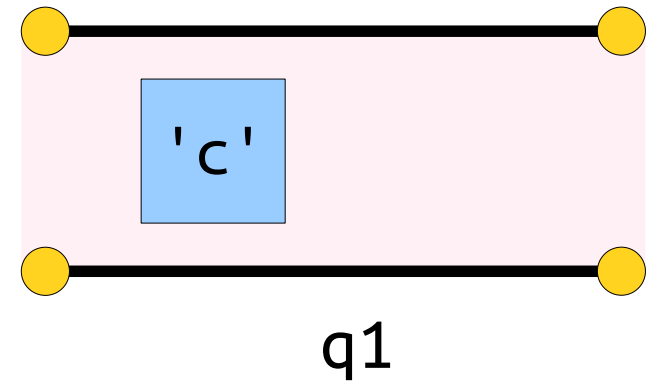


# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



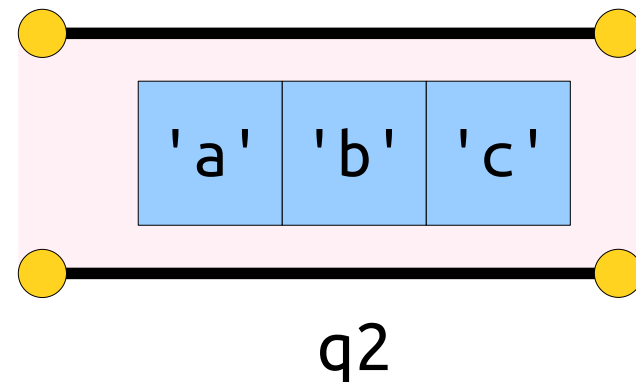
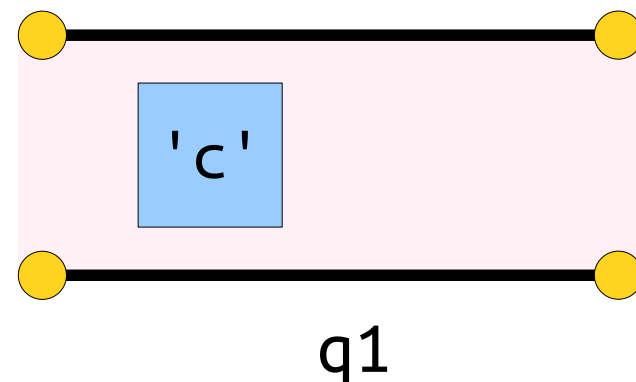
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

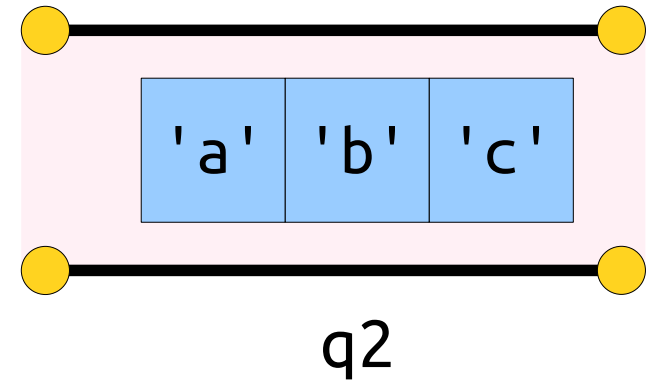
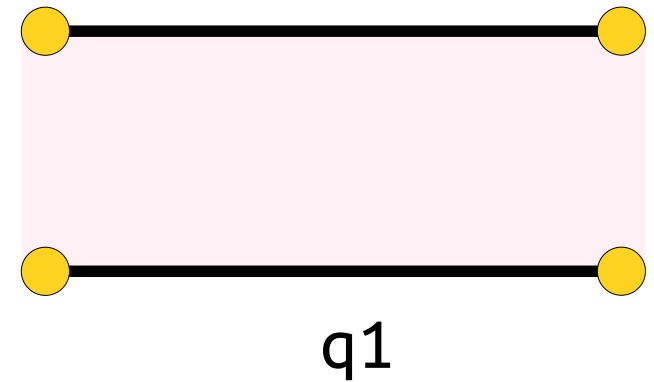


# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



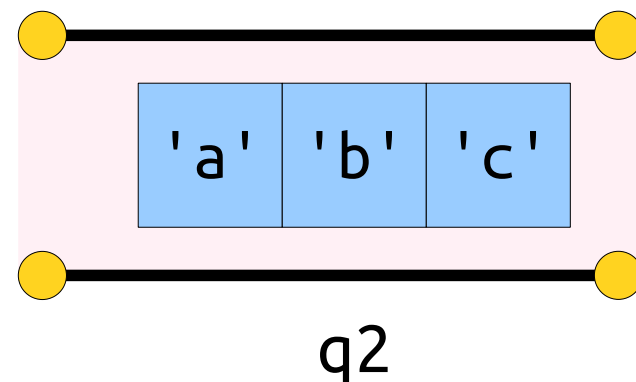
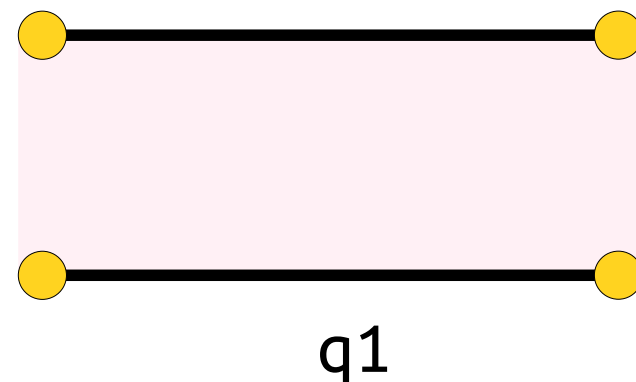
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



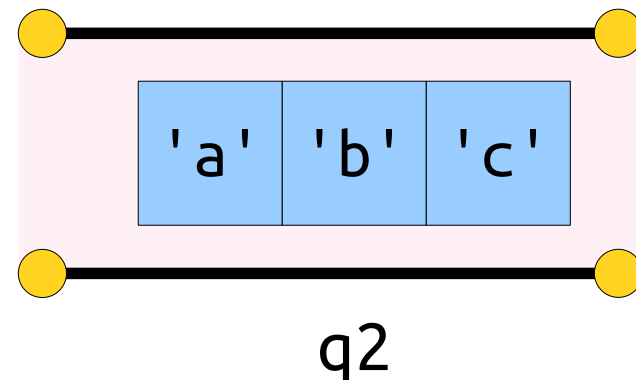
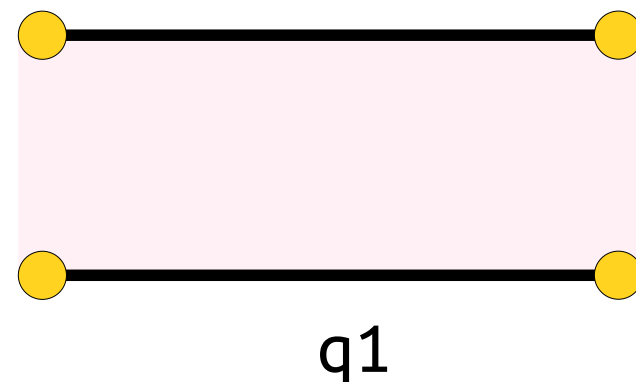
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```





# Queue

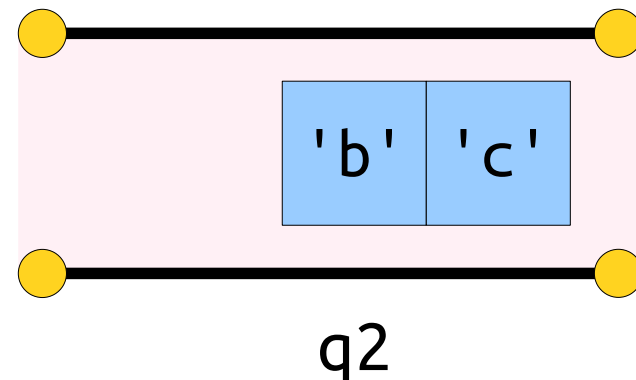
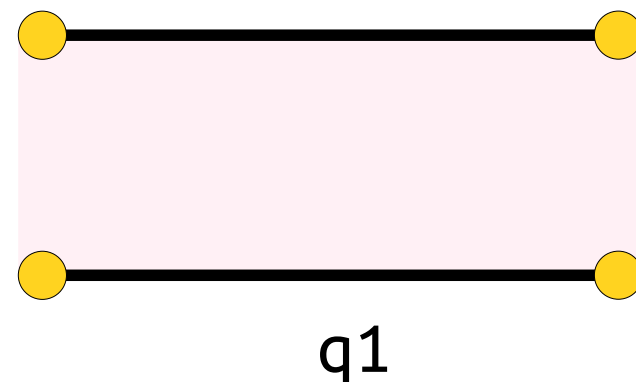
- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

'a'



# Queue

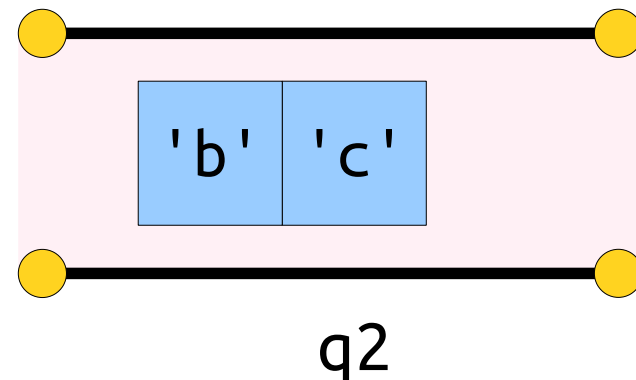
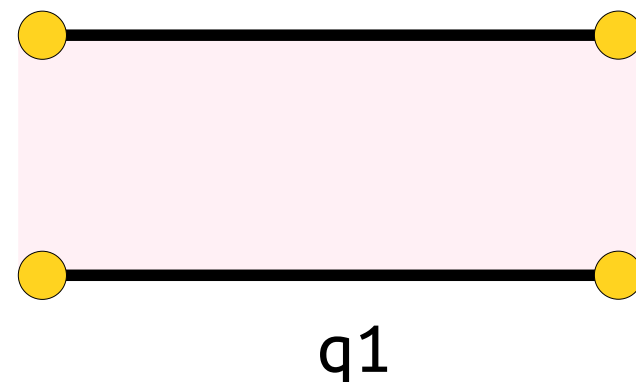
- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

'a'



# Queue

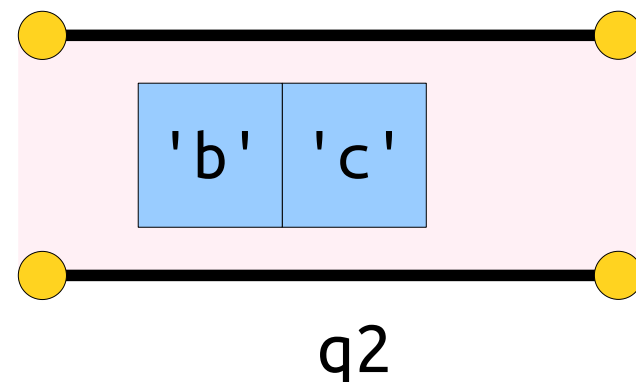
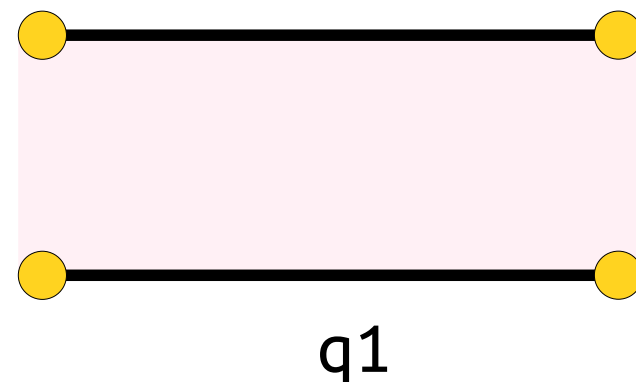
- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

'a'



# Queue

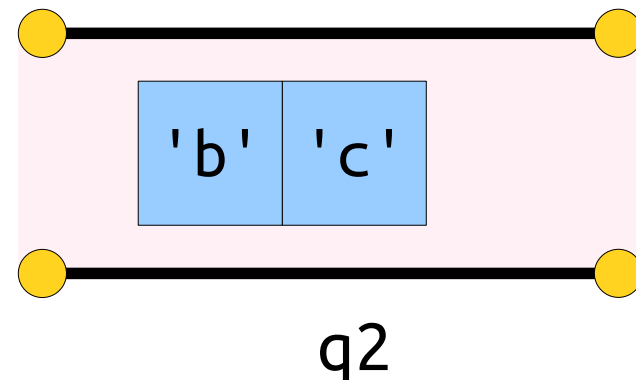
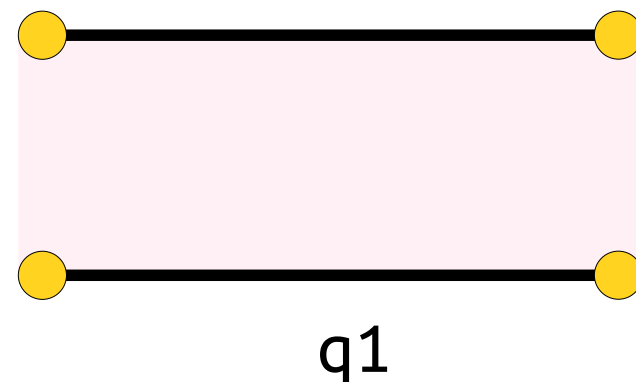
- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

'a'



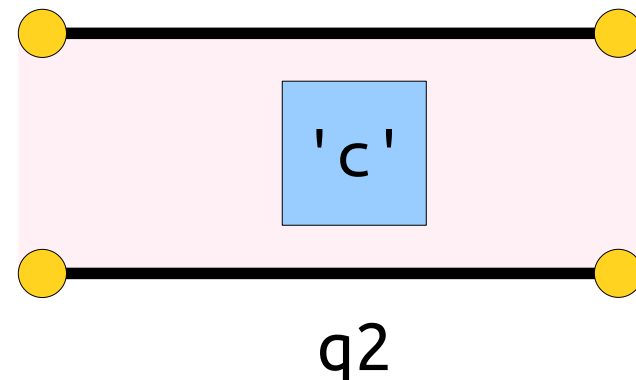
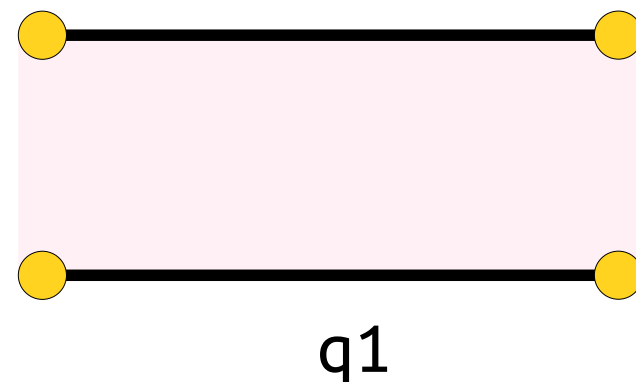
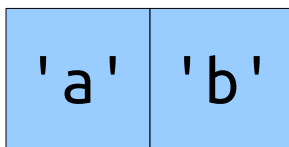
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

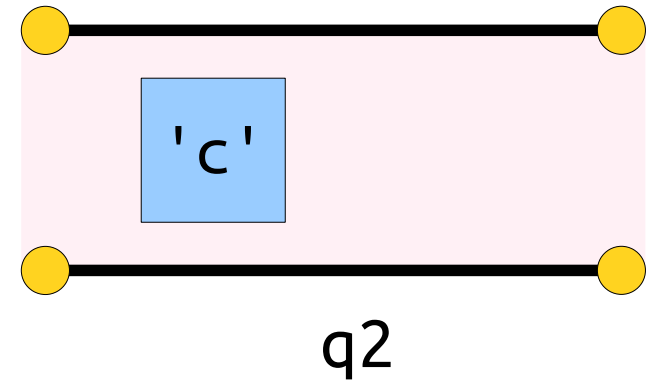
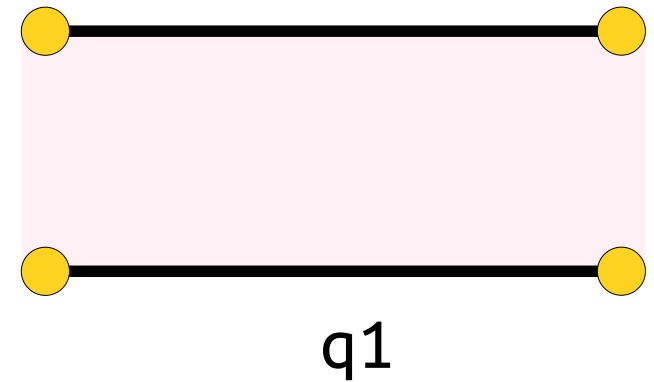
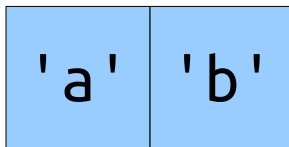
```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');  
  
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



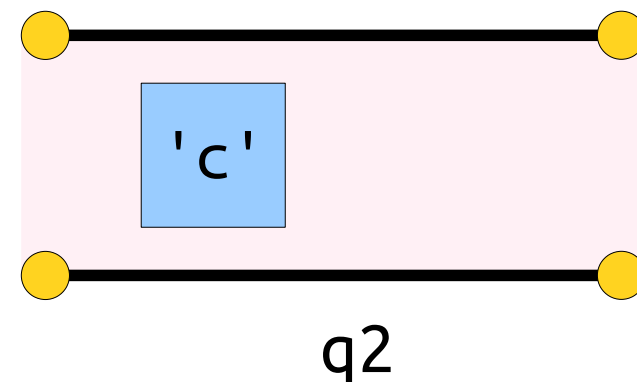
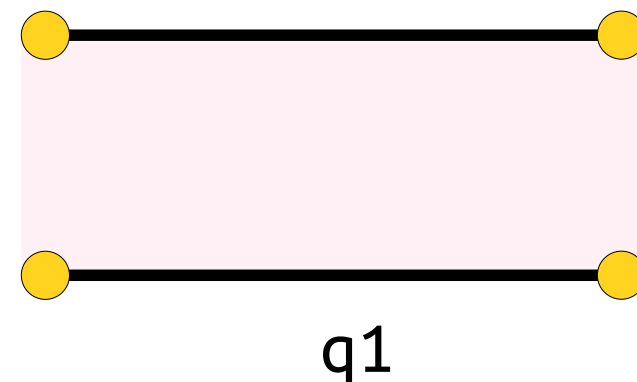
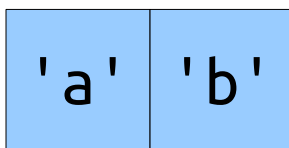
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



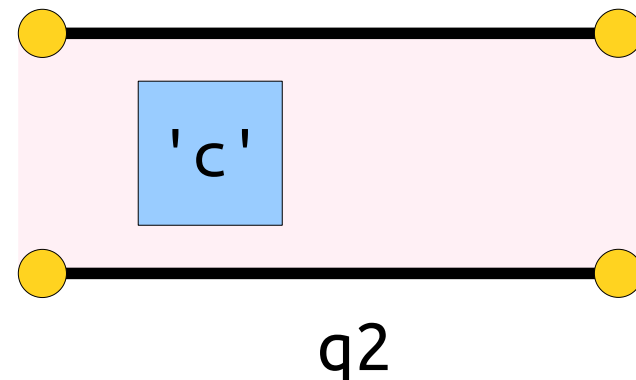
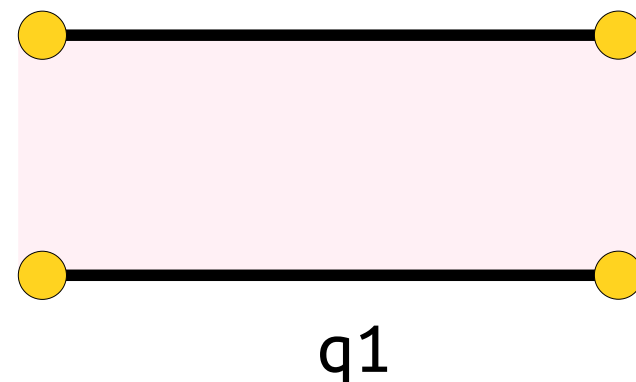
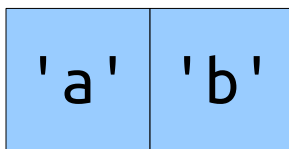
# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');
```

```
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}
```

```
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```

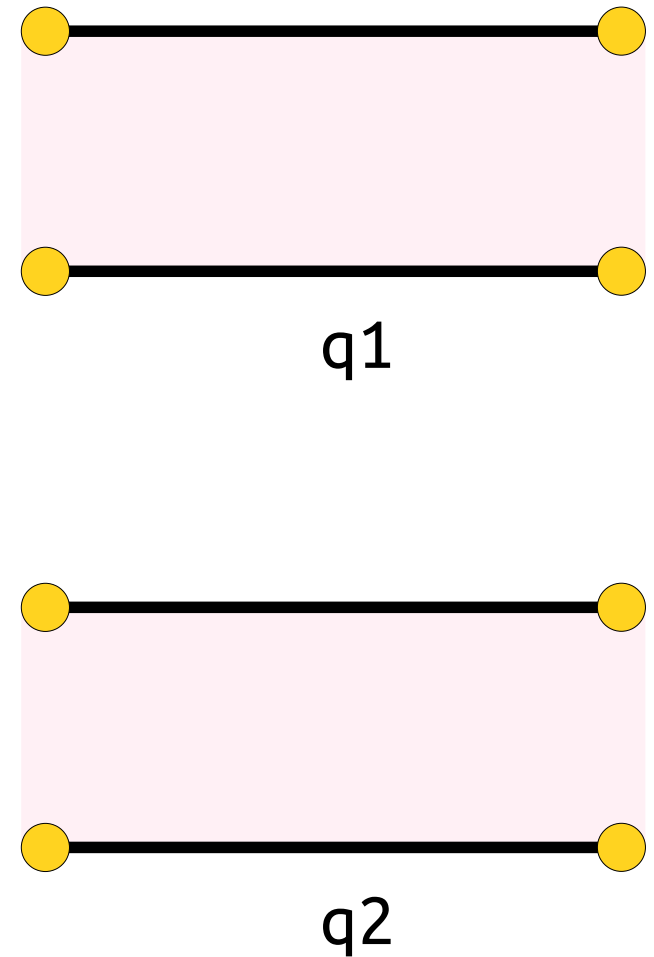
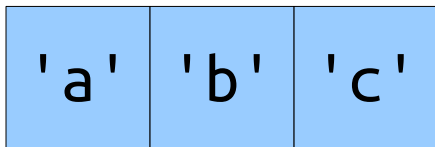




# Queue

- What does this code print?

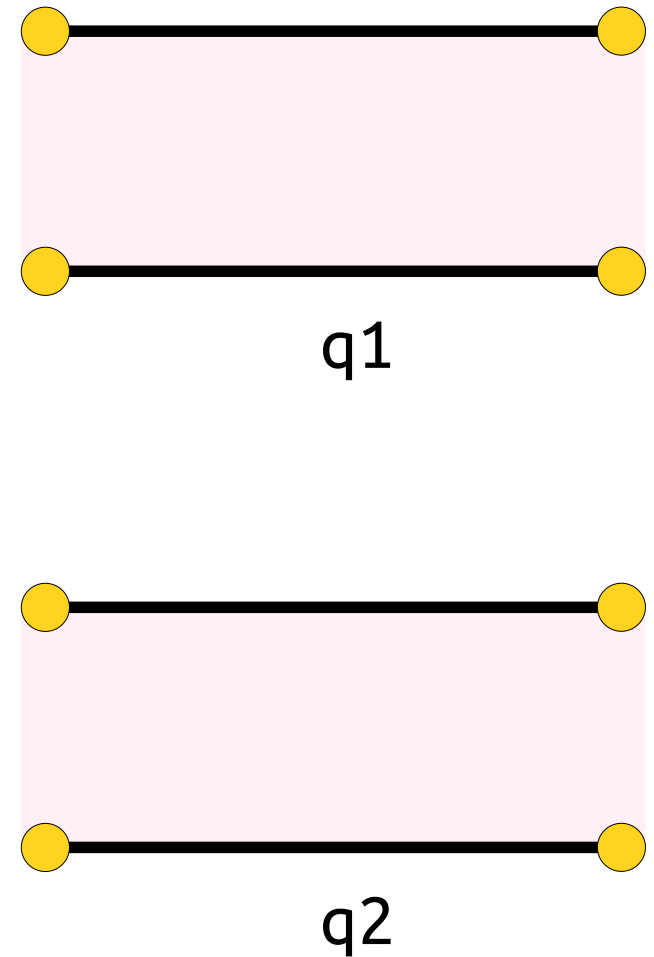
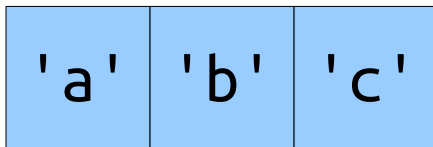
```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');  
  
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



# Queue

- What does this code print?

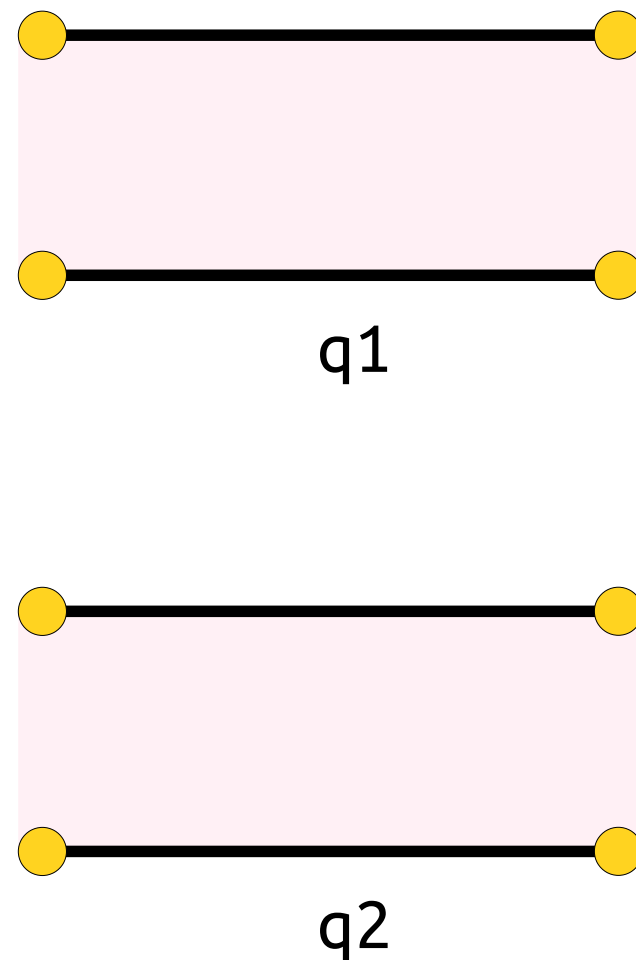
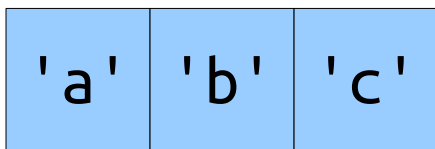
```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');  
  
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



# Queue

- What does this code print?

```
Queue<char> q1, q2;  
q1.enqueue('a');  
q1.enqueue('b');  
q1.enqueue('c');  
  
while (!q1.isEmpty()) {  
    q2.enqueue(q1.dequeue());  
}  
  
while (!q2.isEmpty()) {  
    cout << q2.dequeue() << endl;  
}
```



An Application: *Looper*

# Loopers

- A ***looper*** is a device that records sound or music, then plays it back over and over again (in a loop).
- These things are way too much fun, *especially* if you're not a very good musician.
- Let's make a simple looper using a Queue.

# Building our Looper

- Our looper will read data files like the one shown to the left.
- Each line consists of the name of a sound file to play, along with how many milliseconds to play that sound for.
- We'll store each line using the `SoundClip` type, which is defined in our C++ file.

```
G2.wav 690
G2.wav 230
Bb2.wav 230
G2.wav 460
G2.wav 460
G2.wav 460
G2.wav 230
Bb2.wav 230
G2.wav 230
F2.wav 460
```

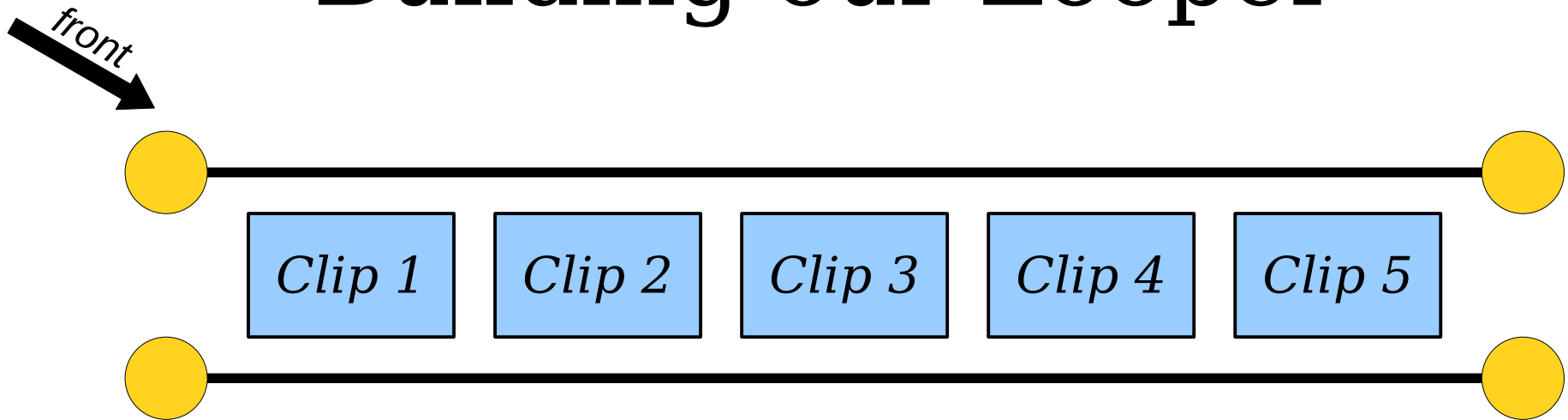
# Building our Looper

# Building our Looper

```
Queue<SoundClip> loop = loadLoop(/* ... */);
```

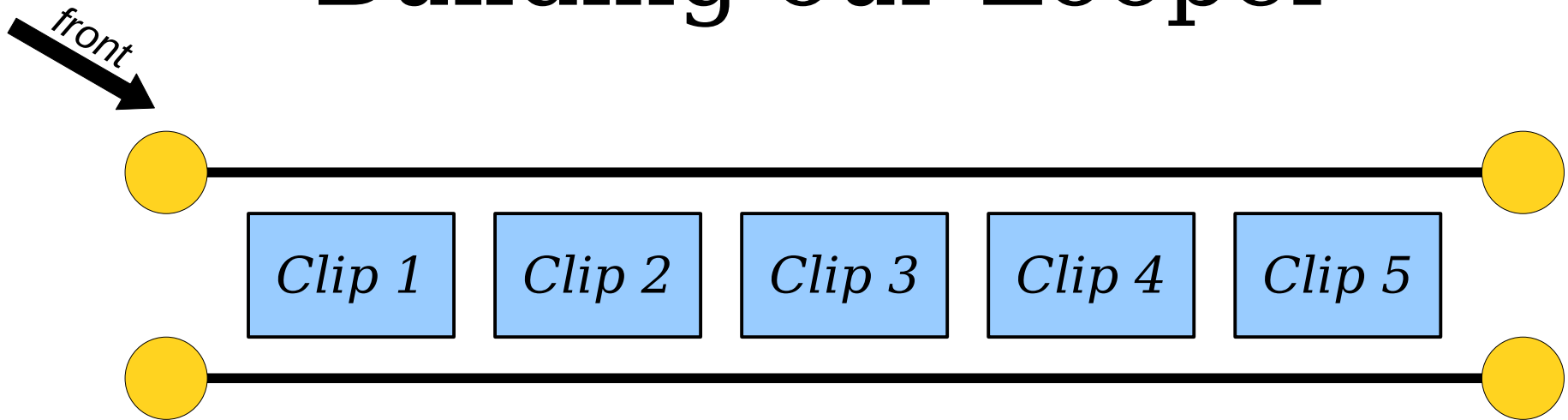


# Building our Looper



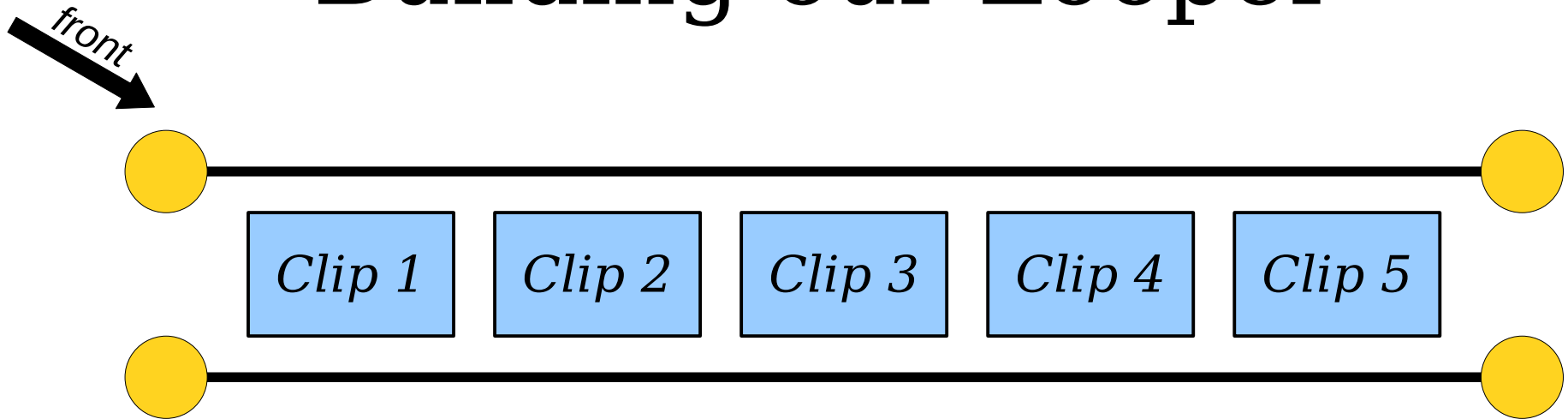
```
Queue<SoundClip> loop = loadLoop(/* ... */);
```

# Building our Looper



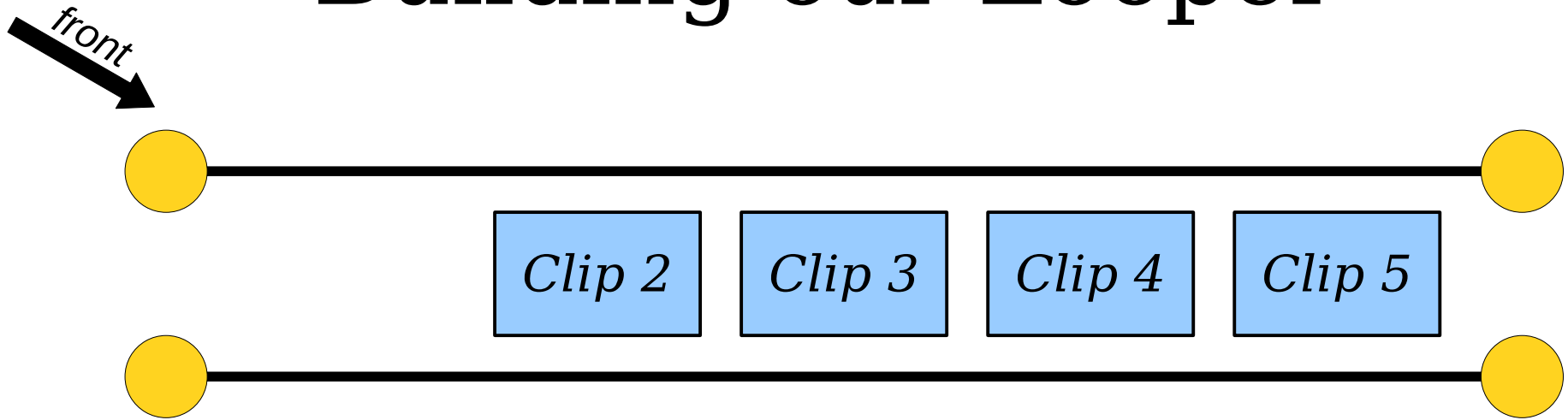
```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
  
  
  
  
}
```

# Building our Looper



```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
  
}
```

# Building our Looper



*Clip 1*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
  
}
```

# Building our Looper

front



The diagram shows a double-ended queue represented by two parallel horizontal black lines. At each end of both lines is a yellow circle. Between the lines, four light blue rectangular boxes are arranged horizontally, labeled 'Clip 2', 'Clip 3', 'Clip 4', and 'Clip 5' from left to right. An arrow labeled 'front' points towards the left end of the top line.

*Clip 1*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
  
}
```

# Building our Looper

front



*Clip 2*

*Clip 3*

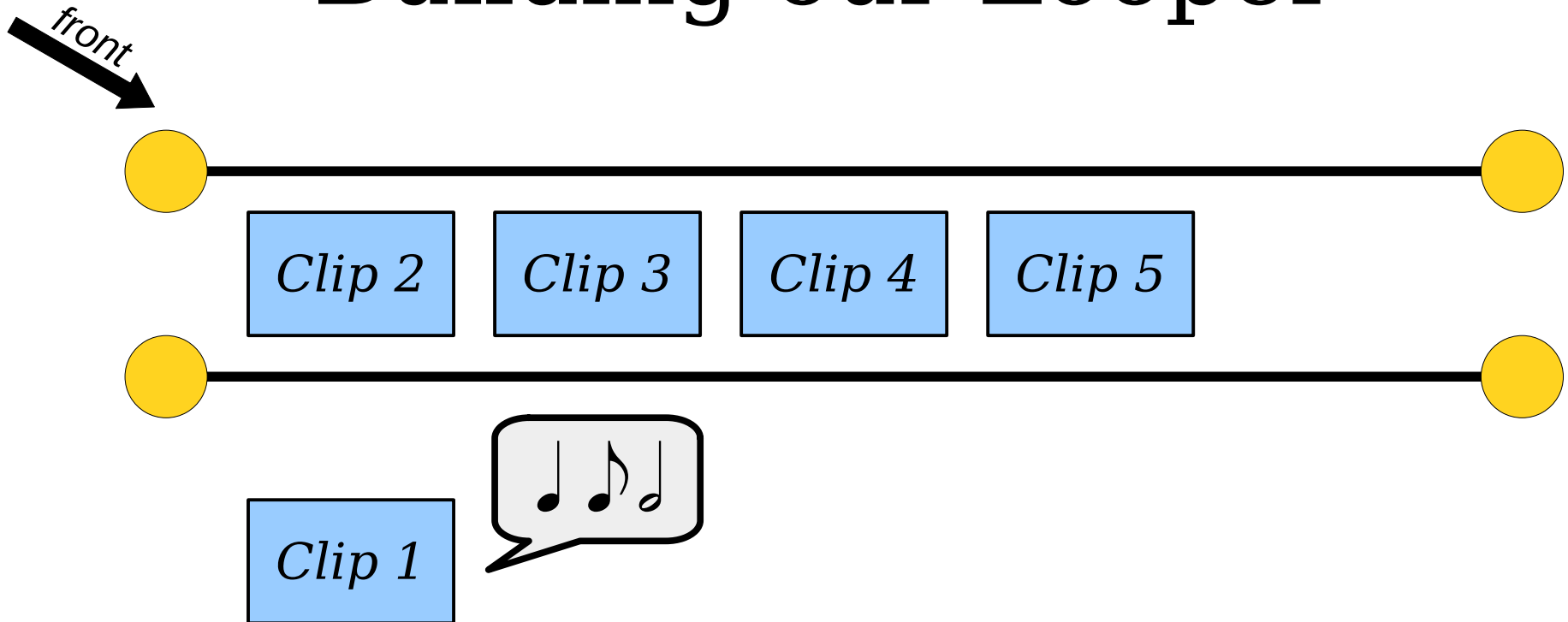
*Clip 4*

*Clip 5*

*Clip 1*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
}
```

# Building our Looper



```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
}
```

# Building our Looper

front



*Clip 2*

*Clip 3*

*Clip 4*

*Clip 5*

*Clip 1*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
}
```



# Building our Looper

front

*Clip 2*

*Clip 3*

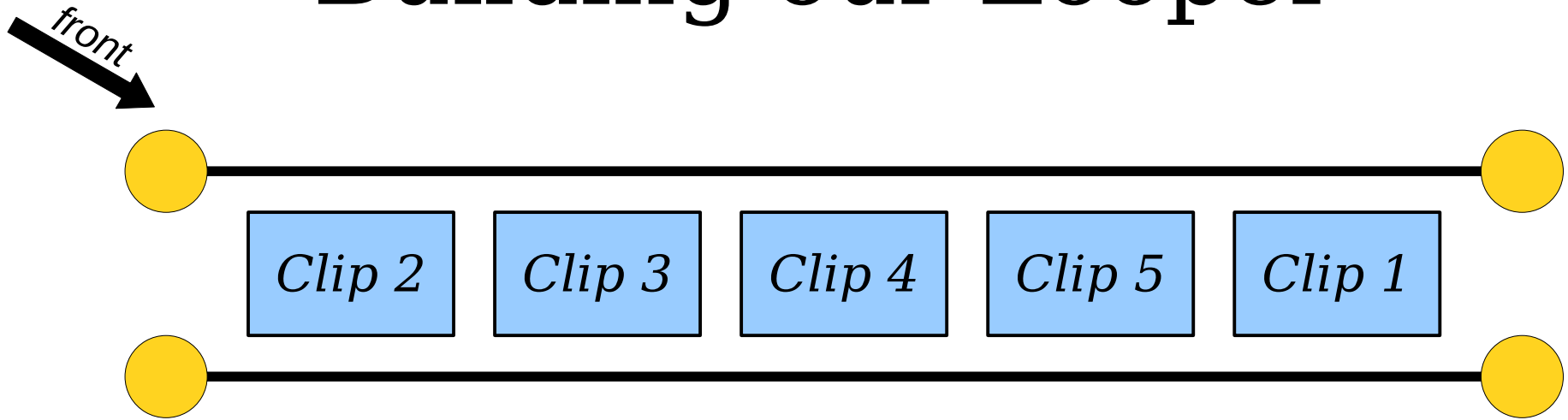
*Clip 4*

*Clip 5*

*Clip 1*

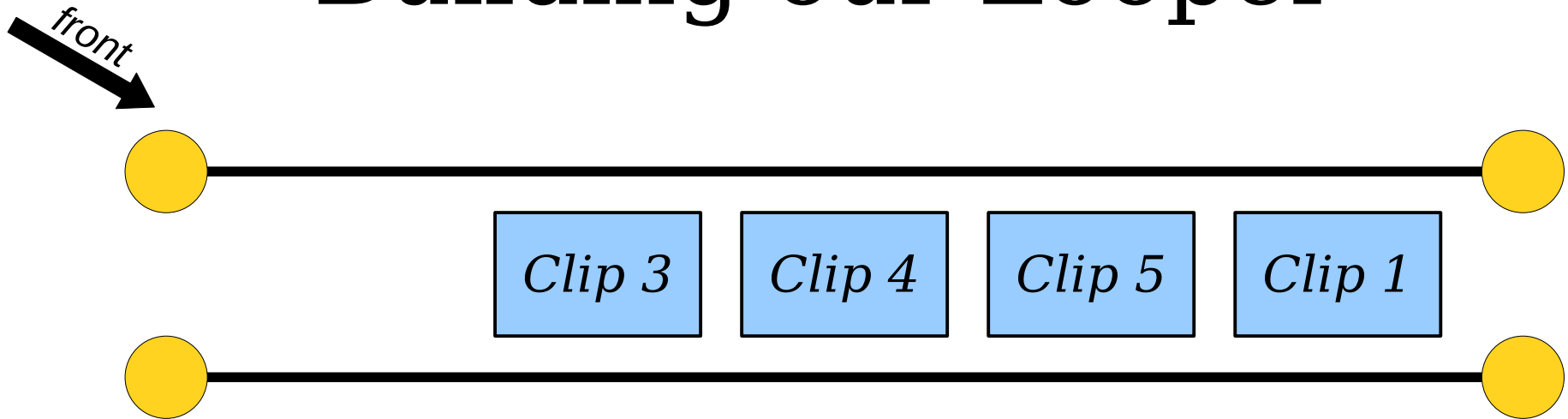
```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper



```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper



*Clip 2*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper

front



*Clip 3*

*Clip 4*

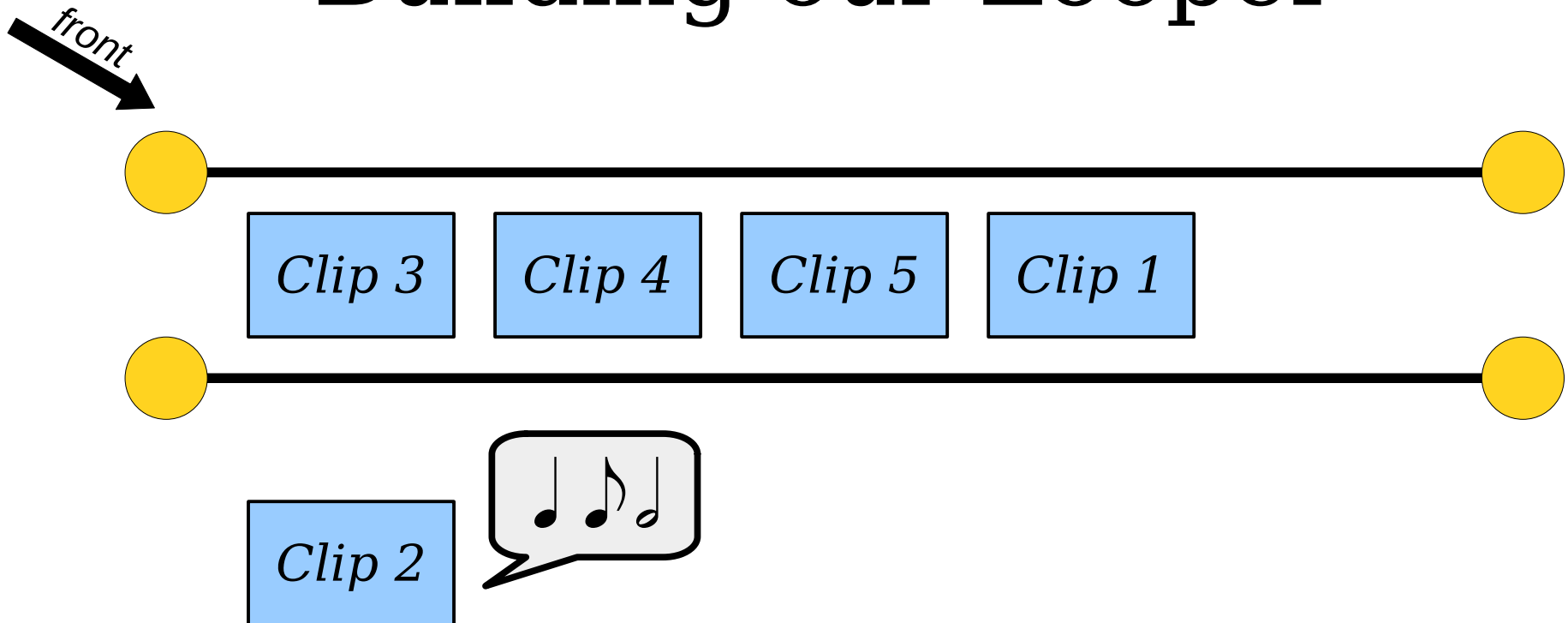
*Clip 5*

*Clip 1*

*Clip 2*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper



```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper

front

*Clip 3*

*Clip 4*

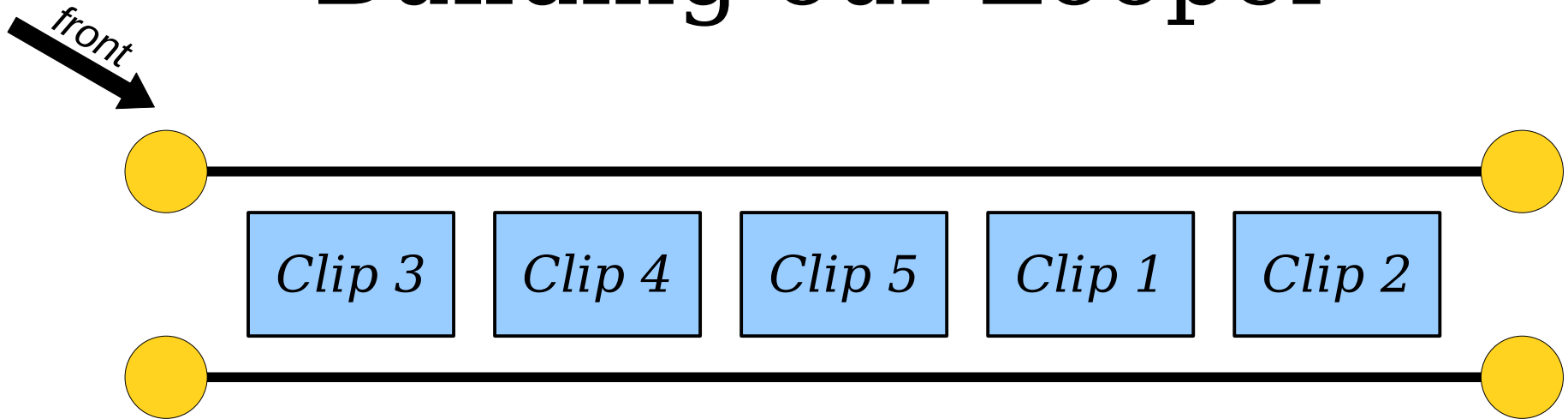
*Clip 5*

*Clip 1*

*Clip 2*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

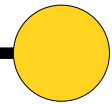
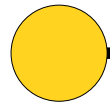
# Building our Looper



```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper

front

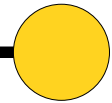
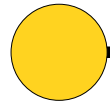


*Clip 4*

*Clip 5*

*Clip 1*

*Clip 2*



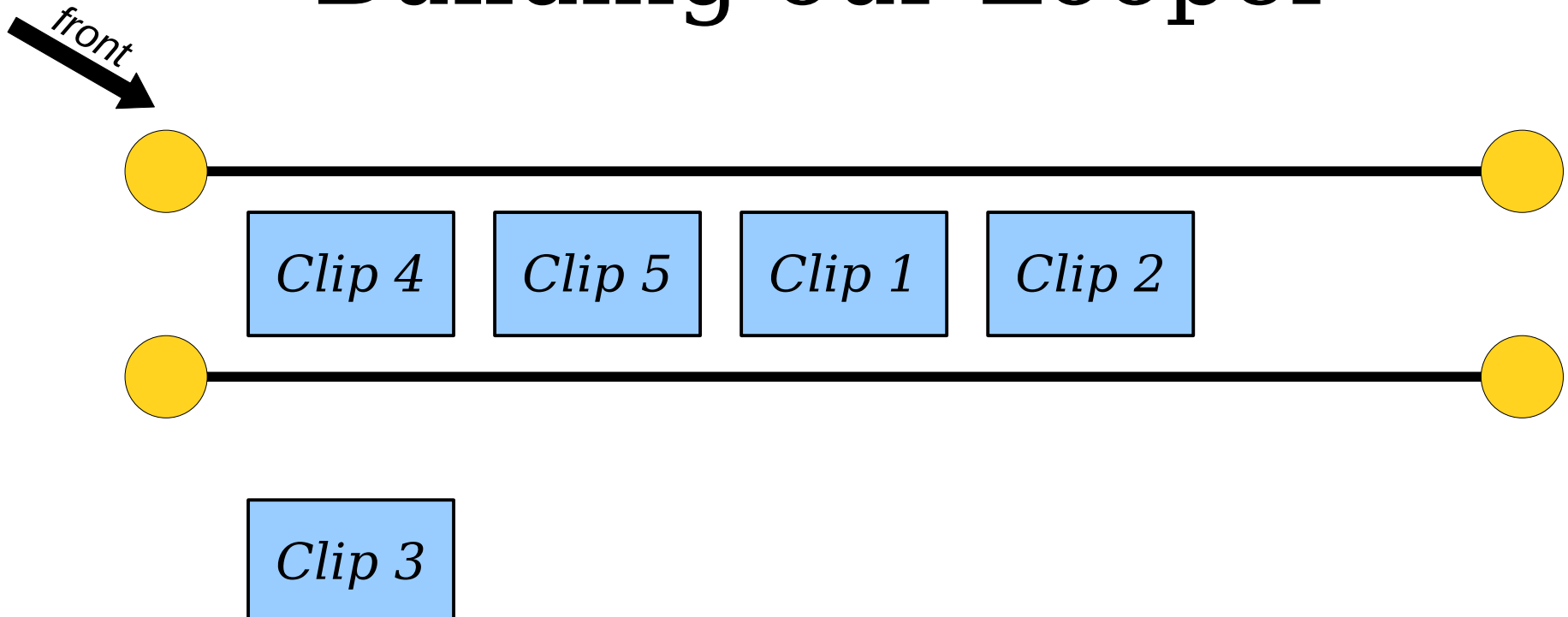
*Clip 3*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```



# Building our Looper

front



*Clip 4*

*Clip 5*

*Clip 1*

*Clip 2*

*Clip 3*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper

front



*Clip 4*

*Clip 5*

*Clip 1*

*Clip 2*

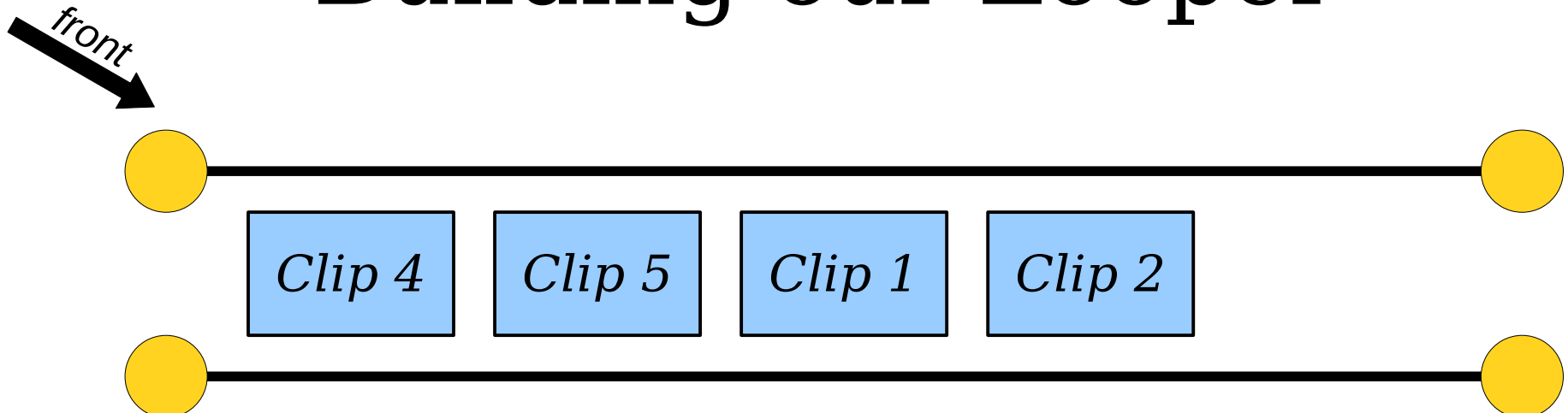
*Clip 3*



```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper

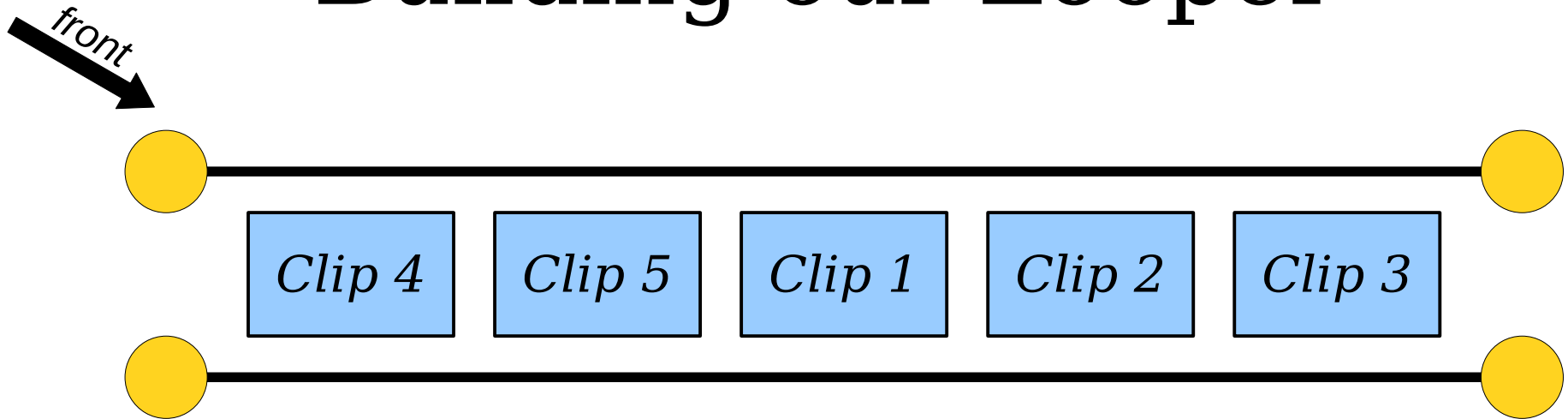
front

A diagram showing a loop structure. Two horizontal black lines represent the loop boundaries, each starting and ending with a yellow circle. Between these lines, four blue rectangular boxes are arranged horizontally, labeled 'Clip 4', 'Clip 5', 'Clip 1', and 'Clip 2' from left to right. An arrow labeled 'front' points from the top-left towards the 'Clip 4' box.

*Clip 3*

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Building our Looper



```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
    loop.enqueue(toPlay);  
}
```

# Enjoying Our Looper

Feeling musical? Want to contribute a loop for the next iteration of CS106B? Send me your .loop file and we'll add it to our collection!

# Changing our Looper

# Changing our Looper

```
Queue<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.dequeue();  
    playSound(toPlay.filename, toPlay.length);  
  
    loop.enqueue(toPlay);  
}
```

# Changing our Looper

```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
  
    loop.push(toPlay);  
}
```



# Changing our Looper

What are you going to hear when we use this version of the looper?

Formulate a hypothesis, but ***don't post anything in chat just yet.***

```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
  
    loop.push(toPlay);  
}
```

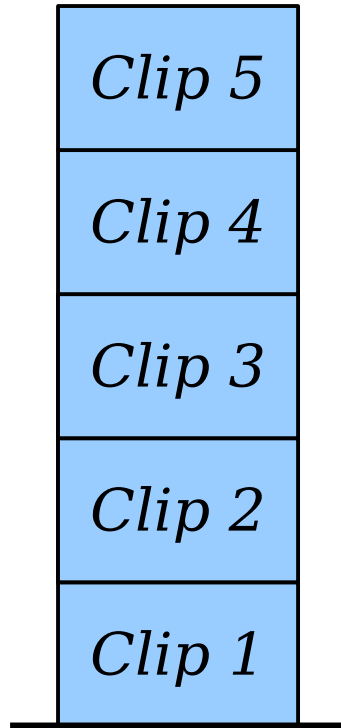
# Changing our Looper

What are you going to hear when we use this version of the looper?

Now, *private chat me your best guess*. Not sure?  
Just answer “??”

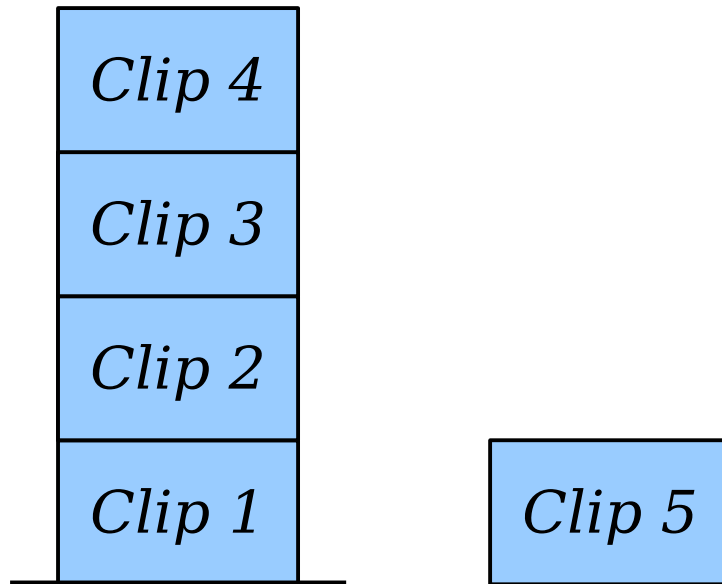
```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Changing our Looper



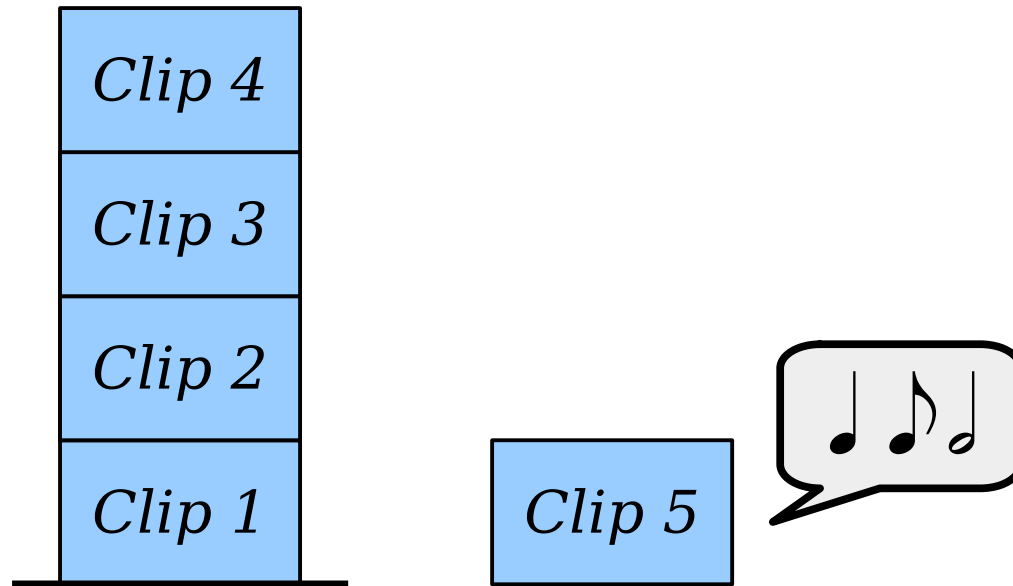
```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Changing our Looper



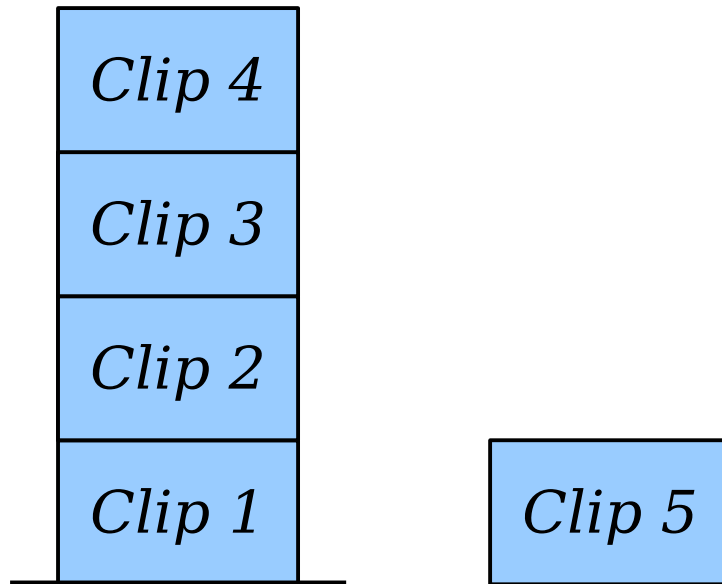
```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Changing our Looper



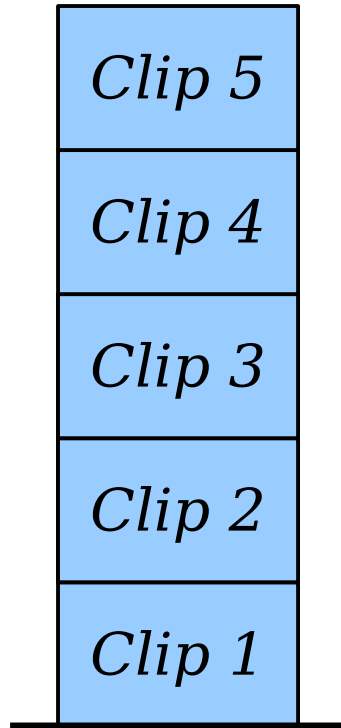
```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Changing our Looper



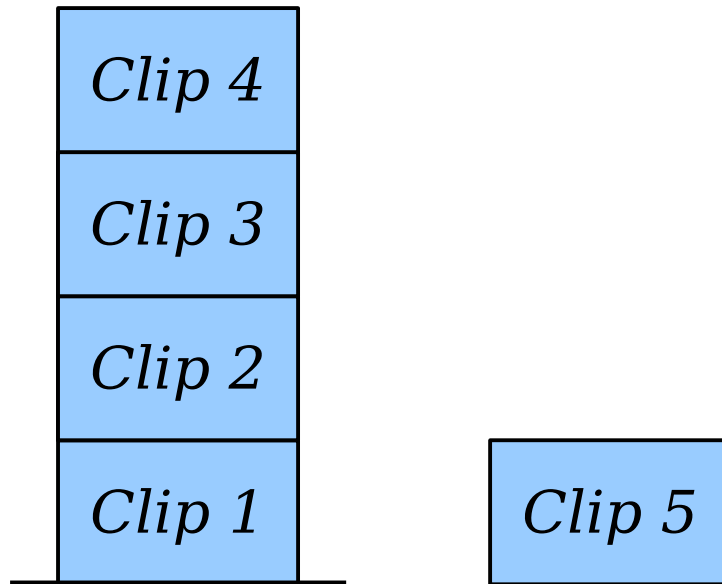
```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Changing our Looper



```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

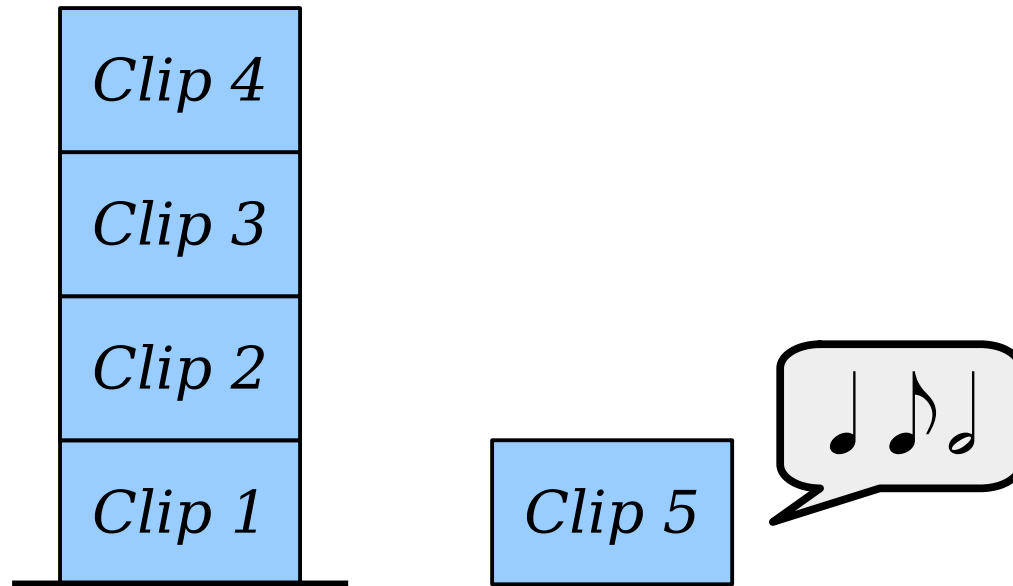
# Changing our Looper



```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

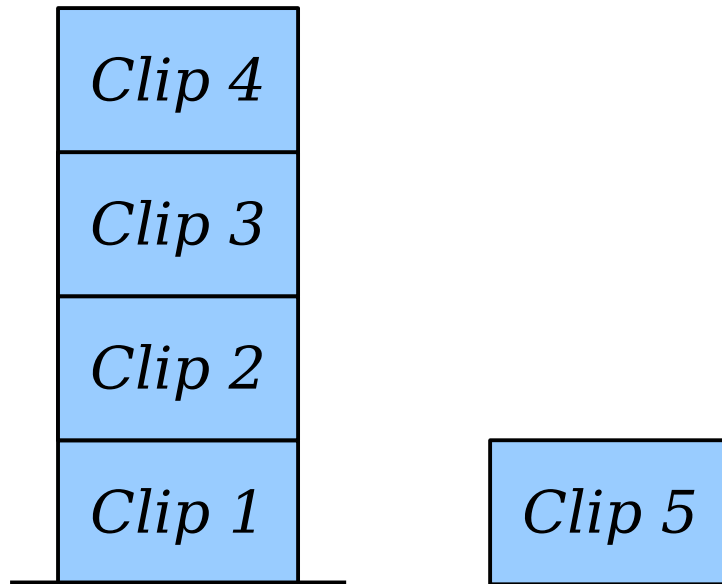


# Changing our Looper



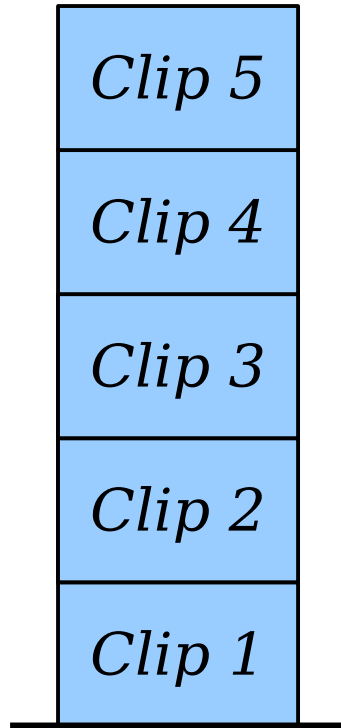
```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Changing our Looper



```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Changing our Looper



```
Stack<SoundClip> loop = loadLoop(/* ... */);  
while (true) {  
    SoundClip toPlay = loop.pop();  
    playSound(toPlay.filename, toPlay.length);  
    loop.push(toPlay);  
}
```

# Your Action Items

- ***Read Chapter 5.2 and 5.3.***
  - These sections cover more about the Stack and Queue type, and they're great resources to check out.
- ***Attend your first section!***
  - How exciting!
- ***Finish Assignment 1.***
  - Read the style guide up on the course website for more information about good programming style.
  - Review the Assignment Submission Checklist to make sure your code is ready to submit.

# Next Time

- ***Associative Containers***
  - Data sets aren't always linear!
- ***Maps and Sets***
  - Two ways to organize information.

